Algoritmer og datastrukturer

Mathilde Haukø Haugum

Dette er et kompendium for emnet TDT4120 Algoritmer og Datastruktur holdt ved NTNU høsten 2018. Kompendiet er basert på forelesningsnotatene og boka ”Introduction to Algorithms” av Cormen Leiserson, Rivest og Stein.

N T N U

Forelesning 1 – Problemer og algoritmer

Vi starter med fagfeltets grunnleggende byggesteiner, og skisserer et rammeverk for å tilegne seg resten av stoffet. Spesielt viktig er ideen bak induksjon og rekursjon, altså at vi kun trenger å se på siste trinn og kan anta at resten er på plass. Pensum er kapittel 1, kapittel 2 (2.1-2-2) og kapittel 3 (3.1). Læringsmålene er:

- Forstå bokas pseudokode-konvensjoner
- Kjenne egenskapene til random-access machine-modellen (RAM)
- Kunne definere problem, instans og problemstørrelse
- **Kunne definere asymptotisk notasjon, O, Ω, Θ, o og ω.**
- **Kunne definere best-case, average-case og worst-case**
- **Forstå løkkeinvarianter og induksjon**
- **Forstå rekursiv dekomponering og induksjon over delproblemer**
- Forstå INSERTION-SORT

Kapittel 1 – Rollen til algoritmer i utregning

Hva er algoritmer og hvorfor er det viktig å studere de? Dette kapittelet er en introduksjon til algoritmer og en motivasjon for faget.

1.1 Algoritmer

En **algoritme** er en veldefinert beregningsmetode som tar en eller flere verdier som input og produserer en eller flere verdier som output. Algoritmen er altså en sekvens av beregninger som omformer input til output. En algoritme kan også ses på som et verktøy for å løse spesifikke beregningsproblem, der problemet vil gi et ønsket forhold mellom input og output og algoritmen beskriver hvordan dette forholdet kan oppnås. For eksempel kan det hende vi ønsker å sortere en sekvens av tall etter økende rekkefølge. Vi begynner med å formelt definere **sorteringsproblemet**:

- **Input:** en sekvens av n tall: $\langle a_1, a_2, \dots, a_n \rangle$
- **Output:** en permutasjon (omordning) av input sekvensen slik at elementene ordnes i økende verdi: $\langle a'_1, a'_2, \dots, a'_n \rangle$

For eksempel kan input være $\langle 31, 41, 59, 26 \rangle$ slik at output blir $\langle 26, 31, 41, 59 \rangle$. En slik input sekvens kalles en **instans** av sorteringsproblemet. Et **instans av et problem** vil generelt bestå av inputen som trengs for å regne ut løsningen på problemet. Sortering er en viktig egenskap i datamaskiner, så det finnes mange sorteringsalgoritmer. Hvilken som er best avhenger av antall elementer som skal sorteres, hvor mye de allerede er sortert, osv. En algoritme er **korrekt**, dersom den gir riktig output for hver input instans. En korrekt algoritme vil altså løse det gitte beregningsproblemet.

Praktiske applikasjoner av algoritmer kan brukes i svært mange områder, for eksempel i kartleggingen av det menneskelige genomet, i søkeprogrammer på Internettet, osv. To egenskaper som er felles for mange algoritmeproblem er:

1. **De har mange mulige løsninger,** der de fleste egentlig ikke løser problemet. Det er en utfordring å finne en korrekt algoritme eller den ”beste”.
2. **De har praktiske bruksområder**. For eksempel vil det å finne den korteste ruten kunne brukes i en GPS, i Internett routing, osv.

Denne boken fokuserer mye på effektive algoritmer. Et vanlig mål på effektivitet er hastighet, altså hvor lang tid en algoritme bruker på å produsere resultatet. Det finnes likevel noen problemer der ingen effektive løsninger er kjent. Disse problemene kalles NP-complete og vi skal se på de i slutten av kompendiet.

Parallellisme

I dagens datamaskiner blir chips laget med flere prosessorer som kjører parallelt, slik at flere beregninger kan kjøres per sekund. For å få best mulig ytelse fra disse multikjerne datamaskinene, må vi designe algoritmer som tar utgangspunkt i parallellisme.

1.2 Algoritmer som en teknologi

Dersom datamaskiner blir uendelig raske og minnet blir gratis vil man kunne bruke enhver riktig metode for å løse et problem. Dette er likevel ikke tilfellet, og beregningstid og minneplass er begrensede ressurser. Derfor er vi på jakt etter algoritmer som er effektive med hensyn til tid og rom.

Effektivitet

Ulike algoritmer som er laget for å løse samme problem, kan ofte ha svært ulik effektivitet. Denne forskjellen kan være mye mer signifikant enn forskjeller pga maskin- og programvare. Vi skal se et eksempel på dette i kapittel 2, der vi ser på to ulike sorteringsalgoritmer. Den første kalles **innsettingssortering** (*insertion sort*) og bruker omtrent $c_1 n^2$ tid på å sortere n elementer (c_1 er en konstant som ikke avhenger av n). Den andre kalles **flettesortering** (*merge sort*) og bruker omtrent $c_2 n \log_2 n$ tid på å sortere n elementer. Som regel vil $c_1 < c_2$, og inputstørrelsen n vil vanligvis ha mye større påvirkning på kjøretiden. Vi kan se at forskjellen mellom disse er at innsettingssortering har en faktor n, mens flettesortering har en faktor $\log_2 n$ som er mye mindre. Innsettingssortering er som regel raskest (pga $c_1 < c_2$), men når n blir stor nok vil flettesortering bli raskere. For store inputstørrelser kan dette ha svært stor effekt på kjøretiden!

Algoritmer og andre teknologier

Vi bør se på algoritmer som en teknologi, på samme måte som maskinvaren til datamaskiner. Den totale ytelsen til et system vil avhenge av at de valgte algoritmene er effektive. Algoritmer er kjernen i de fleste teknologiene som brukes i moderne datamaskiner. Å ha en solid base i kunnskap og teknikk innenfor algoritmer er en av egenskapene som skiller en virkelig dyktig programmerer fra nybegynnere!

Kapittel 2 – Komme i gang

Dette kapitlet tar for seg rammeverket som brukes for å tenke på designet og analysen av algoritmer. Vi skal se på algoritmen for innsettingssortering som brukes for å løse sorteringsproblemet, og hvordan vi kan bruke en pseudokode for å vise hvordan vi skal spesifisere algoritmer. Når vi har spesifisert algoritmen kan vi argumentere for at den er korrekt og analysere kjøretiden. Analysen vil introdusere en notasjon som fokuserer på hvordan kjøretiden øker med antall enheter som skal sorteres.

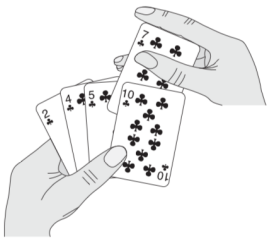
2.1 Innsettingssortering

Innsettingssortering algoritmen løser sorteringsproblemet vi så på i kapittel 1:

- **Input:** en sekvens av n tall: $\langle a_1, a_2, \dots, a_n \rangle$
- **Output:** en permutasjon (omordning) av input sekvensen slik at elementene ordnes i økende verdi: $\langle a'_1, a'_2, \dots, a'_n \rangle$

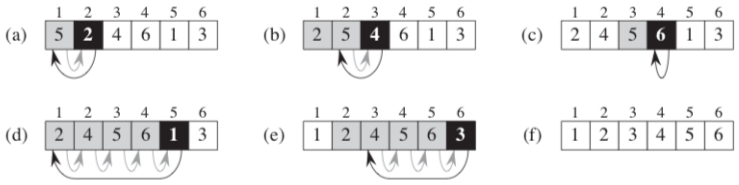
Tallene vi ønsker å sortere kalles også **nøklene** (*keys*), og input kommer som en array med n elementer. Vi skal beskrive algoritmer som programmer som er skrevet i en **pseudokode** som ligner på kode fra C, C++, Java, Python, osv. I motsetning til vanlig kode vil pseudokode bruke de meste presise metodene for å spesifisere algoritmen, vanlige setninger kan brukes for å forklare og den er ikke opptatt av problemer ved programvareskriving (eks: har ikke med error håndtering).

Innsettingssortering er en effektiv algoritme for å sortere et lite antall elementer. Det fungerer slik mange mennesker vil sortere en hånd med kort, altså ved å ta ett og ett kort med én hånd og plassere det i riktig rekkefølge i den andre hånden. For å finne riktig posisjon for et kort, vil vi sammenligne det med alle kortene som allerede er sortert, fra høyre til venstre (se figur). Vi presenterer pseudokoden for innsettingssortering som en prosedyre kalt INSERTION-SORT, som tar an array $A[1, \dots, n]$ med lengde n som parameter. Antall elementer i A er gitt som $A.length = n$. Algoritmen vil endre rekkefølgen på elementene i matrisen, slik at A vil inneholde den sorterte output sekvensen når INSERTION-SORT prosedyren er ferdig.



```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Pseudokoden for INSERTION-SORT prosedyren



Figuren over til høyre viser hvordan denne algoritmen fungerer for $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Indeksen j er "nåværende kort" som blir innsatt i den sorterte kortsekvensen $A[1, \dots, j-1]$. I begynnelsen av hver iterasjon av for-løkken vil submatrisen $A[1, \dots, j-1]$ bestå av elementer fra den opprinnelige matrisen, men i sortert rekkefølge. Disse egenskapene kalles **loop invariant,** som brukes for å forstå hvorfor en algoritme er korrekt. Vi må vise tre ting ved loop invarianten:

1. **Initialisering** – det er sant før den første iterasjonen av loopen
2. **Vedlikehold** – det er sant før en iterasjon av loopen, og fortsetter å være sant før neste iterasjon
3. **Terminering** – når loopen terminerer vil invarianten gi oss en nyttig egenskap som hjelper oss å vise at algoritmen er korrekt.

Når de to første egenskapene holder, vil loop invarianten være sann før alle iterasjoner av loopen. Første egenskap viser grunntilfellet, mens andre egenskap viser det induktive

Dette er forklart på en annen måte på side 15

steget. Den tredje egenskapen er kanskje den viktigste, fordi den sier noe om forholdet som gjør at loopen terminerer og at induksjonen dermed stopper. Vi ser på disse tre egenskapene for innsettingssortering:

1. **Initialisering** – vi skal se om loop invarianten holder før første iterasjon der $j = 2$. Submatrisen $A[1, \dots, j-1]$ vil da kun inneholde et enkelt element $A[1]$ som vil være det første elementet i den originale matrisen. Denne submatrisen vil derfor være sortert og loop invarianten holder før første iterasjon av loopen.
2. **Vedlikehold** – vi skal se om loop invarianten holder for alle iterasjonene. Loopen vil flytte elementet ved posisjon j mot venstre helt til den finner riktig posisjon, der den vil sette inn elementet. Submatrisen $A[1, \dots, j]$ vil dermed inneholde de originale elementene, men i sortert rekkefølge. Deretter vil j øke i neste iterasjon.
3. **Terminering** – betingelsen for loopen er: $j > A.length = n$. Siden j øker med en for hver iterasjon vil vi til slutt få $j = n + 1$. Dersom vi setter dette inn i formelen for submatrisen får vi at submatrisen er $A[1, \dots, n]$, altså vil hele matrisen ha blitt sortert. Dermed kan vi konkludere med at algoritmen er korrekt.

Pseudokode konvensjoner skumles

Vi bruker følgende konvensjoner i vår pseudokode:

- Innrykk indikerer blokk strukturer. For eksempel vil while loopen som begynner på linje 5 inneholde linjene 6-7, men ikke 8. Vi bruker altså ingen parenteser eller end utsagn, fordi det sparer plass og koden blir mer tydelig.
- Loop telleren vil beholde sin verdi etter loopen har blitt terminert. Vi bruker *to* når telleren øker til en verdi og *downto* når den synker til en verdi. Når den endres med mer enn 1 bruker vi *by* ….
- Variabler er lokale for den gitte prosedyren (ikke globale uten at det er oppgitt)
- Vi får tilgang til array elementer ved å se på array navnet etterfulgt av [i]: $A[i]$. $A[1, \dots, j]$ er en array som består av j elementer.
- Sammensatt data blir organisert i objekter, og vi kan få tilgang til en spesifikk attributt vha objekt.attributt, for eksempel $A.length$.
- Return vil med en gang overføre kontrollen tilbake til det kallende punktet i den kallende prosedyren. De fleste return utsagnene vil også sende en verdi tilbake til kalleren. Pseudokode lar oss returnere flere verdier i et enkelt return utsagn
- Boolean operatorene ”og” og ”eller” er kortsluttet. For ”x og y” vil dette bety at vi først evaluerer x og dersom denne er false vil ikke y evalueres fordi ”x og y” må være false. For ”x eller y”, vil y kun evalueres om x er false.
- error indikerer at en feil har oppstått fordi forholdene som kalte prosedyren var feile. Den kallende prosedyren er ansvarlig for å håndtere feil = vi trenger ikke å spesifisere hva som skal gjøres.

2.2 Analysering av algoritmer

Analyse av algoritmer går ut på å forutse ressursene som algoritmen krever, og som regel ser vi på kjøretiden som den viktigste ressursen. Før vi kan analysere en algoritme må vi ha en modell for implementasjonsteknologien som vi skal bruke, inkludert en modell for ressursene til den teknologien og deres kostnader. Vi antar at vi bruker en generisk, enprosessor kalt **RAM** (random-access maskin) modell, slik at algoritmene blir implementert som dataprogram og instruksjoner blir utført en etter en. RAM modellen inneholder instruksjoner for aritmetikk (addisjon, subtraksjon, divisjon, rest, osv.), data (load, lagre og kopiere) og kontroll (return, osv.). Disse instruksene vil ta en konstant

mengde tid (dvs. når en modell inneholder en instruks, kan vi anta at instruksen tar en konstant tid å gjennomføre). Datatypene i RAM er integer og flyttall. Det er også en begrensning på størrelsen til hvert ord med data. Ekte datamaskiner kan også inneholde flere instrukser, noe som representerer et grått område i RAM modellen.

Analyse av innsettingssortering

Hvor mye tid INSERTION-SORT prosedyren bruker avhenger av størrelsen til input og hvor sortert input er fra før av. Siden kjøretiden til en algoritme som regel øker med inputstørrelsen, er det vanlig å beskrive **kjøretiden som en funksjon av inputstørrelsen**. Vi skal se nærmere på disse to begrepene:

- **Inputstørrelse** – kan være mye forskjellig, for eksempel ved sortering er det antall enheter i inputen (dvs. array størrelsen). For multiplikasjon av to integer verdier er inputstørrelsen antall bits som trengs for å representere input med en ordnet, binær notasjon. Denne størrelsen må derfor oppgis ved hvert problem.
- **Kjøretiden** – antall primitive operasjoner eller "steg" som blir utført. Foreløpig ser vi på situasjonen der **hver linje i pseudokoden krever en konstant mengde tid for å utføres**. En linje kan ta mer tid enn en annen, men vi antar at hver utføring av linje i tar tiden c_i, der c_i er en konstant. Dette synspunktet er i tråd med RAM modellen og reflekterer hvordan pseudokoden vil implementeres i de fleste datamaskinene.

Vi skal nå se hvordan vi kan bestemme kjøretiden til INSERTION-SORT prosedyren. Vi begynner med å se på **tidskostnaden for hver påstand og antall ganger hver påstand blir utført**. Vi lar t_j være antall ganger while loop testen (dvs. linje 5) blir utført for denne verdien av j. Når en for eller while loop blir terminert vil testen utføres en gang mer enn loop kroppen. Vi antar at kommentarer ikke er utførbare påstander som derfor ikke bruker noe tid.

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

Her kan vi se at hver linje har en tidskostnad c_i og et antall ganger linjen blir utført. Legg merke til at loop kroppene blir utført en gang mindre enn loop testene.

Kjøretiden til en algoritme vil være summen av kjøretidene til hver påstand som blir utført. En påstand som tar c_i steg å utføre og utføres n ganger vil bidra med $c_i n$ til den totale kjøretiden. Vi finner kjøretiden til INSERTION-SORT med en input på n verdier ($T(n)$) ved å summere produktene av *cost* og *times* kolonnene på figuren over:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Selv om inputstørrelsen (n) er konstant, kan kjøretiden variere avhengig av hvilken input som gis. For eksempel for INSERTION-SORT vil **best-case** være at matrisen

allerede er sortert, fordi da vil vi få at $A[i] \leq key$ for hvert element og while-løkken vil ikke utføres. Dette gjør at $t_j = 1$ for alle j, og best-case kjøretiden blir:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + +c_8(n-1)$$

Dette uttrykket kan skrives på formen $an + b$, der a og b er konstanter som avhenger av c_i. Dette betyr at $T(n)$ er en **lineær funksjon** av n. Worst-case er at elementene i matrisen er ordnet i motsatt rekkefølge. Da må hvert element $A[j]$ sammenlignes med hele den sorterte submatrisen $A[1, \dots, j-1]$, slik at $t_j = j$. Dersom vi bruker formlene på figuren til høyre får vi at:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

Dette uttrykket kan skrives på formen $an^2 + bn + c$, der a, b og c er konstanter som avhenger av c_i. Dette betyr at $T(n)$ er en **kvadratisk funksjon** av n.

Worst-case analyse

I avsnittet ovenfor så vi på både best-case og worst-case scenarioet. Som regel fokuserer vi på **worst-case kjøretid**, som er den lengste kjøretiden for enhver inputstørrelse n. Det er tre grunner for dette:

- Worst-case kjøretid for en algoritme gir oss den øvre grensen på kjøretiden for enhver input. Vi får en garanti på at algoritmen aldri vil ta lengre tid.
- For noen algoritmer vil worst-case forekomme relativt ofte. For eksempel ved søk hender det ofte at det man søker ikke er tilstede i mengden.
- Average-case er ofte nesten like dårlig som worst-case.

I noen tilfeller er vi heller interessert i average-case kjøretid og bruken av sannsynlighetsanalyse på algoritmer. Dette kan være utfordrende, siden det ofte er vanskelig å se hva som utgjør "gjennomsnittlig" input for et bestemt problem. Ofte antar vi at alle input av en gitt størrelse er like sannsynlig. Noen ganger kan vi bruke en randomisert algoritme som tar tilfeldige valg for å tillate en sannsynlighetsanalyse og gi en forventet kjøretid (mer kapittel 5).

Vekstraten

Når vi analyserte kjøretiden til INSERTION-SORT, valgte vi å ignorere den faktiske tidskostnaden ved hver påstand ved å representere alle kostnadene med c_i. Deretter valgte vi å ignorere de abstrakte kostnadene ved å heller bruke konstantene a, b og c. **Vekstraten** eller **vekstordenen** er en videre forenkling av kjøretiden, der vi kun ser på det ledende begrepet i formelen (eks: an^2), siden begrep med lavere orden er relativt ikke-signifikante for store verdier av n. Vi vil også ignorere den konstante koeffisienten til det ledende begrepet, siden konstante faktorer er mindre signifikante sammenlignet med vekstraten når man skal beregne effektivitet for stor n. For worst-case innsettingssortering vil vekstraten derfor være n^2, og vi sier at innsettingssortering har en worst-case kjøretid på $\Theta(n^2)$. Ved stor n vil den mest effektive algoritmen som regel være den som har worst-case vekstrate med lavest orden. Ved liten n kan det være motsatt, siden da vil også konstanter og begrep med lavere orden få betydning.

Kapittel 3 – Funksjonsvekst

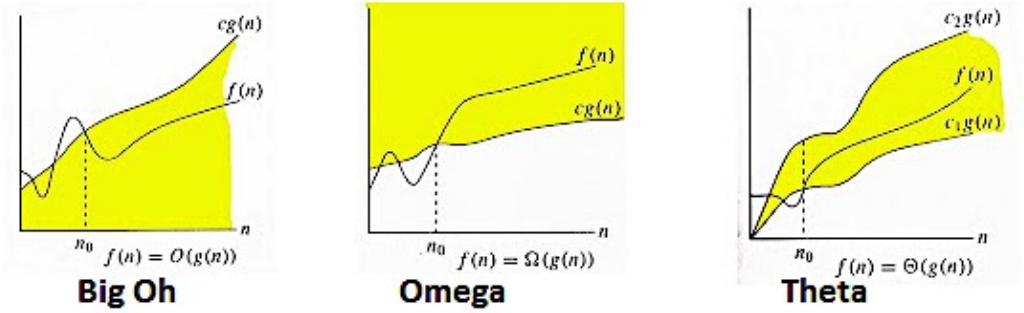
Vekstraten til algoritmens kjøretid gir en enkel karakterisering av algoritmens effektivitet og lar oss sammenligne de relative ytelsene til alternative algoritmer. For eksempel når n blir stor nok vil en algoritme med worst-case kjøretid $\Theta(n \lg n)$ være raskere enn en annen algoritme med kjøretid $\Theta(n^2)$. Når vi lar inputstørrelsen bli så stor at kun vekstraten til kjøretiden blir relevant, sier vi at vi studerer den **asymptotiske** effektiviteten til algoritmer. Altså, vi ser på hvordan kjøretiden til en algoritme øker med størrelsen til input i grensen (dvs. hvordan Θ påvirkes av økende n). En algoritme som er asymptotisk mer effektiv, vil som regel være det beste for alt bortsett fra små input.

3.1 Asymptotisk notasjon

Notasjonen vi bruker for å beskrive asymptotisk kjøretid er definert som funksjoner der domenet er et sett av naturlige tall, $\mathbb{N} = \{0, 1, 2, \dots\}$. Slik notasjon er nyttig for å beskrive worst-case kjøretidsfunksjonen, $T(n)$. Det kan også hende at vi bruker et domene med reelle tall eller begrenser det til kun bestemte naturlige tall.

Asymptotisk notasjon, funksjoner og kjøretider

Vi bruker asymptotisk notasjon for å beskrive kjøretiden til algoritmer, for eksempel at worst-case kjøretid til innsettingssortering er $\Theta(n^2)$. Asymptotisk notasjon vil likevel egentlig gjelde funksjoner. Det vi skriver som $\Theta(n^2)$ er egentlig funksjonen $an^2 + bn + c$, som i tilfellet med innsettingssortering viser seg å karakterisere worst-case kjøretid. Funksjonene som vi bruker asymptotisk notasjon på vil som regel karakterisere kjøretiden til algoritmene. Asymptotisk notasjon kan også brukes på funksjoner for å karakterisere andre aspekter ved algoritmen, for eksempel hvor mye rom de bruker. Figuren nedenfor viser de asymptotiske notasjonene vi skal se på: Θ, O og Ω.

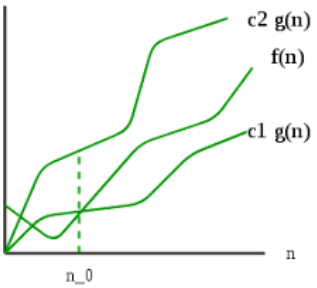


Θ-notasjon – asymptotisk øvre og nedre grense

Vi har sett at worst-case kjøretiden til innsettingssortering er $T(n) = \Theta(n^2)$, og skal nå se hva denne notasjonen betyr. For en gitt funksjon $g(n)$, definerer vi følgende:

$$\boldsymbol{\Theta(g(n)) = \{f(n)}\text{: } \textbf{det eksisterer positive konstanter } \boldsymbol{c_1, c_2} \textbf{ og } \boldsymbol{n_0} \textbf{ slik at } \boldsymbol{0 \le c_1 g(n) \le f(n) \le c_2 g(n)} \textbf{ for alle } \boldsymbol{n \ge n_0}\boldsymbol{\}}$$

En funksjon $f(n)$ vil altså høre til settet $\Theta(g(n))$ dersom det eksisterer positive konstanter som gjør at $f(n)$ kan bli "sandwiched" mellom $c_1 g(n)$ og $c_2 g(n)$, for



tilstrekkelig stor n (se figur). Siden $\Theta(g(n))$ er et sett vil vi skrive $f(n) \in \Theta(g(n))$ for å indikere at $f(n)$ er et medlem av settet, men vi bruker ofte notasjonen $f(n) = \Theta(g(n))$. Vi sier at $f(n)$ er lik $g(n)$ innenfor en konstant faktor, og at $g(n)$ er en **asymptotisk tett grense** for $f(n)$. Asymptotisk tett grense betyr altså at det er en øvre og en nedre grense. Definisjonen over krever at ethvert medlem $f(n) \in \Theta(g(n))$ er **asymptotisk ikke-negativ**, som vil si at $f(n)$ ikke er negativ når n er tilstrekkelig stor. Vi kan anta at alle funksjoner som brukes innenfor Θ-notasjon er ikke-negative.

Vi skal nå se på et eksempel der vi finner at $f(n) = \frac{1}{2}n^2 - 3n$ vil ha kjøretid $\Theta(n^2)$, altså $g(n) = n^2$. For å vise dette må vi bruke at vi kan neglisjere ledd av lavere orden og koeffisienter til ledd med høyeste orden. Vi begynner med å finne positive konstanter c_1, c_2 og n_0 slik at:

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

for alle $n \geq n_0$. Dersom vi deler på n^2 får vi:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Høyre ulikhet vil holde for enhver verdi av $n \geq 1$ ved å velge $c_2 \geq 1/2$. Venstre ulikhet vil holde for enhver verdi av $n \geq 7$ ved å velge $c_1 \leq 1/14$. Dersom vi velger $c_1 = 1/14$, $c_2 = 1/2$ og $n_0 = 7$ kan vi derfor vise at $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Det er altså nok å vise at *noen* valg eksisterer. Vi kan også bruke den formelle definisjonen for å vise $6n^3 \neq \Theta(n^2)$. Vi har at høyre ulikhet blir $6n^3 \leq c_2 n^2$. Vi deler på $6n^2$ og får at $n \leq c_2/6$. Denne ulikheten kan umulig holde for vilkårlig stor n, siden c_2 er konstant.

Lavere-ordens ledd i en asymptotisk positiv funksjon kan ignoreres fordi de blir ikke-signifikante for store n. Ved å sette c_1 til en verdi som er litt lavere enn koeffisienten til høyeste-ordens ledd og c_2 til en verdi som er litt høyere, vil ulikhetene i den formelle definisjonen fortsatt bli tilfredsstilt. Koeffisienten til høyeste-ordens ledd kan ignoreres, siden det vil kun endre c_1 og c_2 med en konstant faktor lik koeffisienten. Generelt for ethvert polynom: $p(n) = \sum_{i=0}^{d} a_i n^i$, der a_i er konstanter og $a_d > 0$ vil:

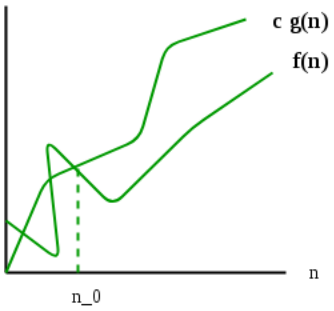
$$p(n) = \Theta(n^d)$$

En konstant er en 0-grad polynom, så vi kan uttrykke en konstant funksjon som $\Theta(n^0) = \Theta(1)$. Denne notasjonen brukes ofte, selv om den ikke indikerer hvilken variabel som går mot uendelig.

O-notasjon – asymptotisk øvre grense

Vi bruker O-notasjon når vi har kun en **øvre grense** på funksjonen:

$$O(g(n)) = \{f(n): \text{det eksisterer positive konstanter } c \text{ og } n_0 \text{ slik at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$



Vi bruker altså O-notasjon for å gi en asymptotisk øvre grense for en funksjon, innenfor en konstant faktor n. På figuren kan vi se at $cg(n)$ alltid er større eller lik $f(n)$ når $n \geq n_0$. $f(n) = O\big(g(n)\big)$ indikerer at funksjonen er et medlem av settet $O\big(g(n)\big)$.

Merk: **dersom $\boldsymbol{f(n) = \Theta(n)}$ må også $\boldsymbol{f(n) = O(n)}$** siden Θ er en sterkere notasjon enn O. Siden hver kvadratiske funksjon $an^2 + bn + c$ har kjøretid $\Theta(n^2)$ vil det derfor bety at de også har kjøretid $O(n^2)$. Alle lineære funksjoner $an + b$ vil være i $O(n)$, og dermed også i $O(n^2), O(n^3)$, osv. Dette skyldes at $O(n) \subset O(n^2)$, altså $O(n)$ er en asymptotisk tettere grense. **Dersom kjøretiden er $\boldsymbol{O(n)}$, er det ikke galt å si at den er $\boldsymbol{O(n^2)}$, men det er et svakere resultat og en løsere grense.** Dette gjelder også for Ω, men motsatt.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted
           sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Merk: nested loops = $O(n^2)$

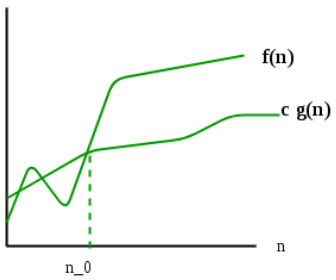
Vha O-notasjon kan man raskt finne kjøretiden til en algoritme ved å studere dens struktur (i hovedsak looper og rekursive elementer). For eksempel vet vi at innsettingssorteringsalgoritmen har en dobbel nøstet loop struktur (se figur) og derfor vet vi at worst-case kjøretid er $O(n^2)$. Kostnaden for hver iterasjon av den indre loopen er $O(1)$, dvs. konstant uavhengig av n. j vil ha maksimum n verdier, og for hver av disse vil i ha maksimum n verdier. Den indre loopen vil utføres maks én gang for hver kombinasjon av i og j, altså vil den utføres $n * n = n^2$ ganger. **$\boldsymbol{O}$-notasjon for worst-case gir en øvre grense, slik at $\boldsymbol{O(n^2)}$ vil gjelde for alle input**. Merk: $\Theta(n^2)$ for worst-case gir også en nedre grense, og vil derfor ikke gjelde for alle input. For eksempel når listen er sortert vil innsettingssortering gi kjøretid på $\Theta(n)$, siden indre loop ikke kjøres (se figur).

At $f(n) = O(n^2)$ er egentlig misbruk av notasjon, fordi for en gitt n kan kjøretiden variere. Men vi tolker denne notasjonen som om at worst-case kjøretid er $O(n^2)$

Ω-notasjon - asymptotisk nedre grense

Vi bruker Ω-notasjon når vi har kun en **nedre grense** på funksjonen:

$$\boldsymbol{\Omega\big(g(n)\big) = \{f(n)}\text{: }\textbf{det eksisterer positive konstanter } \boldsymbol{c}\textbf{ og }\boldsymbol{n_0}\textbf{ slik at } \boldsymbol{0 \leq cg(n) \leq f(n)}\textbf{ for alle }\boldsymbol{n \geq n_0}\boldsymbol{\}}$$



Vi bruker altså Ω-notasjon for å gi en asymptotisk nedre grense for en funksjon, innenfor en konstant faktor n. På figuren kan vi se at $f(n)$ alltid er større eller lik $cg(n)$ når $n \geq n_0$. $f(n) = \Omega\big(g(n)\big)$ indikerer at funksjonen er et medlem av settet $\Omega\big(g(n)\big)$.

Vi får følgende teorem:

<u>Teorem 3.1</u>
For to funksjoner $\boldsymbol{f(n)}$ og $\boldsymbol{g(n)}$ har vi at $\boldsymbol{f(n) = \Theta\big(g(n)\big)}$ hvis og bare hvis $\boldsymbol{f(n) = O\big(g(n)\big)}$ og $\boldsymbol{f(n) = \Omega\big(g(n)\big)}$

For eksempel har vi sett at $an^2 + bn + c = \Theta(n^2)$, noe som krever at $an^2 + bn + c = \Omega(n^2)$ og $an^2 + bn + c = O(n^2)$. Vi kan bruke teoremet over for å vise asymptotisk tette grenser (dvs. Θ) fra asymptotisk øvre (O) og nedre (Ω) grenser.

Ω-notasjon gir en nedre grense for best-case kjøretid, på samme måte som O-notasjon gir en øvre grense for worst-case kjøretid. Altså: Ω: lavest mulig kjøretid, mens O: høyest mulig kjøretid.

For eksempel vil best-case kjøretid til innsettingssortering være $\Omega(n)$ når listen allerede er sortert. Altså vil kjøretiden til innsettingssortering være både $\Omega(n)$ og $O(n^2)$. Disse grensene er så asymptotisk tett som mulig. Man kan si at worst-case kjøretid er $\Omega(n^2)$, men kjøretiden er ikke $\Omega(n^2)$ siden det finnes et input som gir $\Theta(n)$.

Asymptotisk notasjon i ligninger og ulikheter

Når en asymptotisk notasjon står alene på høyre side av ligningen, eks: $n = O(n^2)$, betyr dette at $n \in O(n^2)$. Når en asymptotisk notasjon dukker opp i en formel, eks: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$, betyr det at den representerer en anonym funksjon som vi ikke gidder å nevne. Dersom vi er interessert i den asymptotiske oppførselen (dvs. når n øker), er det ikke vits å spesifisere ledd med lavere orden, og dermed kan vi heller bruke $\Theta\big(g(n)\big)$.
Det er et skille mellom situasjoner der den asymptotiske notasjonen er på høyre og venstre siden. For eksempel kan vi se på:

$$1: \quad 2n^2 + 3n + 1 = 2n^2 + \Theta(n)$$
$$2: \quad 2n^2 + \Theta(n) = \Theta(n^2)$$

Ligning 1 sier at det eksisterer *en* funksjon $f(n) \in \Theta(n)$ slik at $2n^2 + 3n + 1 = 2n^2 + f(n)$ for alle n. I dette tilfellet vil $f(n) = 3n + 1$, som er i $\Theta(n)$. Ligning 2 sier at *for enhver* funksjon $g(n) \in \Theta(n)$ vil det være en funksjon $h(n) \in \Theta(n^2)$ slik at $2n^2 + g(n) = h(n)$. Denne tolkningen gir totalt at $2n^2 + 3n + 1 = \Theta(n^2)$.

o-notasjon – asymptotisk, ikke-tett øvre grense

Den asymptotiske øvre grensen gitt av O-notasjon kan være asymptotisk tett eller ikke (merk: asymptotisk tett = nøyaktig grense for store n). For eksempel vil grensen $2n^2 = O(n^2)$ være asymptotisk tett, mens grensen $2n = O(n^2)$ er ikke det. ***o*-notasjon brukes for å betegne en øvre grense som ikke er asymptotisk tett**:

$$\boldsymbol{o\big(g(n)\big) = \{f(n): \text{for alle konstanter } c > 0 \text{ vil det eksistere } n_0 > 0 \text{ slik at } 0 \leq f(n) < cg(n) \text{ for alle } n \geq n_0\}}$$

Merk: det er $<$ og ikke $\leq$! For eksempel vil $2n = o(n^2)$ og $2n^2 \neq o(n^2)$, fordi siste nevnte er en likhet. Hovedforskjellen er at ***o*-notasjon holder for alle konstanter $c > 0$, mens *O*-notasjon holder for noen konstanter $c > 0$**. I o-notasjon vil $f(n)$ bli insignifikant relativt til $g(n)$ når n går mot uendelig:

$$\log_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

ω-notasjon – asymptotisk, ikke-tett nedre grense

På tilsvarende måte blir ω-notasjon brukt for å betegne en nedre grense som ikke er asymptotisk tett:

$$\boldsymbol{\omega\big(g(n)\big) = \{f(n): \text{for alle konstanter } c > 0 \text{ vil det eksistere } n_0 > 0 \text{ slik at } 0 \leq cg(n) < f(n) \text{ for alle } n \geq n_0\}}$$

Altså den asymptotiske nedre grensen gitt av Ω-notasjon kan være asymptotisk tett eller ikke, mens den asymptotiske nedre grensen gitt av ω-notasjon vil alltid være ikke-tett. For eksempel vil $n^2/2 = \omega(n)$ og $n^2/2 \neq \omega(n^2)$. I ω-notasjon vil $g(n)$ bli insignifikant relativt til $f(n)$ når n går mot uendelig:

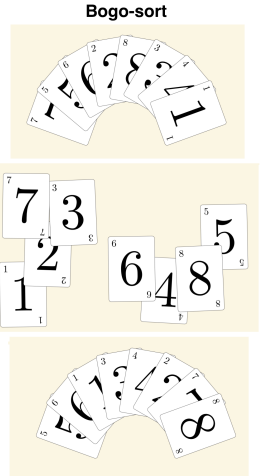
$$\log_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Regler for asymptotisk notasjon

Følgende gjelder for asymptotisk notasjon

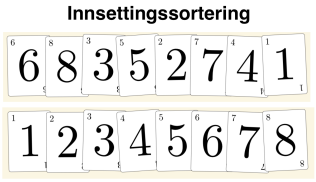
- Transitivitet: $f(n) = \Theta(g(n))$ og $g(n) = \Theta(h(n))$ gir at $f(n) = \Theta(h(n))$. Dette gjelder også for O, Ω, o og ω.
- Refleksivitet: $f(n) = \Theta(f(n))$, $f(n) = O(f(n))$ og $f(n) = \Omega(f(n))$
- Symmetri: $f(n) = \Theta(g(n))$ hvis og bare hvis $g(n) = \Theta(f(n))$
- Transpose symmetri: $f(n) = O(g(n))$ hvis og bare hvis $g(n) = \Omega(f(n))$. Det samme gjelder for : o og ω. Vi har at:
  - $f(n) = O(g(n)) : : a \leq b$
  - $f(n) = \Omega(g(n)) : : a \geq b$
  - $f(n) = \Theta(g(n)) : : a = b$
  - $f(n) = o(g(n)) : : a < b$ ($f(n)$ er asymptotisk mindre enn $g(n)$)
  - $f(n) = \omega(g(n)) : : a > b$($f(n)$ er asymptotisk større enn $g(n)$)

Forelesning 1



Når vi skal sortere noen kort kan en alternativ fremgangsmåte være å skyfle dem rundt og se om vi får en sortert samling. Dette er en brute-force tilnærming som kalles bogo-sort, og det er ikke noen særlig god løsning. I sortering av kort er det kanskje ikke et så viktig problem, men hva om vi for eksempel vil finne flest mulig par med kompatible donorer og resipienter ved for eksempel nyretransplantasjon? En bedre løsning vil bety flere liv som kan reddes!

Når antall elementer som skal sorteres eller behandles på annen måte øker, vil brute-force metoder bli ineffektive, og det er bedre med smartere algoritmer. Mange viktige problemer krever algoritmiske løsninger, og forskjellen på gode og dårlige løsninger er i kosmisk skala.



Innsettingssortering

Vi begynner med å se på bare de første kortene også sorterer vi oss gradvis mot høyre. Vi utnytter strukturen i problemet, nemlig at det er mulig å sortere litt etter litt. Dette ignoreres helt i en brute-force løsning der alle kortene sorteres på en gang. Innsettingssortering er hele tiden kun interessert i å sette inn neste kort og bygger dermed trinnvis på forrige del-løsning. Dermed slipper den å starte fra scratch hele tiden, slik som i en brute-force løsning. For å kunne påstå at innsettingssortering er bedre enn brute-force må vi benytte oss av asymptotisk notasjon.

Asymptotisk notasjon

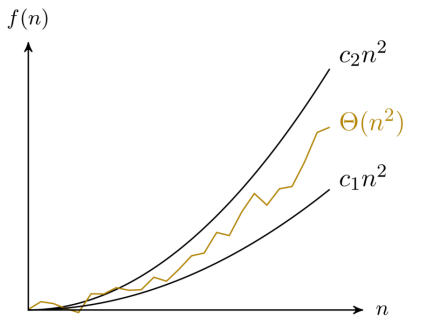
Enkle instruksjoner, slik som aritmetikk (multiplikasjon, addisjon, osv.), flytting av data og programkontroll tar konstant tid. Vi kan håndtere både heltall og flyttall, men vi antar som regel at heltallene er maks $c \lg n$ for en $c \geq 1$. Vi har følgende begrep:

- Problem – relasjon mellom input og output
- Instans – en bestemt input
- Problemstørrelse (n) – lagringsplass som trengs for en instans

Kjøretiden er en funksjon av problemstørrelsen (n), og større problemer krever mer tid. Vi er interessert i hvor fort kjøretiden vokser, altså en veldig grov "størrelsesorden". Derfor bruker vi asymptotisk notasjon, der konstanter og lavere ordens ledd blir droppet. Figuren til høyre viser Θ-notasjonen til noen funksjoner. Vi kunne ha brukt uttrykket til venstre i Θ-notasjonen, men dette hadde blitt mer komplisert.

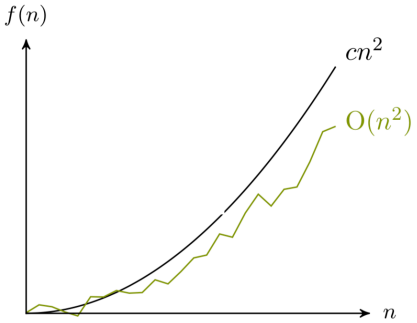
$$42 = \Theta(1)$$
$$2n + 5 = \Theta(n)$$
$$4n^2 + 100n = \Theta(n^2)$$
$$an^3 + bn^2 = \Theta(n^3)$$
$$n^k + n^{k-1} = \Theta(n^k)$$
$$n + \lg n = \Theta(n)$$
$$n^2 + n \lg n = \Theta(n^2)$$
$$2^n + n^k = \Theta(2^n)$$
$$n! + 2^n = \Theta(n!)$$

Θ-notasjonen



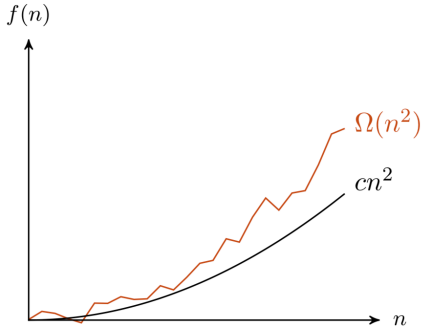
Ligger mellom to skaleringer av samme kurve, for store n

O-notasjonen



Ligger under skalert kurve, for store n

Ω-notasjonen



Ligger over skalert kurve, for store n

Notasjonen $f(n) = O(g(n))$ betyr at den asymptotiske operatoren O produserer "mengder" med funksjoner som $f(n)$ er en del av ($f(n) \in O(g(n))$). Vi bruker = fordi det er praktisk i tilfeller der asymptotiske notasjoner dukker opp i funksjoner, for eksempel $n^2 + O(n)$. Figuren til høyre viser de ulike asymptotiske notasjonene og tolkningen av disse.

ω	$>$
Ω	$\geq$
Θ	$=$
O	$\leq$
o	$<$

Kjøretid

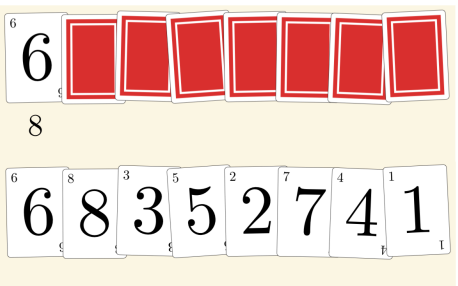
Kjøretiden er en funksjon av problemstørrelsen, og vi skiller mellom:

- Best-case: beste mulig kjøretid for en gitt størrelse
- Worst-case: verste mulig kjøretid for en gitt størrelse
- Average-case: forventet kjøretid, gitt en sannsynlighetsfordeling. Hvis ikke fordelingen er gitt kan vi anta at alle inputs er like sannsynlig.

Som regel bruker vi worst-case kjøretid. Vi skal nå finne kjøretiden til brute-force algoritmen og innsettingssorteringsalgoritmen for sortering av åtte kort.

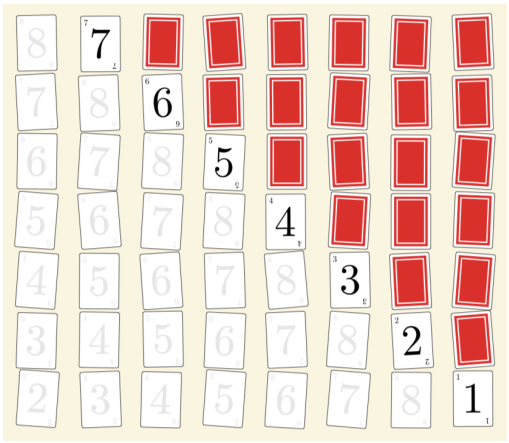
Brute-force

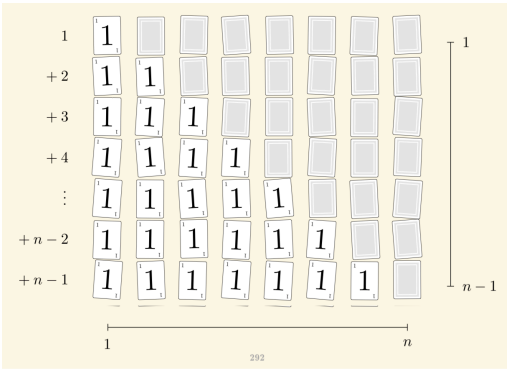
Første posisjon har 8 muligheter, andre har 7, tredje har 6, osv. Mulighetene for hver posisjon er uavhengige av hverandre, så det totale antall muligheter blir produktet: $8 * 7 * 6 * 5 * 4 * 3 * 2 * 1$. Dette er definisjonen av $n!$, som er noe av det verste vi kan få. Altså kjøretiden til brute-force algoritmen er $\Theta(n!)$.



Innsettingssortering

Det verste tilfellet for innsettingssortering er dersom den initielle array er i motsatt sortert rekkefølge, altså dersom $A = \langle 8, 7, 6, 5, 4, 3, 2, 1 \rangle$. Da må $A[j]$ flyttes forbi $A[1, \dots, j-1]$ andre elementer. For eksempel må $A[2] = 1$ som er 7 flyttes forbi $A[1]$ som er åtte. På figuren kan vi se at 7 må flyttes en plass, 6 må flyttes tre plasser, 5 må flyttes fire plasser, osv. Hver flytting er en operasjon som tar konstant tid $\Theta(1)$. Videre må vi finne ut hvor mange flyttinger det er totalt. I dette tilfellet er det $1 + 2 + 3 + 4 + 5 + 6 + 7 = 28$.



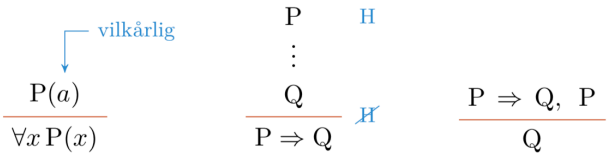


Når n blir stor blir det vanskelig å summere antall flyttinger ved hver iterasjon. Figuren til venstre viser det generelle tilfellet. Totale antall flyttinger vil være gitt av bredden ganger høyden delt på to, siden vi har en trekant med antall flyttinger:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$$

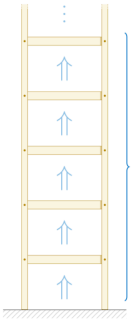
Siden vi dropper koeffisienter og lavere-ordens ledd.

Dekomponering – induksjon og rekursjon

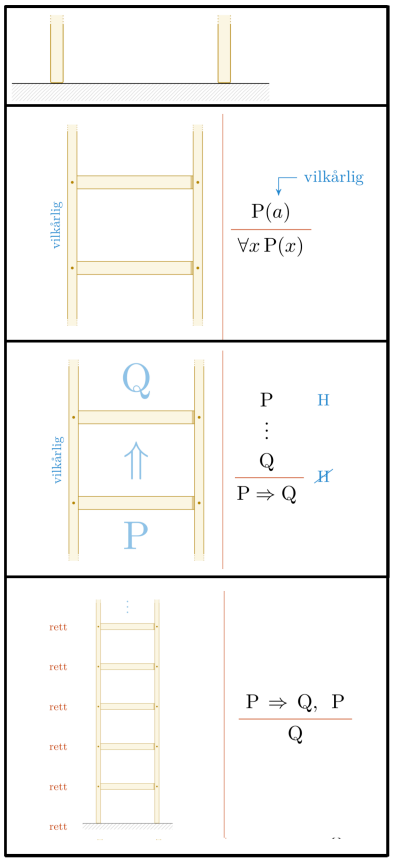


Induksjon kombinerer konseptene på figuren:

- Venstre: $P(a)$ er sann for vilkårlig a, derfor så vil $P(x)$ være sann for alle x.
- Midten: antar midlertidig at P er sann og viser deretter at Q er sann. Hvis P, så Q.
- Høyre: kalles modus ponens, og går ut på at hvis P, så Q kombinert med at P er sann betyr at Q er sann.



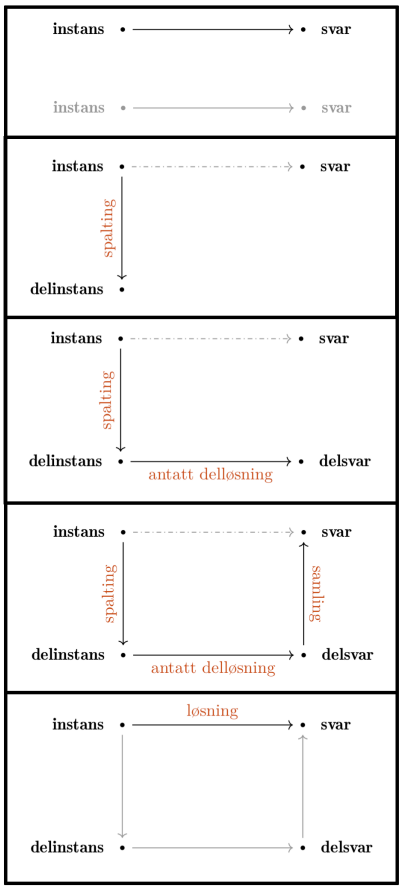
Ved induksjon vil vi altså introdusere og eliminere mange implikasjoner og generell induksjon kan foregå i et nettverk av utsagn. Likevel kan vi alltid ordne dem i en serie med trinn. Grunntilfellet er tilfellet som ikke baserer seg på noen andre. Resten av tilfellene er induktive, som vil si at de følger av tidligere trinn (figur til venstre).

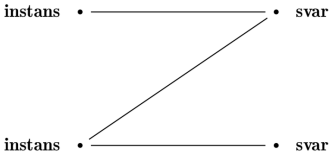


Når vi skal vise en induksjon gjør vi følgende:

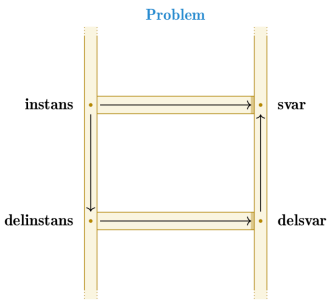
1. Vi må sjekke at grunntilfellet gjelder.
2. Vi beskriver alle induktive trinn og sjekker om et vilkårlig tilfelle gjelder
3. Vi vil vise implikasjonen $(P \to Q)$ som vi gjør ved å anta at forrige trinn gjelder og at dette gjør at neste trinn også gjelder. Dette kalles induksjonshypotesen.
4. Vi vet at grunntilfellet stemmer, noe som vil "smitte" til alle de induktive tilfellene.

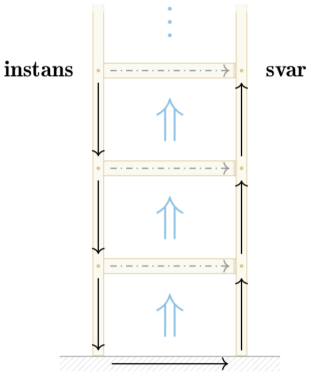
Dette kan ses på som dominobrikker. Brikke nummer en (grunntilfellet) faller. Om brikke nummer k faller (k er vilkårlig) vil også nummer $k + 1$ falle. Disse to faktaene gjør at alle brikkene faller.





Et problem vil være en relasjon mellom instanser og riktige svar. En algoritme finner ett riktig svar for hver instans og må derfor løse alle instanser. Vi ser på en vilkårlig instans som vi enda ikke vet hvordan vi løser. Vi spalter instansen inn i en eller flere delinstanser og antar at vi kan løse delinstansene. Vi kan til slutt samle delsvarene til et endelig svar og dermed har vi en løsning på instansen. Dette fungerer fordi det er basert på induksjon! Vi viser et grunntilfelle, antar delløsning og viser at spaltingen/samlingen er korrekt. Om disse er riktig vil alle svarene være riktig og dermed er løsningen riktig.





Kjerneprinsippet går ut på å bryte problemet ned så det kan løses trinn for trinn. Dette prinsippet lar oss fokusere på ett representativt trinn. En vanlig teknikk ved induksjon er **rekursive prosedyrer, som er prosedyrer som kaller seg selv**. Når vi skal løse et problem med en algoritme må vi:

1. Dele problemet opp i mindre problem
2. Anta at vi kan løse de mindre problemene = **induktiv premiss**
3. Konstruere en fullstendig løsning ut fra delløsningene = **induksjonstrinn**
4. Sørge for at ting terminerer

Det induktive premisset er antagelsen om at vi kan løse delproblemene, altså er dette betingelsen ved induksjonen. Induksjonstrinnet er å konstruere en løsning fra delløsningene, altså selve induksjonen av å gi løsning fra sanne delløsninger.

Loop – rekursjon og induksjon

Loop invarianter (s. 4) er egenskaper som er sann før og etter hver iterasjon. Tre ting vi må sørge for når vi lager en loop:

1. **Initialisering** – invarianten er sann før den første iterasjonen
2. **Vedlikehold i hver iterasjon** – skiller mellom:
    a. **Det induktive premisset** er at vi antar at invarianten er sann før iterasjonen
    b. **Induksjonstrinnet** er at vi viser at det er sant etter iterasjonen.
3. **Terminering** – løkka stopper

For eksempel kan invarianten være ”Det har gått bra så langt”. Dette kan vi anta og ”dra med oss”. Vi skal nå se på forskjellen mellom rekursiv og iterativ dekomponering.

Sum – rekursiv løsning

Vi ønsker å summere elementene i en tabell vha rekursjon. Dette gjør vi ved å få funksjonen til å kalle på seg selv for alle elementene unntatt det siste, fordi dette blir lagt til i den initielle kjøringen av funksjonen. Invarianten er at vi har sortert riktig så langt. Grunntilfellet er summen av en tom sekvens, som skal

```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i − 1)
4  return tmp + A[i]
```

være 0. Dette har vi sikret vha if-setningen. Det induktive premisset er at summen er rett, altså at $\text{SUM}(A, i-1)$ er summen av $A[1, \dots, i-1]$. Induksjonstrinnet er å legge til det siste elementet slik at den endelige summen blir rett, altså return setningen som sikrer at $\text{SUM}(A, i)$ er summen av $A[1, \dots, i]$.

For eksempel for $A = \langle 1, 2, 3 \rangle$ vil $SUM(A, 3)$ gi: $tmp = SUM(A, 2) \rightarrow tmp = SUM(A, 1) \rightarrow tmp = SUM(A, 0)$, som returnerer 0. Dermed vil $SUM(A, 1)$ returnere $0 + A[1] = 1$, $SUM(A, 2)$ vil returnere $1 + A[2] = 3$ og til slutt vil $SUM(A, 3)$ returnere $2 + A[3] = 6$. Legg merke til at tmp oppdateres fordi den er satt lik returverdien til funksjonen.

Sum – iterativ løsning

Vi ønsker nå å summere elementene i en tabell vha induksjon. Invarianten er fortsatt at vi har summert rett så langt og grunntilfellet (initialiseringen) er at en tom sum er null. Det induktive premisset er at summen er rett før iterasjonen, mens induksjonstrinnet er at vi legger til neste element. Termineringen er at vi til slutt har summert alle elementene.

Initialiseringen er at summen så lang er 0, altså res = 0. Induksjonshypotesen er at res er summen av $A[1, \dots, j-1]$. Induktiv trinn er at res er summen av $A[1, \dots, j]$, altså at siste element blir lagt til. Terminering er når $j = n$, slik at res er summen av $A[1, \dots, n]$ der n er lengden til tabellen.

```
SUM(A)
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

Rekursjon vs iterasjon

Rekursjon og iterasjon er i all hovedsak ekvivalente ting. Vi skal likevel sammenligne hvordan de to variantene oppfører seg. Hos begge er det induktive premisset at den "tidligere" delen allerede er summert. I den rekursive varianten gjør vi det rekursivt før vi legger til det siste elementet, mens i den iterative varianten har vi allerede gjort det iterativt når vi skal legge til det siste elementet. Dette er nesten samme sak…

```
SUM(A, i)
1  if i < 1
2      return 0
3  tmp = SUM(A, i − 1)
4  return tmp + A[i]
```

```
SUM(A)
1  res = 0
2  for j = 1 to A.length
3      res = res + A[j]
4  return res
```

i, j		
	5	1
	2	2
	4	3
	7	4

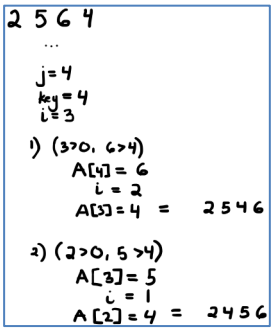
Begge metodene bygger på at vi antar at vi har gjort noe rett for de $n-1$ elementene og bygger så videre til n ved å gjøre noe med det siste elementet. Løsningen blir da rett, enten du bruker rekursjon eller iterasjon. Dette kan brukes for å lage utallige algoritmer fra samme idé!

Insertion-sort

Dekomponeringen er omtrent den samme som ved Sum-algoritmen, vi bare bytter ut sum med sortering. I stedet for å legge til neste element, så setter vi inn neste element på rett plass. Vi vil sortere elementene i en tabell. To mulige fremgangsmåter:

- **Rekursive dekomponering** – sorterer alle unntatt det siste elementet. Invarianten er at sorteringen er rett så langt. Det induktive premisset er at man antar at $n-1$ elementer er sortert riktig, mens induksjonstrinnet er å sette inn siste element.
- **Iterativ dekomponering** – invarianten er at sorteringen er rett så langt og initialisering er at en tom sekvens er sortert. Det induktive premisset er at sorteringen er rett før iterasjon, mens induksjonstrinnet er å sette inn neste element. Termineringen er at vi til slutt har sortert alle elementene.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      i = j − 1
4      while i > 0 and A[i] > key
5          A[i + 1] = A[i]
6          i = i − 1
7      A[i + 1] = key
```



Vi bruker induksjon for å lage algoritmen. Induksjonshypotesen er at $A[1, \dots, j-1]$ er sortert rett, noe som gis av for-løkken. Induksjonstrinnet er at neste element blir satt på rett plass, noe som sikres av at vi lagrer elementet som *key* og setter det inn på riktig plass basert på sammenligning med de andre elementene. Vi bruker i for å betegne elementet foran, Så lenge det er et element foran og dette elementet er større enn key, så skal elementet foran flytte en plass til høyre, mens key skal flyttes en plass til venstre. Deretter oppdaterer vi i, slik at vi ser på det nye elementet foran og prosessen gjentas helt til *key* er på riktig plass. Dermed vil vi få større elementer til høyre og mindre elementer til venstre. Figuren til venstre viser et eksempel på hvordan algoritmen virker.

	Ω	O	Θ
B	n	n	n
A	n^2	n^2	n^2
W	n^2	n^2	n^2
?	n		n^2

Denne figuren viser kjøretiden ved Best-, Average- og Worst-case, samt helt generelt. Ved best-case vil alle elementene allerede være sortert, slik at vi ikke trenger å flytte på noe. Vi trenger kun å gå igjennom tabellen en gang, og se at alt er der det skal. Dvs. én for-løkke som krever n steg. Ved average-case må hvert element flyttes halvparten av veien det må gå ved worst-case. Dermed får vi halvparten av worst-case, og koeffisienter blir droppet. Ved worst-case er kjøretiden n^2 siden det er en dobbelt nestet loop struktur.

Induksjonseksempel

Vi skal se på et eksempel der vi bruker induksjon for å vise at $n! > 2^n$ for alle heltall $n \geq 4$. Dette er det samme som å vise at $n! > 2^n$ for et vilkårlig heltall $n \geq 4$. Vi ser på:

- Grunntilfellet – vi velger $n = 4$, som gir at $P(4) = 4! = 24 > 15 = 2^4$.

Vi vil vise at $P(4) \rightarrow P(5) \rightarrow \cdots$. Siden n er vilkårlig, holder det å vise $P(n-1) \rightarrow P(n)$.

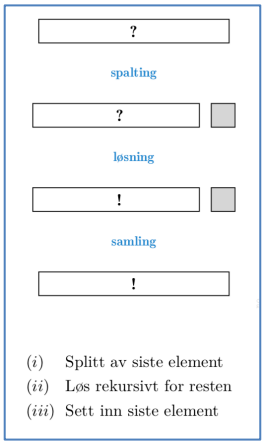
- Induksjonshypotese (premiss) – Vi antar at det forrige elementet er sant, altså $P(n-1)$. Dette gir at $(n-1)! > 2^{n-1}$
- Induksjonstrinn – vi antar $P(n-1)$ og vil utlede $P(n)$. For å gjøre dette bryter vi ned $n!$ og 2^n vha rekursjon:
  - $n! = n * (n-1)$
  - $2^n = 2 * 2^{n-1}$

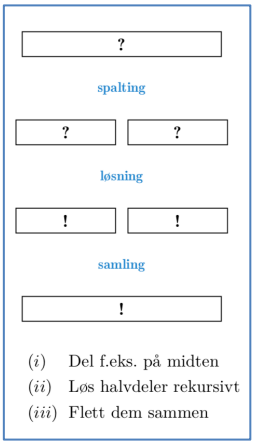
  Vi har antatt at $(n-1)! > 2^{n-1}$ og vet at $n > 2$. Derfor vil $P(n-1) \rightarrow P(n)$

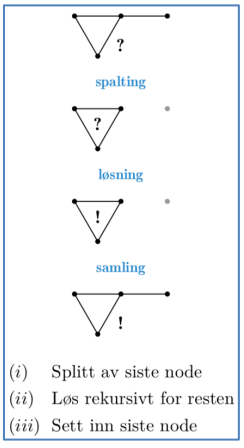
Vil vise	$n! > 2^n$	$P(n)$
Grunntilfelle	$4! > 2^4$	$P(4)$
Induksjonshypotese	$(n-1)! > 2^{n-1}$	$P(n-1)$
Induksjonstrinn	$\Longrightarrow n! > 2^n$	$P(n)$
Rekursjon	$n! = n \cdot (n-1)!$	
	$2^n = 2 \cdot 2^{n-1}$	

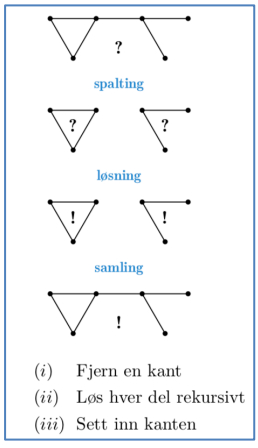
Vi vet nå at $P(n)$ for $n = 4$ og $P(n-1) \rightarrow P(n)$ for $n > 4$. Dermed har vi vist at $n! > 2^n$ for alle $n \geq 4$

Noen vanlige dekomponeringer









Forelesning 2 – Datastrukturer

En datastruktur er en måte å lagre og organisere data for å gjøre det lettere å endre eller få tilgang til dataen senere. Ingen datastruktur vil fungere bra for alle formål, så det er viktig å kjenne styrkene og begrensningene til flere av de. For å unngå grunnleggende kjøretidsfeller er det viktig å kunne organisere og strukturere data fornuftig.
Dynamiske sett er sett som kan vokse, krympe eller endres på en annen måte over tid. Beste måte å implementere ett dynamisk sett avhenger av operasjonene som kreves.

- Forstå hvordan *stakker* og *køer* fungerer (STACK-EMPTY, PUSH, POP, ENQUEUE, DEQUEUE)
- Forstå hvordan *lenkede lister* fungerer(LIST-SEARCH, LIST-INSERT, LIST-DELETE, LIST-DELETE', LIST-SEARCH', LIST-INSERT')
- Forstå hvordan *pekere* og *objekter* kan implementeres
- **Forstå hvordan *direkte adressering* og *hashtabeller* fungerer** (HASH-INSERT, HASH-SEARCH)
- Forstå *konfliktløsing ved kjeding* (*chaining*) (CHAINED-HASH-INSERT, CHAINED-HASH-SEARCH, CHAINED-HASH-DELETE)
- Kjenne til grunnleggende *hashfunksjoner*
- Vite at man for *statiske datasett* kan ha *worst-case* $O(1)$ for søk
- Kunne definere *amortisert analyse*
- Forstå hvordan *dynamiske tabeller* fungerer (TABLE-INSERT)

Kapittel 10 – Elementære datastrukturer

10.1 Stacks og queues

Stacks og queues er dynamiske sett der elementet som fjernes fra settet av DELETE operasjonen er spesifisert på forhånd og krever derfor ingen input annet en arrayen.

- **Stack –** elementet som fjernes er det som sist ble satt inn, altså er det en **last-in, first-out (LIFO) metode**.
- **Queue –** elementet som fjernes er det som har vært lengst i settet, altså er det en **first-in, first-out (FIFO) metode.**

Stacks

Hos en stack vil PUSH være INSERT operasjonen, mens POP er DELETE operasjonen og tar ingen element som input. PUSH vil plassere elementer på slutten av settet, mens POP vil fjerne elementer fra slutten av settet. Vi kan implementere en stack med n elementer vha en array $S[1, \dots, n]$. $S.top$ vil da gi indeksen til elementet som sist ble satt inn, slik at $S[1]$ er elementet ved bunnen av stacken og $S.top$ er elementet ved toppen. **Når $S.top = 0$ vil stacken være tom, mens hvis $S.top = n$ er stacken full.** Hvis vi kjører pop på en tom stack vil stacken ***underflows***, mens hvis vi kjører push på en full stack, slik at $S.top$ overgår n, vil stacken ***overflows.***

```
STACK-EMPTY(S)
1  if S.top == 0
2      return TRUE
3  else return FALSE
```

STACK-EMPTY – sjekker om stacken er tom
Dersom $S.top$ er null vil stacken være tom, og da skal metoden returnere TRUE, hvis ikke skal den returnere FALSE.

```
PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x
```

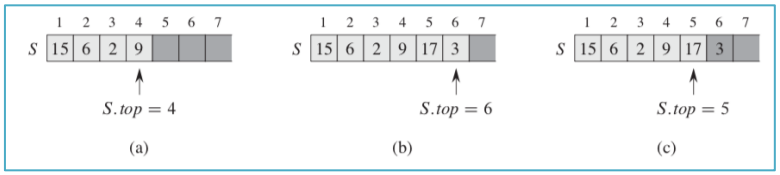
PUSH – legger til element
Vi øker $S.top$, altså indeksen til siste innsatt element, med én og setter det nye elementet inn ved denne indeksen.

```
POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

POP – fjerner et element

Vi sjekker om stacken er tom ved å kalle på STACK-EMPTY. Dersom dette er tilfellet skal en feilmelding med beskjeden "underflow" gis. Hvis ikke vil vi redusere $S.top$ med én og fjerner elementet ved toppen gitt av $S.top + 1$. Legg merke til at dette elementet blir returnert og at det er ingen input siden det er forhåndsbestemt at elementet ved toppen skal fjernes!

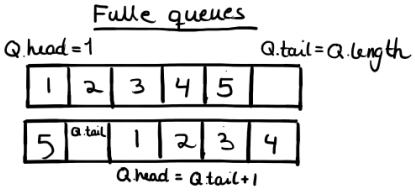
Disse stack-operasjonene tar konstant tid, altså $O(1)$. Figuren viser en stack med array $Q[1, \dots, n]$. $PUSH(S, 17)$ og $PUSH(S, 3)$ vil legge til elementene 17 og 3 (b), mens $POP(S)$ vil fjerne og returnere element 3 (c).



Queues

Hos en queue vil ENQUEUE være INSERT operasjonen, mens DEQUEUE er DELETE operasjonen og tar ingen element som input. FIFO egenskapen til queue gjør at det fungerer som en kø i butikken. En queue har et **hode (*head)*** og en **hale (*tail*).** ENQUEUE vil legge til et element ved halen, mens DEQUEUE vil fjerne et element fra hodet.

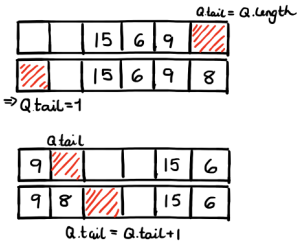
Vi kan implementere en queue med n elementer vha en array $Q[1, \dots, n]$. Kommandoen $Q.head$ gir indeks til hodet, mens $Q.tail$ gir indeks til posisjonen der neste element vil legges til ved halen (dvs. siste element i en queue er ved $Q.tail - 1$). **En queue vil "wrap around" slik at lokasjon 1 følger etter lokasjon n på en sirkulær måte. Dersom $Q.tail = Q.head = 1$ vil queue være tom. Dersom $Q.head = 1$ og $Q.tail = Q.length$ eller $Q.head = Q.tail + 1$ vil queue være full** (OBS: en queue er full når det er kun én ledig plass i array, fordi hvis den er helt full er det ingen måte å skille mellom når den er full eller tom). Hvis vi kjører DEQUEUE på en tom queue vil queue ***underflow***, mens hvis vi kjører ENQUEUE på en full queue vil queue ***overflow***.




```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3       Q.tail = 1
4  else Q.tail = Q.tail + 1
```

ENQUEUE – legger til element

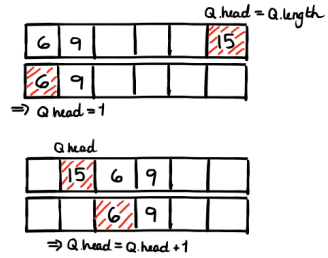
Vi legger til elementet x ved $Q.tail$. Dersom $Q.tail$ er ved enden av arrayen, må halen flyttes til starten (wrap-around). Hvis ikke kan vi øke $Q.tail$ med én.



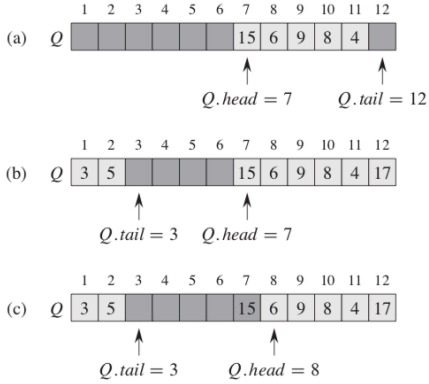

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3       Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

DEQUEUE – fjerner element

Denne metoden har ingen input, siden det er forhåndsbestemt at elementet ved hodet skal fjernes. Vi lagrer dette elementet som x. Dersom $Q.head$ er ved enden av arrayen, må hodet flyttes til starten (wrap-around). Hvis ikke kan vi øke $Q.head$ med én.



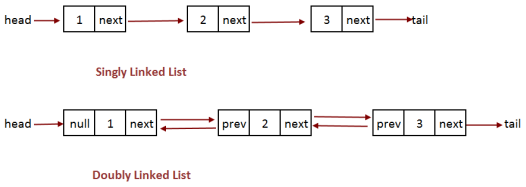
Disse queue-operasjonene tar konstant tid, altså $O(1)$. Figuren viser en queue med $Q[1, \dots, 12]$. Legg merke til at den er wrap-around, altså når vi når enden av stacken går vi til starten. $ENQUEUE(S, 17)$, $ENQUEUE(S, 3)$ og $ENQUEUE(S, 5)$ vil legge til elementene 17, 3 og 5 ved $Q.tail$ (b). Her ser vi at elementene legges til ved starten av array når slutten er fylt. $DEQUEUE(S)$, vil fjerne elementet ved $Q.head$ og krever derfor ingen element input (c).



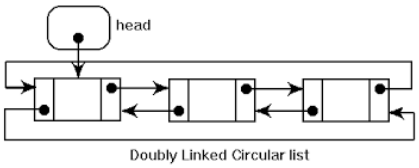
10.2 Linked list

En linked liste er en datastruktur der objektene er ordnet på en lineær måte. I en array blir den rekkefølgen bestemt av indekser, mens **i en linked list blir rekkefølgen bestemt av en peker i hvert objekt**. Vi skiller mellom:

- **Dobbelt linked list – hver element er et objekt med en *key* egenskap og to pekeregenskaper: *next* og *prev***. Gitt et element x, vil $x.next$ peke mot neste element, mens $x.prev$ peker mot forrige element. $x.next = NIL$ betyr at det er ingen neste element og x er halen, mens $x.prev = NIL$ betyr at det er ingen forrige element og x er hodet. $L.head$ vil peke mot første element i listen, så listen er tom dersom $L.head = NIL$.
- **Enkelt linked list –** lik en dobbelt linked list, men **ingen *prev* peker**.



En linked list kan være **sortert,** slik at den lineære rekkefølgen til listen er lik den lineære rekkefølgen til key-verdiene til elementene. Dvs. det minste elementet er ved hodet til listen, og det største er ved halen. I en ikke-sortert linked list vil elementene være tilfeldig ordnet. En linked list kan også være sirkulær, slik at $prev$ til hodet vil peke mot halen og $next$ til halen vil peke mot hodet. Listen er ordnet som en ring av elementer. For en ikke-sirkulær linked list vil $prev$ til hode være NIL og $next$ til halen er NIL. Vi bruker dobbelt linked list som er ikke-sortert. Ofte brukte kommandoer:

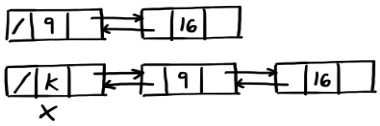


- $x.key$ = gir key-verdien til dette elementet
- $x.next$ = gir neste element
- $x.prev$ = gir forrige element

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3       x = x.next
4  return x
```

LIST-SEARCH – finner element

Denne metoden finner første element med key k i en liste L vha enkel lineær søk og returnerer en peker til dette elementet. Vi begynner med å sette x lik det første elementet i L. Så lenge x ikke er NIL eller har key lik k vil vi sette x lik det neste elementet. Dersom L inneholder et element med key-verdi k vil x være en peker mot dette elementet, ellers vil x være NIL siden $x.next$ til halen er NIL. **For å søke en liste med n objekter vil LIST-SEARCH bruke worst-case tid $\Theta(n)$, siden det kan hende den må søke gjennom hele listen.**




```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3       L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

LIST-INSERT – setter inn element

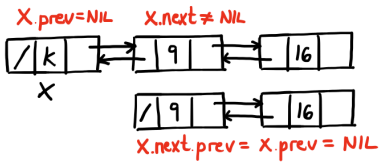
Elementet x har allerede fått en key egenskap og LIST-INSERT vil "skjøte" dette elementet til fronten av listen (figur til høyre). Vi gir x en $next$-peker som peker mot første element i L. Dersom det er et første element i L, lar vi $prev$-pekeren til dette elementet peke mot x. Merk: hvis L er tom vil $x.next = NIL$. Deretter setter vi første element i L lik x og $prev$-pekeren til x er derfor NIL. **Kjøretiden til LIST-INSERT på en liste med n elementer er $O(1)$.**

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2       x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5       x.next.prev = x.prev
```

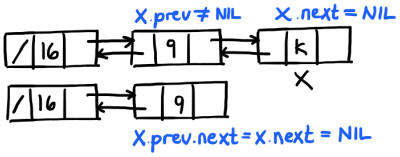
LIST-DELETE – fjerner element

Denne metoden må få elementet x som skal fjernes som input og vil spalte x fra listen ved å oppdatere pekerne. Dersom x har et element foran, må $next$-pekeren til dette elementet settes lik $next$-pekeren til x

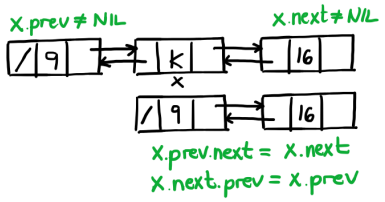
Merk: x vil fortsatt peke på nodene.

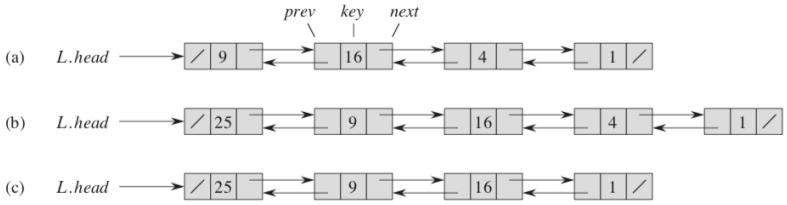


som kan være et annet element eller NIL. Hvis x er det første elementet i listen, må hodet settes lik elementet etter x. Dersom det er et element etter x, må $prev$-pekeren til dette elementet settes lik $prev$-pekeren til x som kan være et annet element eller NIL. Figuren viser de ulike tilfellene. Denne metoden går ut på å "hoppe over" x.



LIST-DELETE bruker $O(1)$ tid for å fjerne et gitt element. Hvis vi ønsker å fjerne et element med en gitt key-verdi, må vi først kalle LIST-SEARCH for å hente en peker x til dette elementet. I dette tilfellet vil LISTE-DELETE bruke $\Theta(n)$ tid i worst-case siden den må søke gjennom n elementer.





Figuren viser en dobbelt-lenket liste for det dynamiske settet {1, 4, 6, 16} og $L.head$ peker mot første element. $LIST\text{-}INSERT(L, x)$ med $x.key = 25$ vil gjøre at dette elementet blir lagt til i starten av listen og peker mot det forrige hodet med $key = 9$ (b). $LIST\text{-}DELETE(L, x)$ med $x.key = 4$ vil fjerne dette elementet og oppdatere pekerne til de andre elementene.

Sentinel

En sentinel er et dummy objekt som lar oss forenkle grensebetingelsene. Et eksempel på en sentinel er objektet $L.nil$ som kan legges til listen for å representere NIL og som har samme egenskaper som de andre objektene i listen. Alle referanser til NIL erstattes av en referanse til $L.nil$. Dette gjør at vi får en **sirkulær, dobbelt linked liste med en sentinel $L.nil$ om ligger mellom hodet og halen**. Derfor vil $L.nil.next$ peke mot hodet (erstatter $L.head$) og $L.nil.prev$ peker mot halen. En tom liste vil kun bestå av sentinel, og da vil $L.nil.next$ og $L.nil.prev$ peke mot $L.nil$.

```
LIST-DELETE'(L, x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

LIST-DELETE' – fjerner element uten grensebetingelser

LIST-DELETE' er en forenklet versjon av LIST-DELETE der vi **ignorerer grensebetingelsene** ved hodet og halen til listen. Dersom x ikke har et neste element vil $x.next$ peke mot $L.nil$. Når vi kaller på $x.next.prev = x.prev$ vil dette gjøre at $prev$-pekeren til $L.nil$ objektet settes lik $prev$-pekeren til x. Samme gjelder dersom x ikke har et element foran.

```
LIST-SEARCH'(L, k)
1  x = L.nil.next
2  while x ≠ L.nil and x.key ≠ k
3      x = x.next
4  return x
```

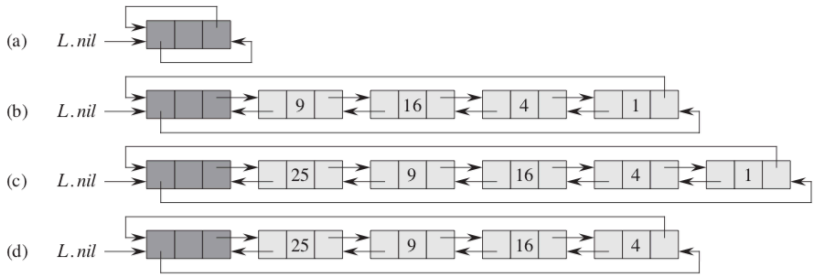
LIST-SEARCH' – søker etter element

LIST-SEARCH' fungerer på samme måte som før, men $L.head$ erstattes med $L.nil$.next og NIL erstattes med referanser til $L.nil$

```
LIST-INSERT'(L, x)
1  x.next = L.nil.next
2  L.nil.next.prev = x
3  L.nil.next = x
4  x.prev = L.nil
```

LIST-INSERT' – fjerner element uten grensebetingelser

LIST-INSERT' er en forenklet versjon av LIST-INSERT der vi **ignorerer grensebetingelsen** ved hodet. Vi trenger ikke å sjekke om listen har et første element, fordi dersom det ikke er tilfellet vil $x.next$ settes lik $L.nil$ objektet og da vil $x.next.prev = x$ gjøre at $prev$-pekeren til $L.nil$ peker mot x. Hvis listen ikke er tom, vil det $prev$-pekeren til det gamle hodet peke mot x. Videre må vi oppdatere $L.nil.next$ til å peke mot x, siden dette er det nye hodet til listen og $x.prev$ må peke mot $L.nil$ objektet.

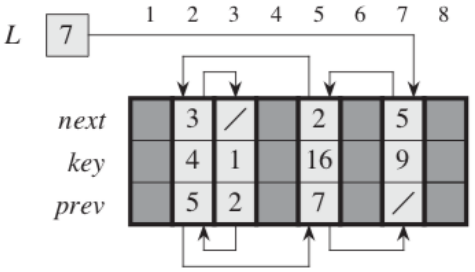


(a) viser en tom liste som kun består av $L.nil$ objektet. (b) viser en linked liste med key 9 ved hodet. (c) viser listen etter LIST-INSERT'(L, x) der $x.key = 25$. $L.nil.next$ vil nå peke mot dette elementet, siden det er det nye hodet og $prev$ hos det gamle hodet, vil peke mot x. (d) viser hodet etter LIST-DELETE'(L, x) der $x.key = 1$. Dette innebærer at $L.nil.prev$ må oppdateres til å peke mot det nye elementet ved halen og dette elementet sin $next$ må peke mot $L.nil$ objektet.

Sentinel vil sjeldent redusere den asymptotiske tiden, men de kan redusere konstante faktorer, altså spare $O(1)$ tid. Målet ved å bruke sentinel i looper er ofte heller en mer oversiktlig kode. Dersom det er mange små lister kan sentinel føre til bortkastet bruk av minne, så de bør kun bruker når de virkelig forenkler koden!

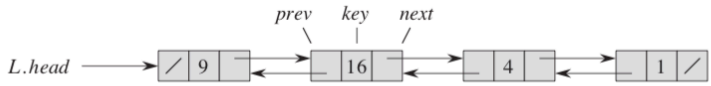
10.3 Implementasjon av pekere og objekter

I noen språk er det ikke gitt implementasjon for pekere og objekter, så derfor skal vi se hvordan vi kan lage disse fra arrays og array indekser.

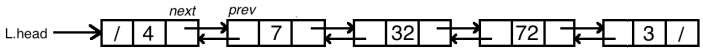


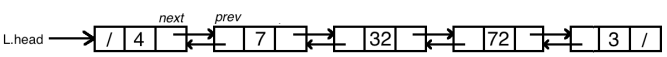
Multiple-array representasjon av objekter

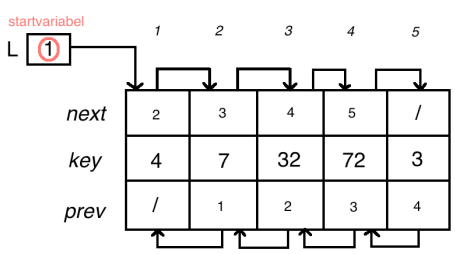
En samling av objekter som har samme egenskaper kan representeres vha en array for hver egenskap. Figuren til høyre viser hvordan vi kan implementere en linked list vha tre arrays: $next$, key og $prev$. **Legg merke til indeksene 1-8 som er pekerne. For en bestemt indeks i vil $key[x]$, $next[x]$ og $prev[x]$ representere et objekt i linked listen.**

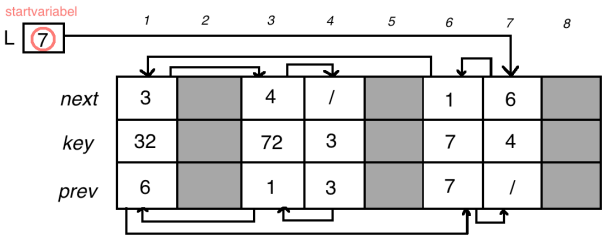


Det er flere måter å implementere denne representasjonen (dvs. vi kan variere hvor vi plasserer de grå, tomme sekvensene og sekvensene med bestemte verdier). På figuren ser vi implementasjonen der det første elementet i linked listen er plassert ved indeks 7, gitt av variabelen L. Siden dette er hodet i listen vil $prev = \text{NIL}$, mens $next$ er neste element i listen som er plassert ved indeks 5. Slik fortsetter det helt til siste element i listen som er plassert ved indeks 3 og har $next = \text{NIL}$. Figuren under viser hvordan en linked list kan implementeres på ulike måter.







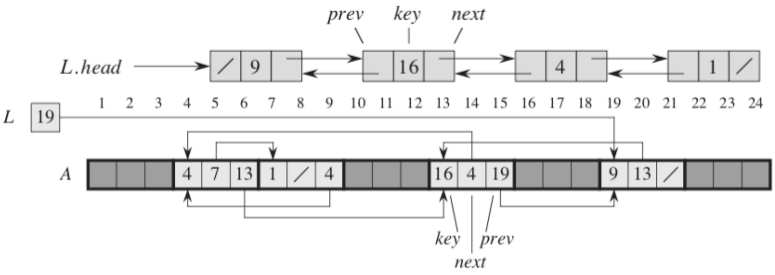


Dersom I er startvariabelen, N er next arrayen, K er key arrayen og P er prev arrayen, vil:

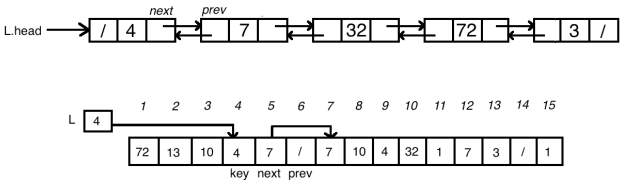
- Venstre: $I = 1, N = \langle 2,\ 3,\ 4,\ 5,\ / \rangle, K = \langle 4, 7, 32, 72, 3 \rangle, P = \langle /, 1,\ 2,\ 3,\ 4 \rangle$.
- Høyre: $I = 7,\ N = \langle 3, 0, 4,\ / , 0, 1, 6, 0 \rangle, K = \langle 32, 0, 72, 3, 0, 7, 4, 0 \rangle,\ P = \langle 6, 0, 1, 3, 0, 7,\ /, 0 \rangle$.

En singel-array representasjon av objekter

Vi kan bruke en enkelt array A for å lagre en linked list. I en slik representasjon vil objektet være en subarray $A[j, \dots, k]$ med lengde 3, mens pekeren er indeksen j (1-24 på figuren). Hver subarray vil ha tre felt, som består av key, $next$ og $prev$ (OBS: i denne rekkefølgen!). Dersom objektet har flere egenskaper sier vi at hver egenskap korresponderer til en offset i rekkevidden 0 til $k - j$ (altså en av verdiene i subarrayen). For eksempel har prev offset 2 som betyr at hvis objektet har indeks i vil $prev$-verdien være ved indeks $i + 2$.



Det er flere måter å implementere denne representasjonen (dvs. vi kan variere hvor vi plasserer de grå, tomme sekvensene og subarrayene). På figuren ser vi implementasjonen der det første elementet i linked listen er plassert ved indeks 19, gitt av variabelen L. Dette er hodet til listen og vil derfor ha $prev = \text{NIL}$, mens $next$ peker mot neste element som er plassert ved indeks 13.



Denne figuren viser et annet eksempel der startvariabelen er $I = 4$ og arrayen er $A = \langle 72, 13, 10, 4, 7, /, 7, 10, 4, 32, 1, 7, 3, /, 1 \rangle$.

Singel-array representasjonen er fleksibel siden den lar oss lagre objekter av ulik lengde i samme array. De fleste datastrukturene vi ser på er likevel homogene, altså lagd av elementer som har samme egenskaper og derfor er multi-array representasjonen tilstrekkelig.

Tildeling og frigjøring av objekter ikke fremhevet i læringsmål

For å sette inn en key i en dobbelt linked liste, må vi tildele en peker som hører til et ledig objekt. Derfor er det lurt å holde styr over objekter som for tiden ikke er i bruk. Anta at arrays i multiple-array representasjonen har lengde m og at det dynamiske settet inneholder $n \leq m$ elementer. Da vil det være n opptatte objekter i settet og $m - n$ frie objekter. De frie objektene er tilgjengelig for å representere elementer som blir satt inn i det dynamiske settet. De kan plasseres i en singel-linked liste kalt den **frie listen** som kun har verdier i *next*-arrayen. Hodet til listen er gitt av den globale variabelen ***free***. Merk at hvert objekt i representasjonen er enten i listen L eller i den frie listen.
Den frie listen oppfører seg som en stack ved at neste objekt som skal settes inn blir tildelt pekeren som sist ble frigjort. Vi implementerer PUSH og POP metodene som vil hhv. frigjøre og tildele objekter:

```
ALLOCATE-OBJECT()
1  if free == NIL
2      error "out of space"
3  else x = free
4      free = x.next
5      return x
```

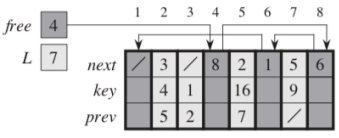
ALLOCATE-OBJECT() – tildeler verdi til x indeksen

Dersom *free* er NIL betyr det at den frie listen er tom og det er ikke plass til å tildele en peker til et objekt. Hvis ikke vil x indeksen settes lik pekeren gitt av *free* variabelen og *free* variabelen oppdateres til neste verdi i den frie tabellen. Merk: denne metoden returnerer indeksen som har blitt tildelt x.

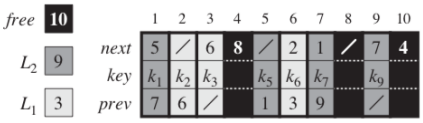
```
FREE-OBJECT(x)
1  x.next = free
2  free = x
```

FREE-OBJECT() – oppdaterer den frie tabellen

Denne brukes etter at et element har fjernet for å gjøre det om fra et opptatt objekt til et ledig objekt. Altså, vil den oppdatere den frie tabellen ved å sette $x.next$ lik nåværende *free* verdi og deretter oppdatere *free* til x. Dette gjør at den frie tabellen blir riktig.



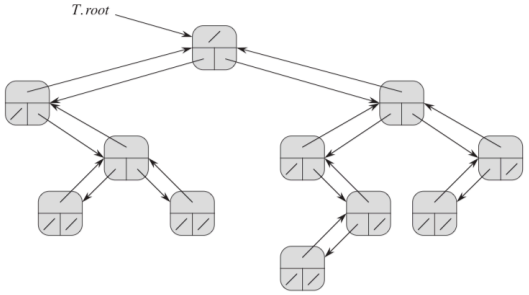
(a) viser hvordan listen L og den frie listen kan representeres sammen. For å få resultatet i (b) må vi først kalle på ALLOCATE-OBJECT() for å finne den ledige indeksen gitt av *free*. Deretter må vi sette $key[4] = 25$ for så å kalle på $LIST\text{-}INSERT(L, 4)$. For å få resultat i (c) må vi kalle på LIST-DELETE(L, 5) som vil fjerne objektet og deretter FREE-OBJECT(5) som vil legge dette objektet til i den frie listen. **Disse prosedyrene bruker tiden $O(1)$.**



Figuren til venstre viser at den frie listen kan brukes samtidig for flere linked lister.

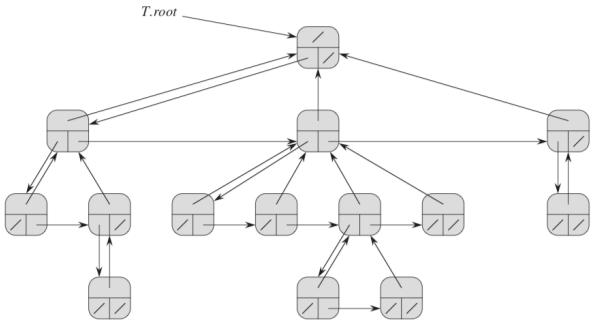
10.4 Representerende *rooted trees*

Vi skal se hvordan vi kan bruke *rooted trees* for å representere linked datastrukturer. Hver node til treet representerer et objekt og inneholder en *key* egenskap og pekere til andre noder. Vi skiller mellom:



- **Binære trær (T)–** hver node kan ha en venstre ($left$) og en høyre ($right$) barnenode og alle bortsett fra roten har en parent node (p). Dersom $x.p = NIL$ vil x være roten. Hvis $x.left = NIL$ vil ikke x ha noe venstre barnenode, og samme for høyre barnenode. Roten til hele treet er gitt av $T.root$, så hvis denne er NIL dersom treet er tomt.
- ***Rooted trees* med ubegrenset forgreining** – brukes for å representere trær med et vilkårlig antall barnenoder og bruker kun $O(n)$ rom for *rooted trees* med n noder. Hver node har en parent peker p og $T.root$ peker mot roten til treet T. I stedet for å ha peker til alle barnenodene, vil derimot **hver node x kun ha to pekere**:
  1. $x.left\text{-}child$ peker mot barnenoden til x som er lengst til venstre
  2. $x.right\text{-}sibling$ peker mot søskennoden som er rett til høyre for x

  Alle barnenoden peker mot foreldrenoden, men foreldrenoden vil altså ikke peke mot alle barnenodene.



En annen representasjon er heap som vi skal se på i kapittel 6.

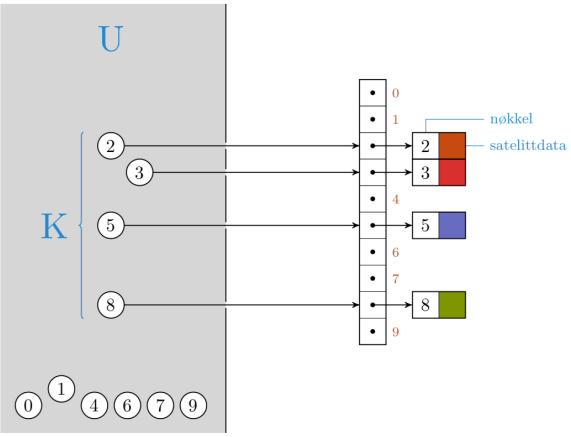
Kapittel 11 – Hash tabeller

Mange applikasjoner krever et dynamisk sett som kun støtter ordbok operasjonene INSERT, SEARCH og DELETE, og en hash tabell er en effektiv datastruktur for å implementere slike sett. SEARCH i en hash tabell kan ta like lang tid som i en linked list

(dvs. $\Theta(n)$ worst-case), men hashing vil generelt være svært effektiv. Den gjennomsnittlige søketiden for et element i en hash tabell er $O(1)$.

Direkte adressering av en array benytter seg av evnen til å undersøke en vilkårlig posisjon i løpet av tiden $O(1)$. Vi kan bruke direkte adressering når vi har en array som har en posisjon for hver mulige *key*. Når antall *keys* er lite relativt til antall mulige *keys* vil hash tabeller bli et effektivt alternativ til direkte adressering, siden hash tabeller som regel bruker en array som har størrelse proporsjonal til antall *keys* som faktisk er lagret.

Hashing er en ekstremt effektiv og praktisk teknikk, siden de grunnleggende ordbok operasjonene krever i gjennomsnitt tiden $O(1)$



11.1 Direkte-adresse tabeller

Direkte adressering er en enkel teknikk som fungerer bra når universet U av *keys* er relativt lite. Anta at vi har et dynamisk sett der hvert element har en nøkkel fra universet $U = \{0, 1, \dots, m-1\}$ der m ikke er for stor og ingen elementer har samme *key*. For å representere settet bruker vi en array, kalt **direkte-adresse tabell** $T[0, \dots, m-1]$, der hver *key* i universet korresponderer til en indeks i tabellen. Settet $K = \{2, 3, 5, 8\}$ er faktisk lagret *keys* som gir lukene i tabellen som inneholder pekere til elementer. ***Key*-verdien *k* vil brukes som indeks i tabellen som vil inneholde en peker som peker mot et element i settet som har *key k*.** Lukene som ikke inneholder pekere inneholder NIL (mørke grå). I noen tilfeller er det også mulig å lagre elementene fra det dynamiske settet i direkte-adresse tabellen.

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```

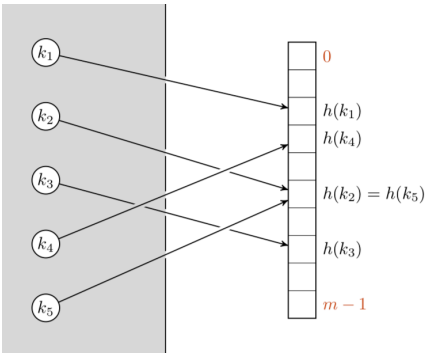
DIRECT-ADDRESS-SEARCH – vil returnere elementet ved *k*
Returnerer elementet i direkte-adresse tabellen som har indeks k eller NIL dersom det ikke er noe element ved denne indeksen. Dette elementet peker mot et element i settet som har *key k.*

DIRECT-ADDRESS-INSERT– vil sette inn element ved *k*
Vil sette elementet x inn i direkte-adresse tabellen ved indeksen gitt av *key*-verdien til elementet.

DIRECT-ADDRESS-DELETE– vil fjerne elementet ved *k*
Vil fjerne elementet x fra direkte-adresse tabellen ved indeksen gitt av *key*-verdien til elementet ved å sette elementet ved denne indeksen lik NIL.

11.2 Hash tabeller

Ulempen med direkte adressering er at dersom universet U er stort, kan det være upraktisk eller umulig å lagre tabellen T av størrelse $|U|$. Det kan også hende at mye av rommet som er tildelt T er bortkastet, siden settet av *keys* som faktisk er lagret (K) kan være svært lite sammenlignet med U. Dersom vi heller bruker hash tabeller kan vi redusere kravet for rommet til $|\Theta(K)|$ og fortsatt ha fordelen med at søk etter et element tar kun $O(1)$ tid. Ulempen er at dette er tilfellet for average-case, mens for direkte adressering vil worst-case søk ta $O(1)$ tid.



Ved direkte adressering er elementet med $key\ k$ lagret i luke k, mens ved hashing er elementet lagret i luke $h(k)$. Altså, **hashing bruker en hash funksjon h på $key\ k$ for å regne ut indeksen til luken.** Hash funksjonen vil ta en *key* fra universet U og regne ut indeksen til en luke i hash tabellen $T[0, 1, \dots, m-1]$, altså $h: U \rightarrow \{0, 1, \dots, m-1\}$. Størrelsen m til hash tabellen er som regel mye mindre enn $|U|$. Vi sier at elementet med *key* k hasher til luke $h(k)$ og at $h(k)$ er hash verdien til $key\ k$. Som vi kan se på figuren vil ikke hver indeks i hash tabellen korrespondere til en nøkkel i U, slik som i direkte adressering.

Ordbok operasjonene som blir nevnt er INSERT og SEARCH:

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == NIL
5            T[j] = k
6            return j
7       else i = i + 1
8  until i == m
9  error "hash table overflow"
```

HASH-INSERT – vil sette *key k* inn i hash tabellen

Denne metoden vil systematisk undersøke den gitte hash tabellen T til den finner en tom luke der den kan plassere k. For å bestemme hva som skal undersøkes vil hash funksjonen inkludere et **probe nummer** i som input, slik at $h: U \times \{0, 1, \dots m-1\} \rightarrow \{0, 1, \dots, m-1\}$. Alle posisjonene i hash tabellen vil tilslutt bli undersøkt. Probe nummeret begynner på 0, mens j settes lik hash verdien til k og i. Dersom luken ved denne hash verdien er tom vil k lagres i denne luken og indeksen til luken returneres. Hvis ikke vil probe nummeret økes med én og prosessen gjentas helt til $i = m$ (gått gjennom hele tabellen) eller vi finner en tom luke. Hvis tabellen er full vil en feilmelding returneres om overflow av hash tabellen.

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == k
5            return j
6       i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

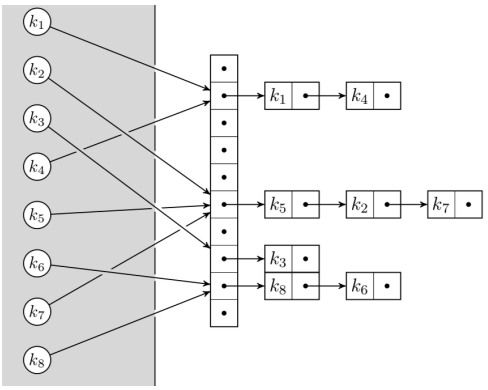
HASH-SEARCH – vil finne elementet med *key k*

Denne metoden vil returnere j dersom den finner en luke j som inneholder k eller NIL dersom k ikke er tilstede i tabellen T. Probe verdien settes lik 0, mens j settes lik hash verdien til k og i. Hvis luken ved posisjon j inneholder k vil metoden returnere j. Hvis ikke vil den øke probe verdien med én helt til den $T[j] = NIL$ (siden k ville ha blitt satt inn der) eller $i = m$ (gått gjennom hele tabellen).

DELETE er vanskelig fordi vi kan ikke fjerne en *key* fra luke *i* ved å endre den til NIL, fordi da kan vi få problemer med å hente *keys* som ved innsetting undersøkte og merket luke *i* som okkupert. Alternativt kan vi markere luken som DELETED og endre INSERTION slik at den vil sette inn *keys* i slike luker. SEARCH er som før.

Kollisjon

En ulempe med hashing er at to *keys* kan hashe til samme luke, noe som kalles en kollisjon. Vi kan velge en hash funksjon som minimerer antall kollisjoner ved å fremstå som tilfeldig (merk: h må være deterministisk, som vil si at et gitt input k alltid skal gi samme output $h(k)$). **Siden $|U| > m$ må det være minst to *keys* som har samme hash verdi**, så derfor er det ikke mulig å fullstendig unngå kollisjoner. Av denne grunn må vi ha metoder for å løse kollisjoner som oppstår, og den enkleste er chaining.



Kollisjon håndtering ved chaining

Ved chaining blir elementer som hasher til samme luke plassert i samme linked liste. Luke *j* vil inneholde en peker som peker mot hodet til listen av elementer som hasher til *j*. Hvis det ikke er noen slike elementer vil j inneholde NIL.

Ordbok operasjonene er enkle å implementere når kollisjoner blir håndtert av hashing.

```
CHAINED-HASH-INSERT(T, x)
1  insert x at the head of list T[h(x.key)]
CHAINED-HASH-SEARCH(T, k)
1  search for an element with key k in list T[h(k)]
CHAINED-HASH-DELETE(T, x)
1  delete x from the list T[h(x.key)]
```

CHAINED-HASH-INSERT – setter inn element
Denne metoden vil sette inn det gitt elementet x ved hodet til linked listen som ligger i luken ved posisjonen gitt av $h(x.key)$.

CHAINED-HASH-SEARCH – finner elementet
Luken ved posisjon $h(k)$ vil peke mot en linked list og metoden vil undersøke om denne listen inneholder et element med $key\ k$.

CHAINED-HASH-DELETE – fjerner element
Denne metoden vil finne luken ved posisjonen $h(x.key)$ og vil fjerne elementet x ved å oppdatere pekerne til elementet foran og etter x.

Worst-case kjøretiden for INSERT er $O(1)$. Denne prosedyren er rask fordi den antar at elementet x som settes inn ikke allerede er tilstede i tabellen. Vi kan sjekke denne antagelsen ved å søke etter et element med *key* lik $x.key$ før vi setter inn x. **For søk vil worst-case kjøretid være proporsjonal med lengden til listen $\Theta(n)$. DELETE prosedyren har kjøretid $O(1)$ dersom listen er dobbelt linked** og **$O(n)$ dersom listen er singel linked.** DELETE tar elementet x som input og ikke k, slik at vi slipper å søke etter x først. Dersom listen er singel-linked må vi først finne x i listen, slik at vi kan oppdatere *next* til elementet foran og dermed blir kjøretiden proporsjonal med lengden til listen, altså $O(n)$.

Analyse av hashing med chaining

Vi skal se hvor lenge hashing bruker på å søke etter et element med en gitt *key*. For en hash tabell med m luker som lagrer n elementer, definerer vi **lastfaktoren $\alpha = n/m$**, altså gjennomsnittlig antall elementer lagret i en chain. Worst-case for hashing er svært dårlig, siden det innebærer at alle n *keys* hasher til samme luke og lager en liste med lengde n. Worst-case kjøretiden blir dermed $\Theta(n)$ pluss tiden det tar for å regne ut hash funksjonen.. **Vi bruker ikke hash tabeller for worst-case tilfellet!**

Average-case kjøretid for hashing bestemmes av hvor god hash funksjonen er til å distribuere settet av *keys* blant de m lukene. Vi antar at sannsynligheten for at et element hasher inn i en luke er lik for alle lukene, uavhengig av hvor andre elementer har hashet. Dette kalles **simpel uniform hashing**. For $j = 0, 1, \dots, m-1$ vil lengden til listen $T[j]$ være n_j, slik at $n = n_0 + n_1 + \cdots n_{m-1}$ og forventet verdi for n_j er $E[n_j] = \alpha = n/m$. Vi antar at det tar **$O(1)$ tid for å regne ut hash verdien** $h(k)$, slik at tiden for å søke etter et element med $key\ k$ vil avhenge av lengden $n_{h(k)}$ til listen ved luken $T[h(k)]$. Vi skal se på antall elementer i listen $T[h(k)]$ som algoritmen vil undersøke for å se om et element har $key\ k$. Vi har to tilfeller

1. **Søket er ikke-suksessfullt** – ingen elementer i tabellen har $key\ k$. **Gjennomsnittlig kjøretid er $\Theta(1+\alpha)$** ved simpel uniform hashing med chaining. Forventet tid for å søke etter et element som ikke er i listen $T[h(k)]$ er lik forventet tid for å søke til enden av listen, som har forventet lengde α. Forventet antall elementer som undersøkes er derfor α. Dersom vi legger til tiden for å regne ut hash funksjonen, vil forventet kjøretid være $\Theta(1+\alpha)$.
2. **Søket er suksessfullt** – søket finnet er element med $key\ k$. I dette tilfellet vil **gjennomsnittlig kjøretid være $\Theta(1+\alpha/2)$**. Grunnen til dette er at lineært søk i en ikke-sortert liste vil gjennomsnittlig søke gjennom halvparten av elementene.

Siden det er $\alpha = n/m$ elementer i listen $T[h(k)]$ vil algoritmen derfor søke gjennom $\alpha/2$ elementer før den finner elementet med $key\ k$. Dersom vi legger til tiden for å regne ut hash funksjonen, vil forventet kjøretid være $\Theta(1 + \alpha/2)$.

Dersom antall luker i hash tabellen er proporsjonal med antall elementer i tabellen, vil $n = O(m)$, altså vil det ta $O(m)$ tid å søke gjennom n elementer. Siden søket etter elementet med $key\ k$ vil innebære å søke gjennom listen i luken $T[h(k)]$ og det er gjennomsnittlig $\alpha = n/m$ elementer i denne listen, vil average kjøretid være:

$$\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$$

Average kjøretid for søk er derfor $O(1)$. Siden INSERTION og DELETION tar $O(1)$ tid worst-case, vil alle ordbok operasjonene ved hashing ta $O(1)$ tid average-case.

11.3 Hash funksjoner

Hva utgjør en god hash funksjon?

En god hash funksjon vil tilnærmet tilfredsstille antagelsen om simpel uniform hashing, altså at hver *key* er like sannsynlig å hashe til enhver av de m lukene, uavhengig av hvor andre *keys* har hashet. Denne egenskapen kan sjeldent undersøkes, siden sannsynlighetsdistribusjonen som regel er ukjent og det kan hende at *keys* ikke er uavhengig av hverandre. Noen ganger kjenner vi distribusjonen. Dersom vi vet at *keys* er tilfeldige, reelle nummer som er uavhengig og uniformt distribuert i rekkevidden $0 \leq k < 1$ vil hashfunksjonen $h(k) = \lfloor km \rfloor$ være en simpel uniform hashing.

En god hash funksjon vil minimere sannsynligheten for at like varianter av *keys* hasher til samme luke. Hashingen må være uavhengig av ethvert mønster i dataen. Det kan hende at vi ønsker at "nære" *keys* skal få hash verdier som er svært ulike. Det er viktig at hash funksjonen er en matematisk funksjon og deterministisk, som vil si at $h(k)$ gir samme verdi dersom vi regner den ut flere ganger.

Tolkning av *keys* som naturlige tall

De fleste hash funksjonene antar at universet av *keys* er settet av naturlige tall: $N = \{0, 1, 2, \dots\}$. Dersom *keys* ikke er naturlige tall, vil vi derfor finne en måte å tolke de som det. Vi kan for eksempel tolke en character string som et heltall vha ASCII tabellen, for eksempel vil $pt = (112, 116)$, siden $p = 112$ og $t = 116$ i ASCII character settet.

Divisjonsmetoden

Ved divisjonsmetoden for å lage hash funksjoner vil vi kartlegge en *key* k inn i en av de m lukene, ved å ta resten etter k dividert med m, altså:

$$h(k) = k \bmod m$$

For eksempel dersom hash tabellen har størrelse $m = 12$ og $k = 100$, vil $h(k) = 4$. Siden det krever kun en divisjonsoperasjon vil denne metoden være ganske rask. Verdien til m kan ikke være av potens 2, siden $m = 2^p$ vil gjøre at $h(k)$ kun er de p laveste-orden bits av k og det kan hende at disse ikke er like sannsynlig. Det beste valget for m er som regel

et primtall som ikke er for nær en toer potens. For eksempel dersom hash tabellen skal holde $n = 2000$ character strings og vi takler 3 elementer ved ikke-suksessfullt søk $(2000/3)$, kan vi velge $m = 701$, siden dette er et primtall nær $2000/3$, men ikke nær en toerpotens. Dermed vil $h(k) = 2000 \bmod 701$.

Multiplikasjonsmetoden

Multiplikasjonsmetoden for å lage hash funksjoner opererer med to steg. Først vil vi hente ut fraksjonsdelen av kA, der $0 < A < 1$. Deretter vil vi multiplisere denne med m og ta gulverdien av resultatet. Altså:

$$h(k) = \lfloor m(ka \bmod 1) \rfloor$$

der fraksjonsdelen er gitt av $ka \bmod 1 = kA - \lfloor kA \rfloor$. En fordel med multiplikasjonsmetoden er at verdien til m ikke er kritisk og vil vanligvis være en toerpotens, fordi da blir den lett å implementere på datamaskiner.

Kapittel 17 – Amortisert analyse

En algoritme kan være rask stort sett hele tiden, men ved bestemte tidspunkt går det plutselig mye tregere. Dette kan skje når man behandler spesifikke datastrukturer. I slike tilfeller er det bedre å bruke amortisert analyse, som kan brukes på datastrukturer og algoritmer som har flere operasjoner. **I amortisert analyse vil vi ta gjennomsnittet av tiden som kreves for å utføre en sekvens av datastruktur operasjoner over alle operasjonene som utføres.** Altså vi ser på gjennomsnitt kjøretid per operasjon etter at mange har blitt utført istedenfor kjøretiden for én enkelt operasjon. Amortisert analyse kan brukes for å vise at gjennomsnittlig kostnad for en operasjon er liten, selv om en enkelt operasjon kan være dyr, ved at vi tar gjennomsnittlig kostnad over en sekvens av operasjoner. Forskjellen fra average-case analyse er at ved amortisert analyse vil ikke sannsynlighet være involvert. En amortisert analyse vil garantere average ytelse for hver operasjon i worst-case.

De tre vanligste teknikkene i amortisert analyse er:

1. **Aggregat analyse** – vi bestemmer en øvre grense $T(n)$ på den totale kostnaden for en sekvens av n operasjoner. Gjennomsnittlig kostnad per operasjon er da $T(n)/n$, og dette er den amortiserte kostnaden per operasjon (derfor lik for alle operasjoner).
2. **Accounting metoden** – bestemmer en amortisert kostnad for hver operasjon, siden denne kan variere avhengig av type operasjon. Denne metoden vil overbelaste noen operasjoner tidlig i sekvensen, slik at dette fungerer som ”forhåndsbetalt kreditt” som vil betale for operasjoner senere i sekvensen.
3. **Potensial metoden** – bestemmer en amortisert kostnad for hver operasjon og kan overbelaste tidlige operasjoner for å kompensere for underbelastet operasjoner senere i sekvensen. Denne metoden bruker kreditt i form at ”potensiell energi” til datastrukturen som et hele.

Amortisert analyse lar oss gå innsikt i en bestemt datastruktur, noe som kan hjelpe oss til å optimalisere designet. Amortisert analyse vil ofte være bedre enn worst-case analyse fordi worst-case kan være altfor pessimistisk.

17.4 Dynamiske tabeller

Vi vil ikke alltid vite på forhånd hvor mange objekter en applikasjon vil lagre i en tabell. Det kan hende vi tildeler rom til en tabell for å senere finne ut at dette ikke var nok. Da må vi omfordele tabellen til en større størrelse og kopiere alle objektene lagret i den originale tabellen inn i den nye, større tabellen. Dersom mange objekter har blitt slettet fra en tabell kan det på samme måte være nyttig å omfordele tabellen til en mindre størrelse. Vi kaller dette for **å dynamisk utvide og kontrahere en tabell**. **Den amortiserte kostnaden for innsetting og sletting er kun $O(1)$,** selv om den faktiske kostnaden til en operasjon er stor når den trigger en ekspansjon eller en kontraksjon.

Vi antar at den dynamiske tabellen støtter operasjonene:

- TABLE-INSERT – vil sette inn et element som okkuperer en enkelt luke i tabellen, altså rom for ett element
- TABLE-DELETE – vil fjerne et element fra tabellen og frigjør dermed en luke

Datastrukturmetoden som brukes kan være en array, stack, heap eller hash tabell. **Lastfaktoren $\alpha(T)$** hos en ikke-tom tabell T er antall enheter som er lagret i tabellen delt på tabellstørrelsen (dvs. antall luker). En tom tabell har størrelse 0 og lastfaktor 1.

Tabell utvidelse

Vi antar at tabellen er en array av luker og at den fylles opp når alle lukene er brukt eller lastfaktoren er 1. I mye programvare vil vi få en feilmelding når vi prøver å sette inn et element i en full tabell. I dette tilfellet derimot skal **innsetting av en enhet i en full tabell føre til at tabellen utvides/ekspanderes ved å tildele en ny tabell med flere luker enn det den gamle tabellen hadde**. Tabellen må alltid ligge i minnet, så derfor må vi tildele en ny array for den større tabellen og deretter kopiere elementene fra den gamle tabellen inn i den nye. En vanlig metode er at **den nye tabellen har dobbelt så mange luker som den gamle**. Hvis den eneste operasjonen tabellen kan utføre er innsetting vil lastfaktoren alltid være minst $1/2$, slik at mengde bortkastet rom aldri overgår halvparten av det totale rommet i tabellen.

Dersom T er tabellen vil vi bruke følgende kommandoer, $T.table$ gir en peker til lagringsblokken som representerer tabellen, $T.num$ gir antall enheter i tabellen og $T.size$ gir totalt antall luker i tabellen. Tom tabell har $T.num = T.size = 0$.

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2      allocate T.table with 1 slot
 3      T.size = 1
 4  if T.num == T.size
 5      allocate new-table with 2 · T.size slots
 6      insert all items in T.table into new-table
 7      free T.table
 8      T.table = new-table
 9      T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

TABLE-INSERT – setter inn element i dynamisk tabell

Denne metoden har to "insert" prosedyrer, TABLE-INSERT prosedyren selv og den elementære innsettingen i en tabell i linje 6 og 10. Dersom tabellstørrelsen er 0 må det tildeles en lagringsblokk for tabellen T, som har én luke og størrelsen må settes lik 1. Dersom tabellen er full, dvs. antall elementer er lik antall luker, må det tildeles en ny tabell. Denne tabellen har dobbelt så mange luker som den gamle og alle elementer fra den gamle tabellen må kopieres inn i den nye tabellen. Lagringsblokken må oppdateres til å lagre denne nye tabellen og størrelsen må dobles. Dermed vil vi ha en tabell som har plass til neste element x, som så settes inn i tabellen ved at den settes inn i lagringsblokken til tabellen. Antall elementer må dermed økes med 1. Linjene 5-9 er selve ekspansjonen som må utføres dersom vi prøver å sette inn et element i en tabell som er full.

Merk: i forelesning brukes en annen måte for å vise at ekstra arbeid per innsetting er $\Theta(1)$.

Vi skal analysere en sekvens av n TABLE-INSERT operasjoner på en initiell tom tabell og ønsker å finne kostnaden c_i for operasjon i. Dersom tabellen har plass til det nye elementet vil $c_1 = 1$, siden vi kun trenger å utføre én elementær innsetting ved linje 10. Dersom tabellen er full og en ekspansjon utføres, vil $c_i = i$. Kostnaden vil være 1 for den elementære innsettingen ved linje 10 pluss $i - 1$ for elementene som må kopieres fra den gamle tabellen til den nye tabellen i linje 6. Dersom vi utfører n operasjoner vil worst-case kostnad for en operasjon være $O(n)$, som fører til en øvre grense på $O(n^2)$ for total kjøretid ved n operasjoner. Denne grensen er ikke tett, fordi vi vil sjeldent utvide tabellen ila n TABLE-INSERT operasjoner. Operasjon i vil kun føre til en ekspansjon når $i - 1$ er nøyaktig en toerpotens. Den amortiserte kostnaden til en operasjon er $O(1)$, som vi kan vise med aggregat analyse:

kostnaden for operasjon i er:

$$c_i = \begin{cases} i, & \text{hvis } i-1 \text{ er en toerpotens} \\ 1, & \text{ellers} \end{cases}$$

Den totale kostnaden for n TABLE-INSERT operasjoner er derfor:

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n$$

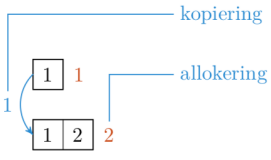
Siden total kostnad av n TABLE-INSERT operasjoner er begrenset av $3n$, vil den amortiserte kostnaden til en enkelt operasjon være maks 3, altså $\Theta(1)$.

Forelesning 2

To <u>svært</u> viktige summer er:

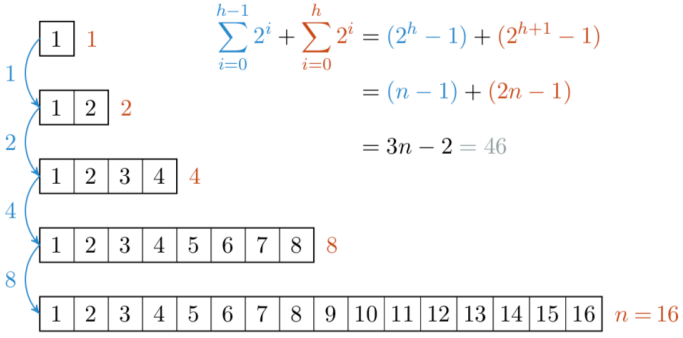
$$\sum_{i=0}^{n-1} \boldsymbol{i} = \frac{\boldsymbol{n(n-1)}}{\boldsymbol{2}} = 1 + 2 + 3 + +4 \ldots$$

$$\sum_{i=0}^{h-1} \boldsymbol{2^i} = \boldsymbol{2^h - 1} = 1 + 2 + 4 + 8 + \cdots$$



Dynamiske tabeller VIKTIG

Når vi vil sette inn et element i en hashtabell, stakk eller kø som er full må vi allokere nytt minne og kopiere inn elementene. Dette tar lineær tid, så derfor vil vi gjøre det sjeldent. På figuren til venstre kan vi se at for hvert trinn vil antall elementer som settes inn før ny tabell blir full og antall elementer som kopieres fra gammel tabell bli doblet. Vi bruker at $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ og finner at total kostnad er $3n - 2$, siden $h = \lg n$. Aggregat analyse (dvs. deler total kostnad på n) gir at amortisert kostnad for innsetting av element i dynamisk tabell er:



$$\sum_{i=0}^{h-1} 2^i + \sum_{i=0}^{h} 2^i = (2^h - 1) + (2^{h+1} - 1)$$
$$= (n-1) + (2n-1)$$
$$= 3n - 2 = 46$$

$$\frac{3n-2}{n} = \Theta(1)$$

Forelesning 3 – Splitt og hersk

Rekursiv dekomponering er kanskje den viktigste ideen i hele faget, og designmetoden splitt og herk er en grunnleggende utgave av det. Denne metoden går ut på å dele instansen i mindre biter, løse problemet rekursivt for disse og deretter kombiner løsningene.

- **Forstå designmetoden *divide-and-conquer* (splitt og hersk)**
- Forstå *maximum-subarray*-problemet med løsninger
- Forstå BISECT og BICSECT' (Appendix C)
- Forstå MERGE-SORT
- Forstå QUICKSORT og RANDOMIZED-QUICKSORT
- **Kunne løse rekurrenser med substitusjon, rekursjonstrær og masterteoremet**
- **Kunne løse rekurrenser med *iterasjonsmetoden* (Appendix B)**
- Forestå hvordan variabelskifte fungerer

Kapittel 2.3 – Design av algoritmer

Det finnes mange designteknikker for algoritmer. For eksempel for innsettingssortering brukte vi en ***incremental*** tilnærming, der vi satt elementet $A[j]$ inn i den sorterte subarrayen $A[1, \dots, j-1]$ ved den riktige plassen, slik at det blir laget en sortert array. En alternativ designmetode er divide-and-conquer, og vi skal bruke denne til å lage en sorteringsalgoritme som har worst-case kjøretid som er mye lavere enn den for innsettingssortering.

Divide-and-conquer tilnærming

Mange algoritmer er **rekursive** som vil si at de løser et gitt problem ved å kalle på seg en eller flere ganger for å løse delproblemer. Disse følger vanligvis en **divide-and-conquer** tilnærming, som vil si at **problemet deles opp i mindre delproblemer som ligner det originale problemet, men er mindre i størrelse. Delproblemene blir løst rekursivt og løsningene blir deretter kombinert for å lage en løsning på det originale problemet**. Denne metoden involverer tre steg:

- **Divide** – splitt problemet inn i et antall delproblemer som er mindre instanser av samme problem
- **Conquer** – hersk delproblemene ved å løse dem rekursivt. Hvis de er tilstrekkelig små kan de også løses rett frem
- **Combine** – slå sammen løsningene på delproblemene for å lage en løsning på det originale problemet.

Merge-sort algoritmen

Merge-sort algoritmen følger splitt og hersk metoden som følger:

- **Divide** – sekvensen av n elementer som skal sorteres deles inn i delsekvenser med $n/2$ elementer hver.
- **Conquer** – de to delsekvensene sorteres rekursivt vha merge sort
- **Combine** – de to sorterte delsekvensene slås sammen (merge) for å produsere den sorterte sekvensen.

Rekursjonen vil "nå bunnen" når delsekvensen som skal sorteres har lengde 1, fordi da vil sekvensen være sortert. Hovedoperasjonen i merge sort er kombinasjonen av to sorterte sekvenser. Dette blir gjort ved å kalle på en hjelpemetode $MERGE(A, p, q, r)$, der A er en array og p, q og r er indekser i arrayen slik at $p \leq q < r$. **Metoden antar at $A[p..q]$ og $A[q+1..r]$ er sortert og vil fusjonere disse for å danne en enkelt sortert array $A[p..r]$.**

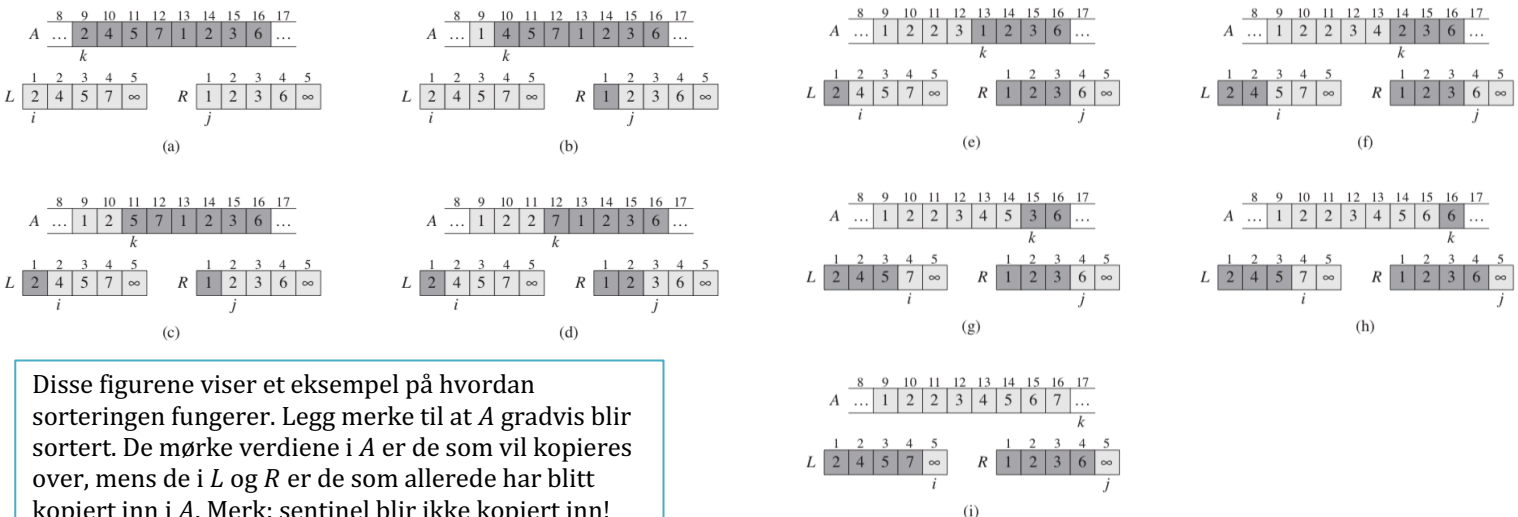
MERGE prosessen tar tiden $\Theta(n)$, der $n = r - p + 1$ er antall elementer som blir fusjonert. Første element fra $A[p..q]$ blir sammenlignet med første element fra $A[q+1..r]$, og det minste blir fjernet fra subarrayen og plassert ved første posisjon i $A[p..r]$. Deretter blir elementet etter det som ble fjernet fra den ene arrayen sammenlignet med første element fra den andre, og det minste blir fjernet fra subarrayen og plassert ved andre posisjon i $A[p..r]$. Dette vil fortsette helt til alle elementene er slått sammen. Hvert steg tar konstant tid, siden vi kun sammenligner to elementer, og vi utfører n steg, så fusjoneringen tar $\Theta(n)$ tid.

Etter hvert vil den ene subarrayen bli tom, mens den andre ikke er det. **For å slippe å sjekke om en av subarrayene er tom ved hvert steg plasserer vi et sentinel element ved slutten av hver subarray.** Denne kan ha verdi ∞, slik at den ikke kan være mindre enn elementet i den andre subarrayen. Når begge subarrayene har denne verdien, betyr det at fusjoneringen er ferdig. Siden vi vet at $n = r - p + 1$ vil vi vite nøyaktig når dette skjer.

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

<u>MERGE – vil slå sammen to sorterte subarrays</u>

Denne metoden tar inn en matrise $A[p..r]$ som består av to sorterte subarrays $A[p..q]$ og $A[q+1..r]$ som skal slås sammen. Vi lar $n_1 + 1$ være lengen til subarray L og $n_2 + 1$ være lengden til subarray R (+1 for å få plass til sentinel verdiene). Deretter må vi lage disse og fylle de opp med korresponderende verdier fra A matrisen. Legg merke til at vi bruker ulike indekser for disse to og legger til sentinel verdiene ved siste posisjon i begge subarrayene. Deretter begynner selve fusjoneringen. Vi skal fylle alle elementene i $A[p..r]$, så derfor må k gå fra p til r. Dersom elementet fra L er mindre eller lik (sørger for stabil sortering) elementet fra R vil vi sette dette elementet ved posisjon k og øke indeksen til L med én. Det samme gjelder for R. Dermed har A blir sortert.



Disse figurene viser et eksempel på hvordan sorteringen fungerer. Legg merke til at A gradvis blir sortert. De mørke verdiene i A er de som vil kopieres over, mens de i L og R er de som allerede har blitt kopiert inn i A. Merk: sentinel blir ikke kopiert inn!

Loop invarianten er at før hver iterasjon av for-loopen på linje 12-17 vil subarrayen $A[p \ldots k-1]$ inneholde de minste elementene fra L og R i en sortert rekkefølge. Samtidig vil $L[i]$ og $R[j]$ (altså de neste elementene) være de minste elementene som fortsatt ikke har blitt kopiert inn i A. Vi må vise at denne loop invarianten holder før første iterasjon, blir vedlikeholdt ved hver iterasjon og gir riktig endelig resultat når loopen terminerer:

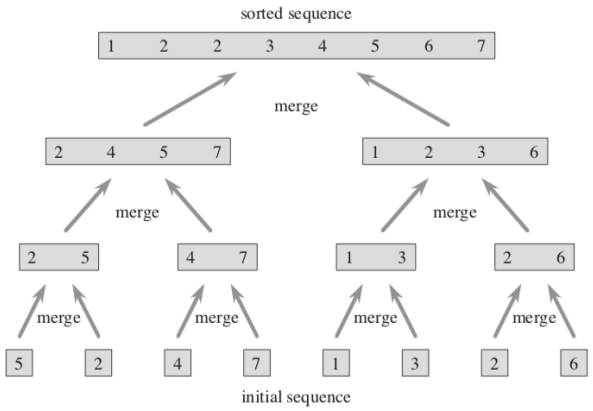
- **Initialisering** – før første iterasjon vil $k = p$ slik at $A[p \ldots k-1]$ er tom og vil inneholde de $k - p = 0$ minste elementene fra L og R. Siden $i = j = 1$ vil også $L[i]$ og $R[j]$ være de minste elementene som enda ikke har blitt kopiert inn i A.
- **Vedlikehold** – dersom $L[i] \leq R[j]$ vil $L[i]$ være det minste elementet som enda ikke er kopiert inn i A. $A[p..k-1]$ inneholder de $k - p$ minste elementene, så når $L[i]$ kopieres inn vil $A[p..k]$ inneholde de $k - p + 1$ minste elementene. Når k og i økes vil loopinvarianten reetableres til neste iterasjon. Samme gjelder for når $L[i] < R[j]$.
- **Terminering** – ved terminering vil $k = r + 1$, altså $r = k - 1$. Av loop invarianten vil $A[p..k-1] = A[p..r]$ inneholde de minste elementene $k - p = r - p + 1$ av L og R i sortert rekkefølge. Det er kun sentinel verdiene som ikke er kopiert inn i A.

MERGE prosedyren tar $\Theta(n)$ tid fordi hver linje tar konstant tid, mens for-loopen ved linje 4-7 tar $\Theta(n_1 + n_2) = \Theta(n)$ tid og for-loopen ved linje 12-17 tar $\Theta(n)$ tid. Husk at vi ignorerer koeffisienter, så dette blir til sammen kjøretid $\Theta(n)$.

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

MERGE-SORT – vil sortere en sekvens vha divide-and-conquer

Dersom $p \geq r$ vil subarrayen ha maksimum lengde 1 og vil derfor allerede være sortert. Derfor skal vi kunne kjøre metodene hvis $p < q$. Vi setter q til å være midten av sekvensen. Deretter kaller vi MERGE-SORT på hver av halvpartene. Disse rekursive kallene vil fortsette helt til $p \geq q$, altså at subarrayen kun består av et element. Da vil de to MERGE-SORT kallene gi to subarrays med ett element hver, og MERGE vil slå de sammen til én subarray med to elementer. De to MERGE-SORT kallene i laget over vil gi to subarrays med to elementer hver, og MERGE vil slå de sammen til én subarray med fire elementer. De to MERGE-SORT kallene i laget over vil gi to subarrays med fire elementer hver, og MERGE vil slå de sammen til én subarray. Dermed er hele arrayen A sortert!



For å sortere en hel sekvens kan vi bruke MERGE-SORT$(A, 1, A.length)$.

Analyse av divide-and-conquer algoritmer

Når en algoritme inneholder et rekursivt kall kan vi ofte beskrive kjøretiden vha en rekurrensligningen (gjentagelse), som beskriver total kjøretid til et problem med størrelse n vha. kjøretiden til mindre input: $T(n) = T(n-1) + n$. Rekurrenser dukker opp når en algoritme har rekursive kall.

$T(n)$ er kjøretiden til et problem med størrelse n. Dersom denne størrelsen er tilstrekkelig liten vil $n \leq c$ for en konstant c og løsningen tar konstant tid: $\Theta(1)$. Anta at

vi deler problemet inn i a delproblemer, som hver er $1/b$ av størrelsen til det originale problemet (merge-sort: $a = b = 2$). Det vil ta $T(n/b)$ tid for å løse et delproblem av størrelsen n/b, og dermed tar det tiden $aT(n/b)$ å løse a delproblemer. Dersom det tar $D(n)$ tid å dele opp problemet i delproblemer og $C(n)$ tid for å kombinere løsningene til delproblemene til en løsning på det originale problemet, får vi rekurrensen:

$$T(n) = \begin{cases} \Theta(1) & n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), & \text{ellers} \end{cases}$$

Kapittel 4 viser hvordan man løser rekurrenser på denne formen.

Analyse av merge sort

Vi antar at de originale problemet er en toerpotens, selv om mergesort også fungerer når antall elementer er et oddetall. Da vil hver oppdeling gi to delsekvenser av størrelse $n/2$. Vi har følgende argumenter for å sette opp rekurrensen for $T(n)$, som er worst-case kjøretid for merge sort på n tall. Merge sort på et element tar konstant tid og når vi har $n > 1$ element deler vi opp kjøretiden som følger:

- **Divide** – dette steget vil kun regne ut midten av subarrayen, noe som tar konstant tid. Altså vil $D(n) = \Theta(1)$
- **Conquer** – dette steget vil rekursivt løse to delproblemer av størrelsen $n/2$ og vil derfor bidra med $2T(n/2)$ til kjøretiden.
- **Combine** – vi har allerede sett at MERGE prosessen tar $C(n) = \Theta(n)$ tid.

Vi har at $D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$. Dersom vi legger dette til $2T(n/2)$, vil vi få rekurrensen for worst-case kjøretid $T(n)$ hos merge-sort:

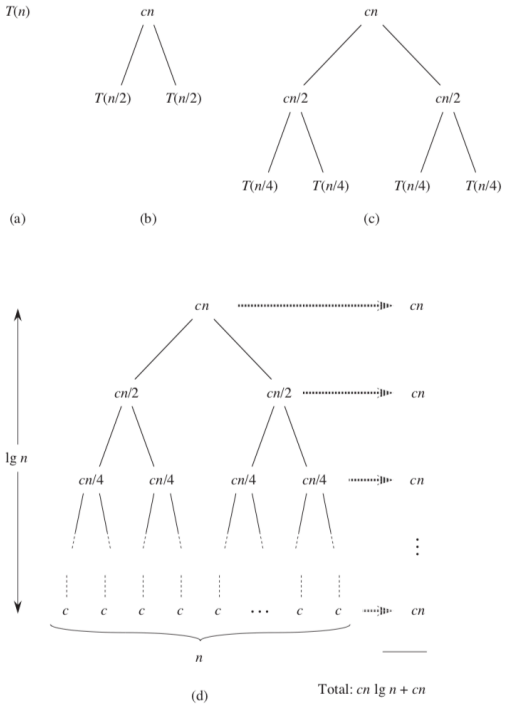
$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(n/2) + \Theta(n) & n > 1 \end{cases}$$

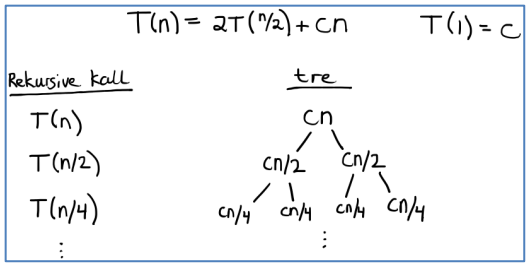
Kapittel 4 vil lære oss om master teoremet som lar oss vise at $T(n) = \Theta(n \lg_2 n)$. Siden logaritme funksjonen vokser tregere enn en lineær funksjon for store input, vil **merge sort $\Theta(n \lg n)$) ha lavere worst-case kjøretid enn innsettingssortering ($\Theta(n^2)$).**

Vi kan også løse rekurrensen ved å se på:

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

Denne kan løses vha et **rekursjonstre** (se figur). Vi antar at n er en toerpotens. Når vi lager et rekursjonstre vil vi for hvert nivå av rekursive kall se på hvor mange operasjoner som utføres utenom det rekursive kallet ($T(\dots)$) pluss antall rekursive kall (gis av koeffisienten til $T(\dots)$). Figur a: $T(n)$ som kan utvides til en ekvivalent rekurrens (b). Figur b: for et rekursivt kall $T(n)$ vil det utføres cn operasjoner pluss to rekursive kall $T(n/2)$. Vi kan videre utvide $T(n/2)$. Figur c: for et rekursivt kall $T(n/2)$ vil det utfører $cn/2$





operasjoner pluss to rekursive kall $T(n/4)$. Neste steg vil være et rekursivt kall $T(n/4)$ der det vil utføres $cn/4$ operasjoner pluss to rekursive kall $T(n/8)$. Vi vil fortsette å utvide hver node i treet ved å bryte den inn i deler bestemt av rekurrensligningen, helt til størrelsene på problemene blir 1 og har kostnad c. Figur (d) viser det resulterende rekursjonstreet.

Neste oppgave er å summere kostnadene ved hvert nivå av treet. Det øverste nivået har total kostand cn, det andre har $cn/2 + cn/2 = cn$, det tredje har $cn/4 + cn/4 + cn/4 + cn/4 = cn$, osv. Generelt vil nivå i ha 2^i noder som hver har kostnad $cn/2^i$ og dermed vil nivå i ha total kostnad $2^i \cdot cn/2^i = cn$. Det nederste nivået har n noder som hver har kostnad c og dermed vil nederste nivå ha total kostnad cn. Totalt antall nivåer i rekursjonstreet er $\lg_2 n + 1$ (mer kapittel 4), der n er antall blader eller input størrelsen. Induktivt bevis (grunntilfellet og $n \to n + 1$): dersom basen ($n = 1$) er det eneste nivået vil $\lg 1 + 1 = 1$ nivå (riktig) og dersom antall nivåer ved $n = 2^i$ er $\lg 2^i + 1 = i + 1$, vil antall nivåer ved $n + 1 = 2^{i+1}$ være $\lg 2^{i+1} + 1 = (i + 1) + 1 = n + 1$ (riktig).

For å regne ut antall kostnader som rekurrensen representerer kan vi summere kostnadene ved alle nivåene. Rekursjonstreet har $\lg n + 1$ nivåer som hver har kostnad cn. Derfor vil total kostnad være $cn(\lg n + 1)$. Dersom vi ignorerer lavere ordens ledd og koeffisienter får vi at **kjøretiden til merge-sort er $\boldsymbol{\Theta(n \lg n + n) = \Theta(n \lg n)}$.**

Se verifikasjon side 57

Pensumoppgave – 2.3-5

Vi ser tilbake på søkeproblemet og observerer at dersom sekvensen A er sortert, kan vi sjekke midtpunktet av sekvensen mot v og eliminere halvparten av sekvensen fra videre undersøkelse. Den binære søk algoritmen vil gjenta denne prosedyren, altså halvere størrelsen til gjenværende del av sekvensen for hvert rekursivt kall. Skriv pseudokoden, enten iterativ eller rekursiv, for binært søk og vis at worst-case kjøretid er $\Theta(lg\ n)$.

Vi skal lage metoden BINARY-SEARCH som har input som er en sekvens av tall $A = \langle a_1, a_2, \ldots, a_n \rangle$ og en verdi v som vi søker etter. Denne metoden skal returnere en indeks i, slik at $v = A[i]$ eller NIL dersom A ikke inneholder v. Metoden skal undersøke elementene ved å dele arrayen i to helt til den finner elementet eller det ikke går an å dele subarrayen. Pseudokoden blir:

BINARY-SEARCH*(A, v)*

```
low = 1
high = A.length

while (low ≤ high)
    mid = ⌊(low + high)/2⌋
    if (A[mid] == v)
        return mid
    else if (A[mid] < v)
        low = mid + 1
    else
        high = mid − 1

return NIL
```

Iterativ BINARY-SEARCH

BINARY-SEARCH*(A, v, low high)*

```
if (low ≤ high)
    mid = ⌊(low + high)/2⌋
    if v == A[mid]
        return mid
    else if (A[mid] < v)
        BINARY-SEARCH(A, v, mid + 1, high)
    else
        BINARY-SEARCH(A, v, low, mid − 1)
else return NIL
```

Rekursiv BINARY-SEARCH

For å finne kjøretiden til denne algoritmen må vi først finne rekurrensen siden dette er en divide-and-conquer algoritme:

- **Divide** – dette steget vil kun regne ut midten av subarrayen, noe som tar konstant tid. Altså vil $D(n) = \Theta(1)$
- **Conquer** – dette steget vil rekursivt søke gjennom et delproblem av størrelsen $n/2$ og vil derfor bidra med $T(n/2)$ til kjøretiden.
- **Combine** – det er ingen kombinering i søke algoritmer

Rekurrensen blir derfor:

$$T(n) = T(n/2) + \Theta(1) = 2T(n/2) +$$

Dette vil gi opphav til et rekursjonstre der kostnaden per nivå er en konstant (c). Størrelsen til problemet ved nivå i vil være $n/2^i$. Ved bladnivå (bunnen) vil størrelsen til delproblemene være 1, slik at $n/2^i = 1$ og $i = \lg_2 n$. Høyden til treet er derfor $\lg_2 n$, slik at den totale kostnaden blir $c \lg_2 n = \Theta(\lg n)$.

Kapittel 4 – Divide-and-conquer

I splitt og hersk metoder vil vi løse et problem rekursivt vha følgende tre steg:

- **Divide** – splitt problemet inn i et antall delproblemer som er mindre instanser av samme problem
- **Conquer** – hersk delproblemene ved å løse dem rekursivt. Hvis de er tilstrekkelig små kan de også løses rett frem
- **Combine** – slå sammen løsningene på delproblemene for å lage en løsning på det originale problemet.

Rekursiv tilfellet er når delproblemene er store nok til å løses rekursivt, mens **base tilfellet** er når delproblemet blir så lite at det ikke lenger kan løses rekursivt (bunnen er nådd). Delproblem som ikke ligner på det originale problemet løses som en del av kombineringen.

Rekurrenser

Rekurrenser gir en naturlig måte å karakterisere kjøretiden til splitt og hersk algoritmer, og det er en ligning eller ulikhet som beskriver en funksjon vha funksjonsverdiene på mindre input. Rekurrenser kan ta mange former, for eksempel hvis den rekursive algoritmen gir to delproblemer med ulik størrelse (2/3 og 1/3) og oppdelingen og kombineringen tar lineær tid, vil rekurrensen bli $T(n) = T(2n/3) + T(n/3) + \Theta(n)$. Delproblemene må ikke alltid være en fraksjon av problemet, for eksempel vil en rekursiv versjon av lineær søk lage et delproblem som inneholder et mindre element, slik at $T(n) = T(n-1) + \Theta(1)$.

Vi skal se tre metoder for å løse rekurrenser, altså finne asymptotiske grenser Θ eller O:

- **Substitusjonsmetoden** – vi tipper en grense og bruker induksjon for å sjekke om dette er riktig
- **Rekursjonstre metoden** – omdanner rekurrensen til et tre der nodene representerer kostnadene ved de ulike nivåene av rekursjonen. løses
- **Master metoden** – gir grenser for rekurrensen på formen $T(n) = aT(n/b) + f(n)$. Brukes på tilfeller der det er a delproblemer hver med størrelse n/b og oppdelingen og kombinasjonen tar $f(n)$ tid.

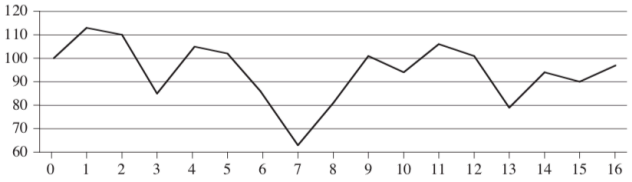
Dersom rekurrensen bruker $\leq$ istedenfor $=$, vil vi finne en øvre grense O istedenfor Θ! Dersom vi har $\geq$ må vi bruke en nedre grense Ω!

Tekniske detaljer ved rekurrenser

Vi utelater noen tekniske detaljer, for eksempel dersom n er et oddetall ved MERGE-SORT vil delproblemene får størrelse $\lceil n/2 \rceil$ og $\lfloor n/2 \rfloor$, slik at rekurrensen for worst-case blir:

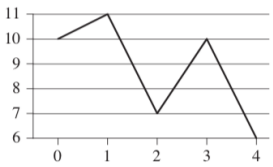
$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & n > 1 \end{cases}$$

Vi vil også ofte utelate grensebetingelser, slik som at $T(n) = \Theta(1)$ for tilstrekkelig liten n. Ofte vil vi ikke ta hensyn til disse under beregningen og heller bestemme senere om de spiller noen rolle. Som regel vil de ikke ha noen signifikant rolle i splitt og hersk algoritmer.



4.1 Maksimum-subarray problem

Vi har et problem der vi kan kjøpe et produkt for en pris og selge det ved et senere tidspunkt til en annen pris. Vi kjenner hvordan prisen vil endres over en periode (se figur) og ønsker å tjene mest mulig. I noen tilfeller vil vi få maksimum profitt ved å kjøpe produktet ved lavest pris, og selge det ved høyest etterfølgende pris, mens i andre tilfeller vil den høyeste prisen komme før den laveste prisen, så dette blir ikke mulig (se figur til venstre).



Brute-force løsning

Brute-force løsningen er å prøve alle mulige par med kjøp og salg datoer der kjøpet er før salget. En periode av n datoer har $\binom{n}{2}$ slike par som er $\Theta(n^2)$. Dersom evalueringen av hvert par tar konstant tid, vil denne løsningen ta $\Omega(n^2)$ tid.

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

En transformasjon

Vi skal se på input på en annen måte. Vi ønsker å finne en sekvens av dager der netto endring fra første til siste dag er maksimert. I stedet for å se på de daglige prisene, vil vi se på den daglige endringen i prisen, slik at endringen på dag i vil være forskjellen i pris ved dag $i - 1$ og dag i. Vi kan se disse verdiene i Change raden i tabellen og de kan plasseres i en array A. Dermed kan vi finne den sammenhengende subarrayen der verdiene har størst sum og kaller denne for **maksimum subarray** (se figur). For å maksimere profitten må du kjøpe produktet rett før dag 8 og selge det etter dag 11, slik at du tjener 43 kr.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

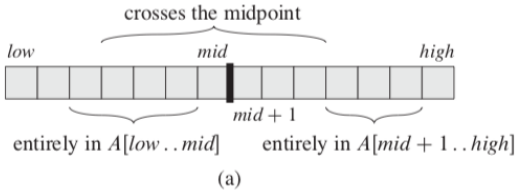
maximum subarray

Vi kaller dette maksimum subarray problemet, og dersom det løses vha brute force vil kjøretiden være $\Theta(n^2)$. Vi ønsker nå å finne en mer effektiv løsning til dette problemet. Dersom denne arrayen ikke hadde inneholdt negative tall, ville helle arrayen gitt den største summen, så problemet er kun interessant når det er negativ og positive tall!

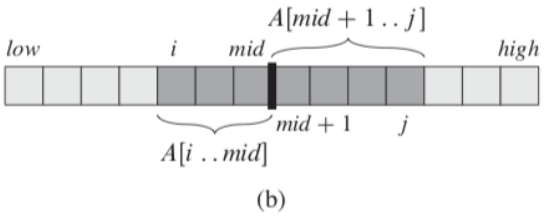
En løsning vha divide-and-conquer

Vi ønsker å finne maksimum subarray fra arrayen $A[low \ldots high]$, og splitt og hersk metoden går ut på at vi deler denne arrayen inn i to subarrays med så lik størrelse som mulig. Altså vil vi finne et midtpunkt (mid) til arrayen og ser på subarrayene: $A[low \ldots mid]$ og $A[mid + 1 \ldots high]$. Alle sammenhengende subarrays må ligge i en av følgende:

- Helt i venstre subarray $A[low \ldots mid]$
- Helt i høyre subarray $A[mid + 1 \ldots high]$
- I subarray som krysser midtpunktet



En maksimum subarray må altså ligge i eksakt en av disse plassene. Derfor kan vi finne maksimum subarray ved å finne subarray som gir størst sum i hver av de tre plassene og deretter sammenligne disse. Vi kan finne maksimum subarrays hos $A[low \ldots high]$ og $[mid + 1 \ldots high]$ vha rekursjon, siden disse er delproblemer som ligner det originale problemet. Derfor gjenstår det kun å finne maksimum subarray som krysser midtpunktet. Denne kan ikke finnes med rekursjon, fordi det har et ekstra krav om at subarrayen må krysse midtpunktet. Enhver subarray som krysser midtpunktet vil være laget av to subarrays $A[i \ldots mid]$ og $A[mid + 1 \ldots j]$, se figur (b). Derfor kan vi finne maksimum subarrays på formen $[i \ldots mid]$ og $A[mid + 1 \ldots j]$ og deretter kombinere disse.




```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4      sum = sum + A[i]
 5      if sum > left-sum
 6          left-sum = sum
 7          max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11      sum = sum + A[j]
12      if sum > right-sum
13          right-sum = sum
14          max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

FIND-MAX-CROSSING-SUBARRAY

Vi setter *left-sum* og *right*-sum lik minus uendelig for å sikre at *sum* ikke kan være mindre enn disse ved første iterasjon. Vi begynner med å finne maksimum subarray på venstre side av midtpunktet. *sum* vil være summen av elementene fra mid til i og dersom summen blir større enn tidligere største sum gitt av *left-sum*, skal *left-sum* oppdateres til denne sumverdien og *max-left* settes lik indeksen ved denne iterasjonen. Dermed vil *left-sum* etter for-løkken være verdien til maksimum subarray som ligger mellom *max-left* og *mid*. Vi gjør det samme for høyre side av midtpunktet. Legg merke til at loopen går mot venstre for venstre side av midtpunktet og til høyre for høyre side av midtpunktet. Metoden vil returnere *max-left* og *max-right* som er hhv. start- og sluttindeks til maksimum subarray og *left-sum + right-sum* som er verdien til maksimum subarray.

Dersom arrayen $A[low..high]$ inneholder $n = high - low + 1$ elementer, vil denne prosedyren ta $\boldsymbol{\Theta(n)}$ **tid**, siden hver iterasjon av for-loopene tar $\Theta(1)$ tid og det er totalt n iterasjoner (første loop har $mid - low + 1$ og andre har $high - mid$, som til sammen blir $n = high - low + 1$).

Dermed kan vi lage divide-and-conquer algoritmen for å løse maksimum subarray problemet:

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
1  if high == low
2      return (low, high, A[low])              // base case: only one element
3  else mid = ⌊(low + high)/2⌋
4      (left-low, left-high, left-sum) =
           FIND-MAXIMUM-SUBARRAY(A, low, mid)
5      (right-low, right-high, right-sum) =
           FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6      (cross-low, cross-high, cross-sum) =
           FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7      if left-sum ≥ right-sum and left-sum ≥ cross-sum
8          return (left-low, left-high, left-sum)
9      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10         return (right-low, right-high, right-sum)
11     else return (cross-low, cross-high, cross-sum)
```

FIND-MAXIMUM-SUBARRAY

Denne metoden vil returnere en tuple som inneholder indeksene og verdien til maksimum subarray. Vi begynner med å teste grunntilfellet der subarrayen har kun ett element, slik at maksimum subarray vil ha verdi lik elementverdien.
Linje 3 er oppdelingen (*divide*) av arrayen. Linje 4 og 5 er de rekursive kallene (*conquer*), altså finner vi maksimum subarray til venstre subarray og høyre subarray. Linje 6-11 er kombineringen (*combine*) av subarrays. Husk at det å finne maksimum subarray til subarrayen som krysser midtpunktet ikke er et delproblem som ligner det originale problemet, og blir derfor regnet som en del av kombineringen. Linje 7 til 11 vil sjekke hvilken av de tre maksimum subarrayene som er størst og vil returnere indeksene og verdien til denne maksimum subarrayen.

Analyse av divide-and-conquer algoritmen

Vi vil sette opp rekurrenser som beskriver kjøretiden til den rekursive FIND-MAXIMUM-SUBARRAY prosedyren. Vi antar at problemstørrelsen er en toerpotens, slik at størrelsen til alle delproblem er heltall. $T(n)$ er kjøretiden til prosedyren på en subarray med n elementer. Linje 1 tar konstant tid, slik at grunntilfellet ($n = 1$) har kjøretid:

$$T(1) = \Theta(1)$$

Det rekursive tilfellet oppstår når $n > 1$. Linje 3 vil ta konstant tid. Siden hver av delproblemene på linje 4 og 5 er på størrelse $n/2$ vil det ta $T(n/2)$ tid å løse hver av de. Siden vi må løse to delproblemer vil de rekursive kallene gi totalt kjøretid $2T(n/2)$. Vi har allerede sett at linje 6 tar $\Theta(n)$ tid og linje 7-11 tar konstant tid $\Theta(1)$. Derfor vil $D(n) + C(n) = \Theta(n) + \Theta(1) + \Theta(1) = \Theta(n)$ og dermed $T(n) = 2T(n/2) + \Theta(n)$. Rekurrensen blir dermed:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(n/2) + \Theta(n) & n > 1 \end{cases}$$

Altså, den samme som for merge sort. Dermed vet vi at denne rekurrensen har løsning $\boldsymbol{T(n) = \Theta(n \lg n)}$. Altså, har divide-and-conquer metoden gitt en algoritme som er asymptotisk raskere enn brute-force algoritmen med kjøretid $\Theta(n^2)$!

4.3 Substitusjonsmetoden

Substitusjonsmetoden kan brukes for å løse rekurrenser med to steg:

1. **Tippe formen til løsningen**
2. **Bruke induksjon for å finne konstanter og vise at løsningen er riktig**

Vi kan bruke denne metoden for å etablere enten en nedre eller en øvre grenser på en rekurrens. For eksempel kan vi ønske å finne en øvre grense for:

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

Side 57 viser en annen fremgangsmåte for induksjonsdelen av substitusjonsmetoden

Siden denne ligner på rekurrensen på forrige side, **tipper vi at den har løsning $T(n) = O(n \lg n)$ og dermed må vi vise grensen $T(n) \leq cn \lg n$ for en konstant c** (merk: $\leq$ siden vi har tippet O). Vi legger merke til det rekursive kallet $T(\lfloor n/2 \rfloor)$ i rekurrensen og ønsker å formulere grensen vha denne. Med $\lfloor n/2 \rfloor$ som input vil grensen bli $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Vi setter dette uttrykket for $T(\lfloor n/2 \rfloor)$ inn i rekurrensen:

$$
\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor) + n \\
T(n) &\leq 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\
&\leq cn \lg(n/2)) + n \\
&= cn \lg n - cn \lg 2 + n \\
&= cn \lg n - cn + n \\
&\leq cn \lg n
\end{aligned}
$$

der siste steg holder for $c \geq 1$. Vi har altså vist at løsningen holder for store n, og neste steg i det induktive beviset er å **vise at løsningen holder for grensebetingelsene**. For denne rekurrensen innebærer det å vise at vi kan velge c høy nok slik at grensen $T(n) \leq cn \lg n$ også gjelder for liten n. Dersom vi ser på $n = 1$ får vi at grensen blir $T(1) \leq 0$, men $T(1) = 1$, altså feiler grunntilfellet i vårt induktive bevis. Vi kan løse dette ved å bruke at **den asymptotiske notasjonen krever at $T(n) \leq cn \lg n$ for $n \geq n_0$, der vi kan velge n_0**! Vi må derfor finne en verdi for n_0 som oppfyller dette kravet, altså som gjør at rekurrensen ikke direkte avhenger av $T(1)$. Ved $n > 3$ vil stegene bli:

$$
\begin{aligned}
T(4) &= 2T(2) + 4 \\
T(5) &= 2T(2) + 5 \\
T(6) &= 2T(3) + 6 \\
T(7) &= 2T(3) + 7 \\
T(8) &= 2T(4) + 8 \\
&\ldots
\end{aligned}
$$

Altså kan vi heller bruke $T(2)$ og $T(3)$ som grunntilfellene i det induktive beviset, slik at $n_0 = 2$. Grunntilfellet for rekurrensen vil fortsatt være $T(1)$, men vi kan altså velge at grunntilfellet til det induktive beviset er $T(2)$ og $T(3)$. Vi har at $T(2) = 3$ og $T(3) = 5$. Neste steg er å vise at vi kan velge c slik at $T(2) \leq c2 \lg 2$ og $T(3) \leq c3 \lg 3$, noe som er tilfellet for $c \geq 2$. Vi har vist at løsningen holder for grunntilfellene og for store n, og dermed kan vi si at gjetningen $T(n) = O(n \lg n)$ er riktig.

Tippe bra

Ingen generell måte, men kan bruke heuristikk (tommelregler vi utvikler basert på tidligere erfaring) og rekursjonstrær. Dersom rekurrensen ligner på en du har sett tidligere, kan du tippe at den har ligningene løsning. For eksempel for $T(n) = 2T(\lfloor n/2 + 17 \rfloor) + n$, er det kun leddet +17 som er annerledes, og ved store n blir denne neglisjerbar. Derfor kan vi tippe at denne har løsning $O(n \lg n)$ og verifisere det med substitusjonsmetoden. En annen måte er å bevis løse øvre og nedre grenser og deretter redusere rekkevidden for usikkerheten. For eksempel for en rekurrens som inneholder begrepet n, kan vi starte med $T(n) = \Omega(n)$ og $T(n) = O(n^2)$. Deretter kan vi gradvis senke øvre grense og øke nedre grense til vi lander på $T(n) = \Theta(n \lg n)$.

Finjusteringer

Noen ganger kan vi tippe riktig, men matten gir feil i induksjonen, noe som ofte skyldes at den induktive antagelsen ikke er sterk nok til å bevise detaljerte grenser. Ofte kan dette fikses ved å trekke fra et ledd med lavere orden.

For eksempel ser vi på:

$$T(n) = 2T(\lfloor n/2 \rfloor) + 1$$

Vi tipper at løsningen er $T(n) = O(n)$ og må derfor vise at $T(n) \leq cn$. Dersom vi ser på $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor$, vil:

$$T(n) \leq 2c\lfloor n/2 \rfloor + 1$$
$$T(n) = cn + 1$$

som ikke gir at $T(n) \leq cn$. Vi kan vise at $T(n) = O(n^2)$, men $O(n)$ er riktig! Vi prøver derfor å trekke fra et ledd med lavere orden, altså vil vårt nye tipp være $O(n - d)$ der $d \geq 0$ er en konstant. Altså: $T(n) \leq cn - d$. Dersom vi ser på $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor - d$, vil:

$$T(n) \leq 2(c\lfloor n/2 \rfloor - d) + 1$$
$$T(n) = cn - 2d + 1$$
$$T(n) \leq cn - d$$

så lenge $d \geq 1$ $(-2d + 1 \leq -d)$ og vi velger c stor nok til å håndtere grensebetingelsene.

Unngå fallgruver

Vanlig å bruke asymptotisk notasjon feil, for eksempel kan vi tippe $T(n) \leq cn$, og si at:

$$T(n) \leq 2(c\lfloor n/2 \rfloor) + n$$
$$T(n) \leq cn + n$$
$$= O(n)$$

som er feil! Her har vi fått at $T(n) \leq cn + n$ og ikke $T(n) \leq cn$. Det er viktig at vi får nøyaktig den grensen vi tipper! Ellers kan vi ikke si at $T(n) = O(n)$.

Variabelskifte VIKTIG

Vi kan bruke algebra for å manipulere ukjente rekurrenser til en form vi har sett tidligere. For eksempel:

$$T(n) = 2T\left(\left\lfloor \sqrt{n} \right\rfloor\right) + \lg n$$

Vi ser bort fra floor funksjonen og setter $m = \lg n$, altså $n = 2^m$. Rekurrensen blir:

$$T(2^m) = 2T\left(2^{m \cdot 1/2}\right) + \lg 2^m$$
$$T(2^m) = 2T\left(2^{m/2}\right) + m$$

Vi endrer notasjonen: $S(m) = T(2^m)$ (dvs. disse rekurrensene har samme verdi, men ulike variabler), slik at $T\left(2^{m/2}\right) = S(m/2)$. Dermed vil:

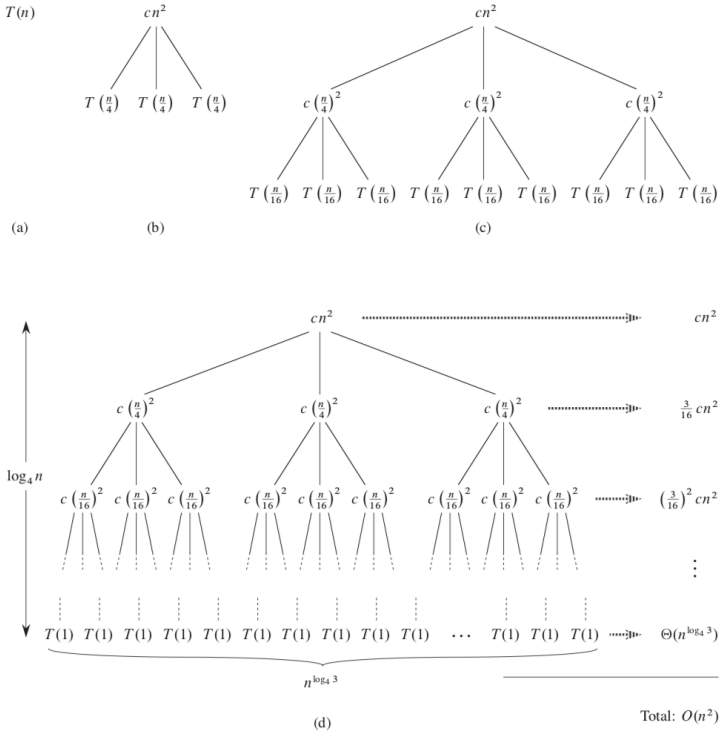
$$S(m) = 2S(m/2) + m$$

Denne ligner på rekurrensen vi så forrige side, og har samme løsning $S(m) = O(m \lg m)$. Siden $S(m) = T(2^m)$, vil også $T(2^m) = O(m \lg m)$. Vi bruker at $m = \lg n$, og får:

$$T(n) = T(2^m) = O(m \lg m) = O(\lg n \lg(\lg n))$$

4.4 Rekursjonstre metoden

Et rekursjonstre kan brukes for å gjette godt i substitusjonsmetoden. I et rekursjonstre vil hver node representere kostnaden til rekursivt kall, også kalt et delproblem. Ved å summere kostnadene på hvert nivå vil vi finne et sett som består av per-nivå kostnader. Ved å summere disse kan vi finne den totale kostnaden til alle nivåene av rekursjonen. Vi skal se hvordan vi kan bruke rekursjonstreet for å gjette løsningen til $\boldsymbol{T(n) = 3T(n/4) + \Theta(n^2)}$**.** Gulv og tak funksjoner har som regel liten betydning i løsningen av rekurrenser, så derfor ser vi bort fra disse. Vi lager derfor et rekursjonstre for $T(n) = 3T(n/4) + cn^2$.



Figuren viser rekursjonstreet. Vi antar at n er en firerpotens slik at størrelsen til delproblemene $n/4^i$ er heltall. Figuren viser følgende:

a) $T(n)$ som skal utvides
b) Representerer rekurrensen som er ekvivalent med $T(n)$. Roten cn^2 er antall operasjoner ved øverste nivå av rekursjonen, mens de tre forgreningene er kostnadene forbundet med delproblemene med størrelse $n/4$. Dette er altså kostnaden ved de tre rekursive kallene: $3T(n/4)$.
c) Kostnaden til de tre rekursive kallene blir videre utvidet. For $T(n/4)$ vil det utføres $c(n/4)^2$ operasjoner pluss tre rekursive kall: $3T(n/16)$.
d) Vi fortsetter utvidelsen av hver node helt til vi når $T(1)$, altså at størrelsen til delproblemet blir 1.

Siden hver utvidelse vil redusere størrelsen til delproblemene med en faktor på 4 vil vi tilslutt nå en grensebetingelse. Størrelsen til delproblem ved nivå i er $n/4^i$, så størrelsen vil bli 1 når $n/4^i = 1$, altså $i = \log_4 n$. Treet vil derfor ha $\log_4 n + 1$ nivåer (0, 1, 2, …, $\log_4 n$).

Neste steg er å bestemme kostnaden til hvert nivå. Kostnaden til hver node ved nivå i vil være $c\left(n/4^i\right)^2$ siden størrelsen til delproblemene ved dette nivået er $n/4^i$. Hvert nivå vil ha tre ganger antall noder som nivået over, så derfor vil antall noder ved nivå i være 3^i. Total kostnad for nivå i blir dermed $3^i\, c\left(n/4^i\right)^2 = (3/16)^i cn^2$. For å finne kostnaden til hele treet må vi summere kostnadene ved hvert nivå. Ved bunnen der $i = \log_4 n$ vil antall noder være $3^{\log_4 n} = n^{\log_4 3}$ som hver har kostnad $T(1)$. Den totale kostnaden blir $n^{\log_4 3}\, T(1)$, som er $\Theta\left(n^{\log_4 3}\right)$ når $T(1)$ er konstant. Nå kan vi finne total kostnad:

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right)$$

Vi bruker at $\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$, og får:

$$T(n) = \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta\left(n^{\log_4 3}\right)$$

Rekursjonstre krever ikke at vi er svært nøyaktige, så derfor kan vi også bruke en geometrisk serie $\sum_{k=0}^{n} x^k = \frac{1}{1-x}$, som en øvre grense:

$$T(n) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) = \frac{1}{1-3/16} cn^2 + \Theta\left(n^{\log_4 3}\right) = \frac{16}{13} cn^2 + \Theta\left(n^{\log_4 3}\right) = O(n^2)$$

Dermed har vi funnet at vi kan gjette: $T(n) = O(n^2)$. I dette tilfellet vil kostnaden til roten dominere den totale kostnaden til treet (siden kostnaden til roten er cn^2). Dersom $O(n^2)$ er en øvre grense for rekurrensen må det være en tett grense. Rekurrensen er $T(n) = 3T(n/4) + \Theta(n^2)$, så derfor vil det det første rekursive kallet bidrar med en kostnad på $\Theta(n^2)$, noe som betyr at $\Omega(n^2)$ må være en nedre grense for rekurrensen.

Vi bruker substitusjonsmetoden for å sjekke om vi har gjettet riktig, altså at $O(n^2)$ er øvre grense for $T(n) = 3T(n/4) + \Theta(n^2)$. Dette gjør vi ved å vise at $T(n) \leq dn^2$ for en konstant $d > 0$. Vi ser på $T(\lfloor n/4 \rfloor) \leq d\lfloor n/4 \rfloor^2$ og setter dette inn i $T(n)$:

$$\begin{aligned} T(n) &= 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leq 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &\leq \frac{3}{16} dn^2 + cn^2 \\ &\leq dn^2 \end{aligned}$$

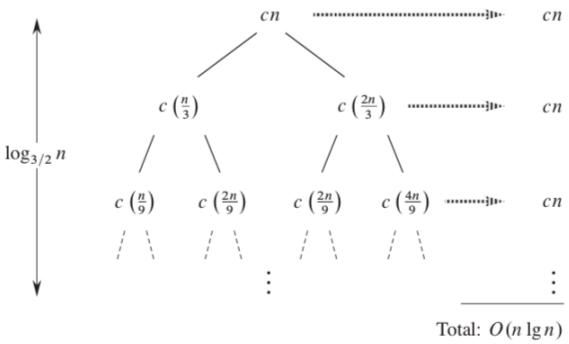
når $d \geq \frac{16}{13} c$. Videre må vi vise at denne løsningen også gjelder for grensebetingelsene. Dette gjør vi ved å vise at vi kan velge d slik at grensen $T(n) \leq dn^2$ også gjelder for liten n. Ved $n = 1$ vil $T(1) = 3T(\lfloor 1/4 \rfloor) + c1^2 = c \leq d$ dersom vi velger $d \geq c$. Dermed har vi vist at løsningen gjettet vha rekursjonstreet er riktig.

Eksempel 2 - rekursjonstre

Vi bruker et rekursjonstre for å gjette løsningen til rekurrensen:

$$T(n) = T(n/3) + T(2n/3) + O(n) = T(n/3) + T(2n/3) + cn$$

Utelukker tak og gulvfunksjoner for å forenkle. Figuren viser rekursjonstreet. Legg merke til at de første forgreningene har ulike størrelser. Ved det ene rekursive kallet vil størrelsen til delproblemene reduseres med en faktor på 3 for hver utvidelse, mens for det andre rekursive kallet vil størrelsen reduseres med en faktor på 2/3. Størrelsen til hhv. venstre og høyre node ved nivå i vil derfor være $n/3^i$ eller $n2^i/3^i$, slik at den totale størrelsen blir n. Siden kostnaden per node er $c *$ størrelse, vil total kostnad per nivå være cn (se figur).



For å finne $O\big(g(n)\big)$ må vi se på den lengste banen, siden dette er et usymmetrisk tre og den lengste banen vil ha lengst kjøretid (Merk: vi går til venstre for å finne $\Omega\big(g(n)\big)$. Den lengste banen vil være den til høyre, siden den opererer med høyeste verdier (tar lengre tid før den når bunnen der størrelsen til delproblemet er 1). Langs denne banen ved nivå i vil størrelsen være $n2^i/3^i$ og siden størrelsen til delproblemet ved bunnen er 1, vil dette nivået være $n(2/3)^i = 1 \;\Rightarrow\; i = \log_{3/2} n$. Dvs., høyden til treet er $\log_{3/2} n$. Total kostnad til rekursjonstreet er derfor $cn\log_{3/2} n$, altså: $T(n) = O\big(cn\log_{3/2} n\big) = O(n \lg n)$.

Et problem er at ikke alle nivåene vil ha kostnad cn. Hvis vi antar at rekursjonstreet er et fullstendig binært tre, dvs. hver node har to barnenoder helt til bunnen, vil antall noder ved nivå i være 2^i. Ved bladene (dvs. bunnivå) vil $i = \log_{3/2} n$, slik at antall noder blir $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$. Siden hver node har en konstant kostnad $T(1)$ vil total kostnad for dette nivået være $\Theta\big(n^{\log_{3/2} 2}\big)$. Siden $\log_{3/2} 2 > 1$ vil dette være større enn $\Theta(n) = cn$, og derfor vil en løs nedre grense for dette nivået være $\omega(n \lg n)$ og ikke cn! Samtidig vil ikke rekursjonstreet være et fullstendig binært tre, fordi noen blader være ved tidligere nivåer (noen baner når størrelse 1 raskere).

Siden rekursjonstre kun er ment å gi gode gjetninger til substitusjonsmetoden, gjør vi en forenkling og tipper at løsningen er $O(n \lg n)$. Vi vil vise at $T(n) \leq dn \lg n$ for en konstant $d > 0$. Vi ser på $T(\lfloor n/3 \rfloor) \leq d\lfloor n/3 \rfloor \lg\lfloor n/3 \rfloor$ og $T(\lfloor 2n/3 \rfloor) \leq d\lfloor 2n/3 \rfloor \lg\lfloor 2n/3 \rfloor$:

$$\begin{aligned}
T(n) &= T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + cn \\
&\leq T(n/3) + T(2n/3) + cn \\
&\leq (d(n/3)\lg(n/3)) \; + (d(2n/3)\lg(2n/3)) \; + cn \\
&= d(n/3)\lg(n) - d(n/3)\lg(3) \; + d(2n/3)\lg(n) + d(2n/3)\lg(2/3) + cn \\
&= dn\lg(n) - d\big((n/3)\lg(3) - (2n/3)\lg(2) + (2n/3)\lg(3)\big) + cn \\
&= dn\lg(n) - dn\big(\lg 3 - (2/3)\big) + cn \\
&\leq dn\lg(n)
\end{aligned}$$

når $-d\big(\lg 3 - (2/3)\big) + c \leq 0 \;\Rightarrow d \geq c/\big(\lg 3 - (2/3)\big)$. Dermed har vi vist at gjetningen fra rekursjonstreet var riktig.

4.5 Master teoremet

Master teoremet gir en oppskrift for å løse rekurrenser på formen:

$$T(n) = aT(n/b) + f(n)$$

der $a \geq 1$ og $b > 1$ er konstanter og $f(n)$ er en asymptotisk positiv funksjon. Master teoremet krever at vi husker tre tilfeller, men lar oss løse mange rekurrenser! Rekurrensen over beskriver kjøretiden til en algoritme som deler et problem av størrelse n inn i a delproblemer av størrelse n/b som hver løses rekursivt ila tiden $T(n/b)$. $f(n)$ er tiden som trengs for å dele opp problemet og kombinere delløsningene. Gulv eller takfunksjoner vil ikke påvirke den asymptotiske oppførselen til rekurrensen, og vil derfor ofte se bort fra disse.

Master teoremet

La $a \geq 1$ og $b > 1$ være konstanter, $f(n)$ er en asymptotisk positiv funksjon og $T(n)$ er definert som rekurrensen:

$$T(n) = aT(n/b) + f(n)$$

der n/b betyr enten $\lfloor n/b \rfloor$ eller $\lceil n/b \rceil$. Da vil $T(n)$ ha følgende grenser:

- Hvis $\boldsymbol{f(n) = O\left(n^{\log_b a - \epsilon}\right)}$ for en konstant $\epsilon > 0$, så vil $\boldsymbol{T(n) = \Theta\left(n^{\log_b a}\right)}$
- Hvis $\boldsymbol{f(n) = \Theta\left(n^{\log_b a}\right)}$, så vil $\boldsymbol{T(n) = \Theta\left(n^{\log_b a} \lg n\right) = \Theta(f(n) \lg n)}$
- Hvis $\boldsymbol{f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)}$ for en konstant $\epsilon > 0$ og $\boldsymbol{af(n/b) \leq cf(n)}$ for en konstant $c < 1$ og tilstrekkelig stor n, så vil $\boldsymbol{T(n) = \Theta\left(f(n)\right)}$

I de tre tilfellene sammenligner vi $f(n)$ med funksjonen $n^{\log_b a}$, og den største av disse to vil bestemme løsningen til rekurrensen. I tilfelle 1 er $n^{\log_b a}$ størst og vil derfor bestemme løsningen. I tilfelle 3 er $f(n)$ størst og vil derfor bestemme løsningen. I tilfelle 2 er de like store og da vil vi multiplisere med en logaritmisk faktor. Det er også noen mer spesifikke krav for å kunne bruke master teoremet:

Huskeregel for rekkefølge i delingen: **størst først**

- **Tilfelle 1: $\boldsymbol{f(n)}$ må være polynomial mindre enn $\boldsymbol{n^{\log_b a}}$. Dette sjekkes ved å se på $\boldsymbol{n^{\log_b a}/f(n)}$ og observere om dette er asymptotisk mindre eller lik $\boldsymbol{n^{\epsilon}}$.** Merk: kravet er at $f(n) \leq cn^{\log_b a}n^{-\epsilon}$. Dersom $f(n) = n$ og $n^{\log_b a} = n^2$ vil $n^{\log_b a}/f(n) = n$ som vil være lik n^{ϵ} for $\epsilon = 1$. Rekurrensen kan derfor løses med metode 1.

- **Tilfelle 3: $\boldsymbol{f(n)}$ må være polynomial større enn $\boldsymbol{n^{\log_b a}}$. Dette sjekkes ved å se på $\boldsymbol{f(n)/\ n^{\log_b a}}$ og observere om dette er asymptotisk større eller lik $\boldsymbol{n^{\epsilon}}$.** Dersom $f(n) = n \lg n$ og $n^{\log_b a} = n^{0.793}$ vil $f(n)/n^{\log_b a} \approx n^{(1-0.793)} \lg n$, som er større enn n^{ϵ}, for $\epsilon \approx 0.2$. Rekurrensen kan derfor løses med metode 3. Dersom $f(n) = n \lg n$ og $n^{\log_b a} = n$ vil $f(n)/n^{\log_b a} \approx n^{(1-1)} \lg n = \lg n$, som ikke er større enn n^{ϵ}, for $\epsilon > 0$ og derfor vil vi ikke kunne bruke master teoremet på denne rekurrensen.

Når du ser at $f(n)$ er mindre eller større enn $n^{\log_b a}$ vil neste steg derfor være å sjekke om denne forskjellen er polynomial vha metodene over. Dersom de er like kan vi bruke master teoremet med en gang.

Eksempler for bruk av master teoremet

Når vi får en rekurrens må vi bestemme om vi kan bruke master teoremet og i så fall hvilket tilfelle. Noen eksempler er:

$\boldsymbol{\underline{T(n) = 9T(n/3) + n}}$: vi har $a = 9, b = 3$ og $f(n) = n$. Dermed vil $n^{\log_b a} = n^{\log_3 9} = n^2$. Vi ser at $f(n) \leq n^{\log_b a}$, og må derfor sjekke om denne forskjellen er polynomial. Siden $n^{\log_b a}/f(n) = n$ som er lik n^{ϵ} for $\epsilon = 1$, vil forskjellen være polynomial: $f(n) = O\left(n^{\log_3 9 - \epsilon}\right)$. Tilfelle 1 gir: $T(n) = \Theta\left(n^{\log_3 9}\right) = \Theta(n^2)$.

$\boldsymbol{\underline{T(n) = T(2n/3) + 1}}$: vi har $a = 1, b = 3/2$ og $f(n) = 1$. Dermed vil $n^{\log_b a} = 1 = \Theta(1)$. Vi ser at $f(n) = n^{\log_b a}$, så derfor vil vi bruke tilfelle 2 som gir at $T(n) = \Theta(\lg n)$

$\boldsymbol{T(n) = 3T(n/4) + n \lg n}$: vi har $a = 3, b = 4$ og $f(n) = n \lg n$. Dermed vil $n^{\log_b a} = n^{\log_4 3} = n^{0.79248} = O(n^{0.793})$ (Bruker O fordi vi runder opp). Vi ser at $f(n) \geq n^{\log_b a}$, og må derfor sjekke om denne forskjellen er polynomial. Siden $f(n)/n^{\log_b a} = n^{(1-0.793)} \lg n$ som større enn n^{ϵ} for $\epsilon \approx 0.2$ vil forskjellen være polynomial: $f(n) = \Omega(n^{\log_4 3+\epsilon})$. Neste steg blir å sjekke den andre betingelsen. For tilstrekkelig stor n vil $af(n/b) = 3(n/4)lg\ (n/4) \leq (3/4)n \lg n \leq cf(n)$ for $c = 3/4$. Derfor kan vi bruke tilfelle 3, altså $T(n) = \Theta(n \lg n)$.

$\boldsymbol{T(n) = 2T(n/2) + n \lg n}$: vi har $a = 2, b = 2$ og $f(n) = n \lg n$. Dermed vil $n^{\log_b a} = n$. Vi ser at $f(n) \geq n^{\log_b a}$, og må derfor sjekke om denne forskjellen er polynomial. Siden $f(n)/n^{\log_b a} = \lg n$ som ikke er større eller lik n^{ϵ} for $\epsilon > 0$ vil ikke denne forskjellen være polynomial. **Dermed kan vi ikke bruke master teoremet for å finne $\boldsymbol{T(n)}$**

$\boldsymbol{T(n) = 2T(n/2) + \Theta(n)}$**:** vi har $a = 2, b = 2$ og $f(n) = \Theta(n)$. Dermed vil $n^{\log_b a} = n$. Vi ser at $f(n) = n^{\log_b a}$, så derfor vil vi bruke tilfelle 2 som gir at $T(n) = \Theta(n \lg n)$.

$\boldsymbol{T(n) = 8T(n/2) + \Theta(n^2)}$**:** vi har $a = 8, b = 2$ og $f(n) = \Theta(n^2)$. Dermed vil $n^{\log_b a} = n^3$. Vi ser at $f(n) \leq n^{\log_b a}$, og må derfor sjekke om denne forskjellen er polynomial. Siden $n^{\log_b a}/f(n) = n$ som er lik n^{ϵ} for $\epsilon = 1$ vil forskjellen være polynomial og tilfelle 1 gir $T(n) = \Theta(n^3)$.

$\boldsymbol{T(n) = 7T(n/2) + \Theta(n^2)}$**:** vi har $a = 7, b = 2$ og $f(n) = \Theta(n^2)$. Dermed vil $n^{\log_b a} = n^{2.807}$. Vi ser at $f(n) \leq n^{\log_b a}$, og må derfor sjekke om denne forskjellen er polynomial. Siden $n^{\log_b a}/f(n) = n^{2.807-2}$ som er lik n^{ϵ} for $\epsilon \approx 0.807$ vil forskjellen være polynomial og tilfelle 1 gir $T(n) = \Theta(n^{lg7})$.

Oppgave 4.5-3 (finn kjøretid til binær søk)

$\boldsymbol{T(n) = T(n/2) + \Theta(1)}$**:** vi har $a = 1, b = 2$ og $f(n) = \Theta(1)$. Dermed vil $n^{\log_b a} = n^0 = \Theta(1)$. Vi ser at $f(n) = n^{\log_b a}$, så derfor vil vi bruke tilfelle 2 som gir at $T(n) = \Theta(\lg n)$.

Kapittel 7 – Quicksort

Quicksort algoritmen har worst-case kjøretid på $\boldsymbol{\Theta(n^2)}$. Selv om quicksort har en dårlig worst-case kjøretid, blir den ofte brukt fordi den har kort a**verage kjøretid på $\boldsymbol{\theta(n \lg n)}$** med små konstante faktorer. En annen fordel er at den **sorterer *in place*.**

7.1 Beskrivelse av quicksort

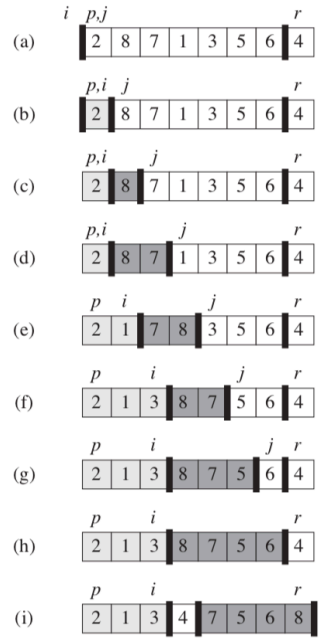
I likhet med Merge sort bruker Quicksort divide-and-conquer metoden:

- **Divide** – Arrayen $A[p \ldots r]$ deles inn i to subarrays $A[p \ldots q-1]$ og $A[q+1 \ldots r]$, på en slik måte at alle elementer i $A[p \ldots q-1]$ er mindre eller lik $A[q]$ og alle elementer i $A[q+1 \ldots r]$ er større eller lik $A[q]$. Utregningen av q er en del av denne oppdelingen.
- **Conquer** – sorter de to subarrayene vha rekursive kall til quicksort.
- **Combine** – siden de to subarrayene allerede er sortert i forhold til hverandre, krever det ingen ekstra arbeid å kombinere de. $A[p \ldots r]$ er nå sortert.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

PARTITION – deler opp *A* via sammenligning med pivotelement

Denne metoden vil omorganisere subarrayen $A[p \ldots r]$ innenfor den gitte plassen. Metoden begynner med å velge et pivotelement x som den setter lik $A[r]$, dvs. siste element i subarrayen. De andre elementene i subarrayen skal sammenlignes med x, og de vil plasseres i en av to områder: (1) de som er mindre enn x og (2) de som er større enn x. Vi bruker indeksen i for å plassere elementene i område 1. For-løkken vil gå fra starten av subarrayen til elementet foran pivotelementet. Dersom et element er mindre eller lik pivotelementet skal det bytte plass med elementet ved posisjon $i + 1$, som vil være posisjonen til høyre for skillet mellom de to områdene. For eksempel på figur e) og f) kan vi se at når $j = 5$ vil $A[j] = 3$ som er mindre enn pivotelementet $x = 4$. Dermed vil element med verdi 3 bytte plass med element med verdi 7, siden dette er første element til høyre for grensen mellom de to områdene. Merk: i vil også økes i a) når 2 blir byttet med seg selv. Til slutt vil pivotelementet bytte plass med elementet ved posisjon $i + 1$, slik at det blir plassert mellom de to områdene. Metoden vil returnere posisjonen til pivotelementet i $A[p..r]$.

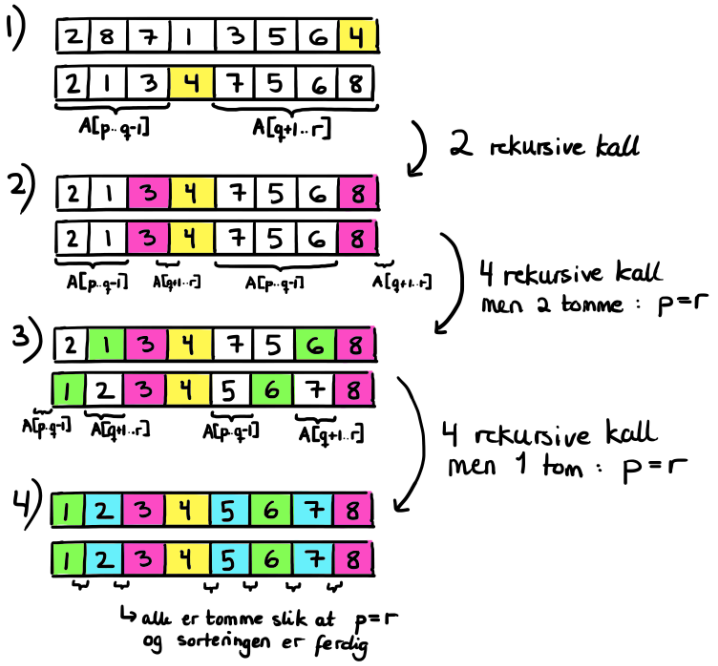



```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

QUICKSORT – sorterer en array

Denne metoden vil ta inn en array A som den vil sortere, samt start- og sluttpunkt. For å sortere hele arrayen må $p = 1$ og $r = A.length$. Dersom arrayen ikke er tom vil den kalle på PARTITION metoden som vil dele opp A i to subarrays, der den ene inneholder elementer mindre enn $A[q]$ og den andre inneholder elementer som er større enn $A[q]$. Deretter vil den algoritmen kjøre rekursive kall på hver av disse subarrayene ved å endre på start- og sluttverdiene. De rekursive kallene vil endre på rekkefølgen i A *in place*, slik at A blir stadig mer sortert. Til slutt vil vi nå tilfellet der subarrayene har lengde 1, og som derfor er ordnet. Da vil A være fullstendig sortert.

Figuren til høyre viser et eksempel. Ved standard quicksort vil x være det siste elementet i subarrayen. Etter partisjoneringen vil subarray til venstre ha elementer mindre enn x, mens subarray til høyre har elementer større enn x. Deretter kaller vi quicksort på disse subarrayene. I hver subarray velges det en ny x og elementene blir delt opp i to subarrays avhengig av om de er mindre eller større enn x. Quicksort blir deretter kalt på alle disse subarrayene. For hvert kall blir A sortert litt mer, og til slutt når alle subarrayene har lengde 1 vil den være fullstendig sortert.



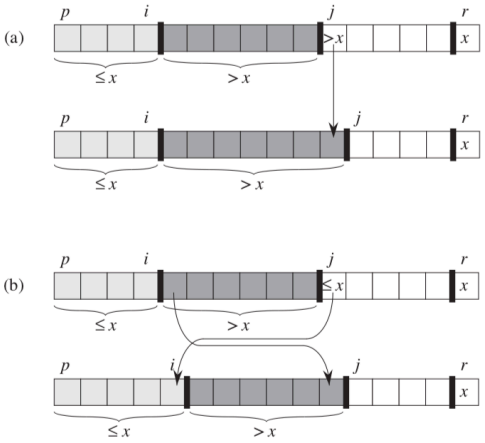
På figuren kan vi se at quicksort vil sortere *in place* siden den ikke behøver mer plass enn det arrayen A har.

Partisjoneringen kan beskrives med en loop invariant. Ved begynnelsen av hver iterasjon av loopen vil:

1. Hvis $p \leq k \leq i$, så vil $A[k] \leq x$
2. Hvis $i + 1 \leq k \leq j - 1$, så vil $A[k] > x$
3. Hvis $k = r$, så vil $A[k] = x$

Elementene mellom j og $r - 1$ er ikke inkludert i disse tre tilfellene, og har ingen spesifikt forhold til pivotelementet (de er enda ikke sammenlignet). For å vise at loop invarianten er sann, må vi beskrive de tre egenskapene (s. 4):

1. **Initialisering** (sant før første iterasjon) – før første iterasjon vil $i = p - 1$ og $j = p$. Dette betyr at det er ingen elementer mellom p og i eller mellom $i + 1$ og $j - 1$, slik at de første to kravene er oppfylt. Tildelingen i linje 1 gjør at krav 3 også er oppfylt.
2. **Vedlikehold** (sant før neste iterasjon) – vi har to tilfeller avhengig av utfallet i linje 4. Hvis $A[j] > x$ vil eneste handling være å øke j. Etter denne økningen vil krav 2 holde, og alle andre verdier er uendret (figur a). Hvis $A[j] \leq x$ skal i økes med én, vi skal bytte plass på $A[i]$ og $A[j]$ og til slutt øke j. Pga byttet vil $A[i] \leq x$ og dermed vil krav 1 være oppfylt. Vi vil også ha at $A[j - 1] > x$, slik at krav 2 er oppfylt.
3. **Terminering** (loopen terminerer) – ved terminering vil $j = r$. Derfor vil alle elementer i arrayen være i en av de tre områdene beskrevet av invarianten.



Kjøretiden til PARTITION på subarrayen $A[p \ldots r]$ er $\Theta(n)$ der $n = r - p + 1$.

7.2 Ytelsen til Quicksort

Kjøretiden til Quicksort avhenger av om partisjoneringen er balansert eller ubalansert, som igjen avhenger av hvilket element som brukes for partisjoneringen (dvs. som pivotelement):

- **Balansert partisjonering – asymptotisk like rask som merge sort**
- **Ubalansert partisjonering – asymptotisk like treg som innsettingssortering**

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

Worst-case partisjonering

Worst-case tilfellet er når partisjoneringen gir en subarray med $n - 1$ elementer og en med 0 elementer (pivotelementet er det $n$te elementet). Vi antar at denne ubalanserte partisjoneringen skjer ved hvert rekursivt kall. Kjøretiden til Quicksort algoritmen vil være summen av kjøretiden til partisjoneringen og de to rekursive kallene (se figur). Selve partisjoneringen tar tiden $\Theta(n)$. Dersom $T(n)$ er kjøretiden for å sortere en array med n elementer, vil kjøretiden for å sortere en subarray med 0 elementer være $T(0) = \Theta(1)$, mens kjøretiden for å sortere en subarray med $n - 1$ elementer er $T(n - 1)$ (merk: vi bruker rekurrenser siden dette er rekursive kall i en divide-and-conquer algoritme). Rekurrensen for blir:

$$T(n) = T(n - 1) + T(0) + \Theta(n) = T(n - 1) + \Theta(n)$$

Når du skal finne kjøretiden til et rekursivt kall, ta utgangspunkt i $T(n)$ der du erstatter n med størrelsen til inputen som gis det rekursive kallet. Eks: halve subarrayen sendes inn: $T(n/2)$ eller alle elementer uten ett sendes inn: $T(n - 1)$

Vi kan bruke iterativ metode (s.56) for å finne at løsningen på denne rekurrensen er $T(n) = n(n + 1)/2$ (se figur) når $T(1) = 1$. Altså vil $\boldsymbol{T(n) = \Theta(n^2)}$, noe som kan bevises vha substitusjonsmetoden (s. 58). Hvis partisjoneringen er ubalansert ved hvert rekursivt nivå vil kjøretiden være $\Theta(n^2)$, slik at **worst-case kjøretid for Quicksort er like dårlig som for innsettingssortering**. Dette vil også være tilfellet når input arrayen allerede er sortert, og da vil innsettingssortering ha kjøretid $O(n)$. Altså, vil innsettingssortering i dette tilfellet være mye bedre en Quicksort!

$$\begin{aligned}
T(n) &= n \\
&+ n - 1 && (1) \\
&+ n - 2 && (2) \\
&+ n - 3 && (3) \\
&\vdots && \vdots \\
&+ 1 && (n - 1)
\end{aligned}$$

$T(n-(n-1)) = T(1) = 1$

$$T(n) = n(n + 1)/2$$

Best-case partisjonering

Når partisjoneringen er så jevn som mulig vil en av de to subarrayene ha størrelse $\lfloor n/2 \rfloor$, mens den andre har størrelse $\lceil n/2 \rceil - 1$. I dette tilfellet vil quicksort kjøre mye raskere, fordi rekurrensen blir:

$$T(n) = 2T(n/2) + \Theta(n)$$

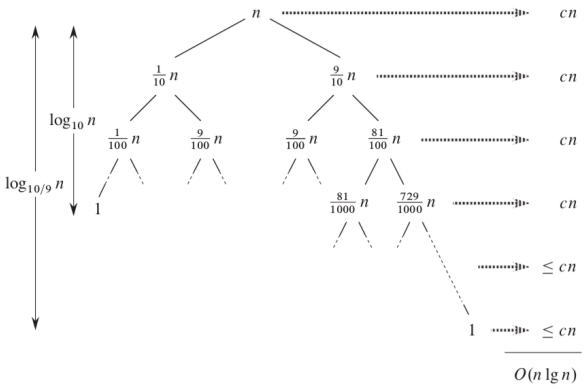
når vi tolererer litt unøyaktighet (dropper takfunksjon og -1). Vi har sett at denne rekurrensen har løsning $\boldsymbol{T(n) = \Theta(n \lg n)}$.

Average-case partisjonering

Average-case partisjonering er en balansert partisjonering som har kjøretid mye nærmere best-case enn worst-case (seksjon 7.4). For å forstå dette må vi forstå hvordan balansen i partisjoneringen er reflektert i rekurrensen som beskriver kjøretiden. For eksempel anta at partisjoneringen alltid produserer en 9-til-1 oppdeling som virker ganske ubalansert. Dette vil gi rekurrensen:

$$T(n) = T(9n/10) + T(n/10) + cn$$

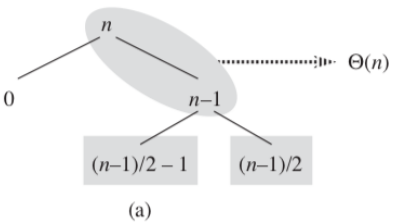
der cn er partisjoneringen ($\Theta(n)$). Figuren viser rekursjonstreet for denne rekurrensen. Legg merke til at hvert nivå har total kostnad cn helt til vi når dybde $\log_{10} n = \Theta(\lg n)$. Etter dette vil nivåene ha maks kostnad $cn = O(n)$. Rekurrensen vil terminere ved dybden $\log_{10/9} n = \Theta(\lg n)$, så derfor vil total kostnad være $cn\log_{10/9} n = O(n \lg n)$. Selv med en 9-til-1 partisjonering ved hvert nivå i rekursjonen vil Quicksort altså ha kjøretid $O(n \lg n)$, altså asymptotisk den samme som om partisjoneringen var på midten. Det samme gjelder for en 99-til-1 oppdeling. **Enhver oppdeling med konstant proporsjonalitet (dvs. konstant forhold) vil gi et rekursjonstre med dybde $\boldsymbol{\Theta(\lg n)}$ og nivåkostnad $\boldsymbol{O(n)}$, slik at kjøretiden blir $\boldsymbol{O(n \lg n)}$.**

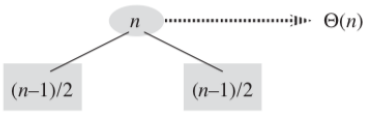


Intuisjon for average-case

Oppførselen til quicksort avhenger av den relative ordningen av verdier i input arrayen, og ikke av de spesifikke verdiene. Vi antar at alle permutasjoner av input verdiene er like sannsynlig. Ved en tilfeldig input vil det være svært lite sannsynlig at partisjoneringen skjer på samme måte ved hvert nivå, slik vi antok i den uformelle analysen. I stedet vil noen oppdelinger være balanserte, mens andre er ubalanserte. I average-vil partisjoneringen produsere en blanding av gode (dvs. balanserte, ca. 80%) og dårlige (ca. 20%) oppdelinger, og disse vil være tilfeldig fordelt i et rekursjonstre. Vi kan se for oss at de gode og dårlige nivåene alternerer i treet og at de gode er best-case, mens de dårlige er worst-case partisjoneringer. Figur a viser oppdelingen ved to påfølgende nivåer:

1. **Worst-case:** ved roten vil en array av n elementer deles opp i to subarrays med størrelse $n - 1$ og 0. Partisjoneringskostnaden blir $\Theta(n)$
2. **Best-case:** ved neste nivå vil subarrayen med størrelse $n - 1$ deles opp i to subarrays med størrelse $(n-1)/2 - 1$ og $(n-1)/2$. Partisjoneringskostnaden blir $\Theta(n-1)$





(b)

Vi antar at grensebetingelseskostnaden er 1 for subarrayen med størrelse 0. Kombinasjonen av dårlig og god oppdeling gir tre subarrays med størrelse 0, $(n-1)/2-1$ og $(n-1)/2$, og den kombinerte kostnaden er $\Theta(n)+\Theta(n-1)=\Theta(n)$. Altså, vil kostnaden være den samme som når et enkelt nivå blir oppdelt i to subarrays med størrelse $(n-1)/2$ (figur b), slik at kostnaden er $\Theta(n)$. Intuitivt vil kostnaden til den dårlige oppdelingen ($\Theta(n)$) kunne absorberes inn i kostnaden til den gode oppdelingen ($\Theta(n-1)$), og den resulterende oppdelingen er god. **Average-case kjøretid for quicksort vil derfor være like kjøretiden for kun gode partisjoneringer: $O(n \lg n)$, men med en litt større konstant.**

7.3 En randomisert versjon av quicksort

Alle permutasjonene av input verdiene vil ikke alltid være like sannsynlig. Noen ganger kan vi legge randomisering til i algoritmen for å få en god forventet ytelse ved alle input. Randomisert quicksort blir mye brukt for sortering av tilstrekkelig store input.

Randomisert quicksort bygger på **random sampling**, som går ut på at **pivotelementet blir tilfeldig valgt fra subarrayen $A[p \ldots r]$ istedenfor å alltid være $A[r]$.** Random samling gjør at det er like sannsynlig at pivotelementet er ethvert av elementene i subarrayen. **Siden pivot elementet blir tilfeldig valgt, kan vi forvente at oppdelingen blir tilnærmet balansert i average-case**. Ellers er algoritmen uendret.

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

RANDOMIZED-PARTITION – tilfeldig oppdeling
Denne metoden vil gi en randomisert partisjonering ved å bytte ut pivotelementet $x = A[r]$ med $x = A[i]$ der i er et tilfeldig element i subarrayen $A[p \ldots r]$. Deretter kaller den på den vanlige PARTITION metoden.

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

RANDOMIZED-QUICKSORT – tilfeldig quicksort
Denne metoden vil være en randomisert quicksort ved at den kaller på RANDOMIZED-PARTITION istedenfor PARTITION. Ellers er den uendret.

7.4 Analyse av quicksort

Skal analysere quicksort mer nøyaktig.

7.4.1 Worst-case analyse – quicksort og randomisert quicksort

Vi har sett at worst-case partisjonering ved hvert nivå av rekursjonen i quicksort vil føre til en worst-case kjøretid på $\Theta(n^2)$. Vi kan bruke substitusjonsmetoden for å vise at kjøretiden til quicksort er $O(n^2)$. Dersom $T(n)$ er worst-case kjøretid for quicksort ved inputstørrelse n, vil vi ha rekurrensen:

$$T(n) = \max_{0 \le q \le n-1} \big(T(q) + T(n-q-1)\big) + \Theta(n)$$

der max(…) representerer worst-case partisjonering. Quicksort har to rekursive kall på delproblem av størrelse q og $n-q-1$ (-1 siden delproblemet vil være fra $q+1$ til n og størrelsen blir derfor 1 mindre). q vil ligge mellom 0 og $n-1$ fordi partisjonen produserer to delproblem med total størrelse $n-1$. $\Theta(n)$ representerer tiden som

trengs for partisjoneringen (divide-steget). Vi gjetter at $T(n) \leq cn^2$, som innsatt i rekurrensen gir:

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n-q-1)^2) + \Theta(n) \end{aligned}$$

Uttrykket $(q^2 + (n-q-1)^2)$ vil nå et maksimum ved endepunktene, altså $q = 0$ og $q = n-1$ (kan sjekkes ved å se på den deriverte lik 0). Dersom vi setter dette inn i uttrykket får vi at $(q^2 + (n-q-1)^2) \leq (n-1)^2 = n^2 - 2n + 1$. Dermed vil:

$$\begin{aligned} T(n) &\leq c \cdot (n^2 - 2n + 1) + \Theta(n) \\ &= cn^2 - c(2n-1) + \Theta(n) \\ &\leq cn^2 \end{aligned}$$

Siden vi kan velge c slik at $c(2n-1)$ utligner $\Theta(n)$. Altså vil $T(n) = O(n^2)$. Vi har sett at et spesielt tilfelle vil gi at kjøretiden til quicksort også er $\Omega(n^2)$ når partisjoneringen er ubalansert. **Derfor vil worst-case kjøretid for quicksort være $\boldsymbol{\Theta(n^2)}$.**

7.4.2 Forventet kjøretid for randomisert quicksort

Den forventede kjøretiden til randomisert quicksort er $O(n \lg n)$, fordi kostnaden for de dårlige partisjoneringene blir absorbert i de gode. Dette kombinert med best-case kjøretid på $\Theta(n \lg n)$, gir at **forventet kjøretid for randomisert quicksort er $\boldsymbol{\Theta(n \lg n)}$**. Da har vi antatt at det er distinkte verdier som blir sortert. Vi skal nå bevise dette.

Kjøretid og sammenligninger

Den eneste forskjellen mellom quicksort og randomisert quicksort er hvordan de velger pivotelement. Vi kan derfor analysere randomisert quicksort vha standard quicksort, men med antagelsen om at pivotelementene blir tilfeldig valgt fra subarrayen. Kjøretiden til PARTITION metoden vil dominere kjøretiden til QUICKSORT. Ved hvert kall til PARTITION metoden vil det velges et pivotelement som ikke blir en del av noen fremtidige rekursive kall til QUICKSORT og PARTITION, siden pivotelementet ikke blir med som en del av de to subarrayene. Derfor kan det være maksimum n kall til PARTITION og hvert kall tar $O(1)$ tid pluss en mengde tid som er proporsjonal med antall iterasjoner av for-loopen i linje 3-6 (se figur). Hver iterasjon vil sammenligne pivotelementet med et annet element i subarrayen og evt. flytte på elementet. Vi kan binde tiden som brukes i for-loopen dersom vi finner antall ganger if-setningen ved linje 4 utføres.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Vi lar X være antall sammenligninger som utføres i linje 4 av PARTITION metoden totalt for hele quicksort utføringen. Da vil kjøretiden til QUICKSORT være $O(n + X)$. Dette fordi algoritmen vil ha maksimum n kall på PARTITION metoden og hver av disse vil utføre en konstant mengde arbeid og en for-loop. For å analysere X må vi forstå når algoritmen sammenligner to elementer og når den ikke gjør det. Vi kaller elementene i A for $z_1, z_2, \ldots, z_n$, der z_i er det i'ende minste elementet. Vi definerer $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ til å være settet av elementer mellom z_i og z_j. Vi lar X_{ij} være situasjonen der z_i blir sammenlignet med z_j. Alle elementer blir maksimalt sammenlignet en gang, siden ett av elementene må være pivotelementet og det blir ikke inkludert i subarrayene som gis til det rekursive kallet. Derfor vil totalt antall sammenligninger være:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Denne kan vi løse vha indikator random variabel som er beskrevet i kapittel 5, som ikke er pensum. Det er derfor ikke nødvendig at du forstå alt ved resten av beviset.
Vi definerer X_{ij} vha en indikator random variabel:

$$X_{ij} = I\{z_i \text{ sammenlignes med } z_j\} = \begin{cases} 1, & \text{hvis } z_i \text{ sammenlignes med } z_j \\ 0, & \text{ellers} \end{cases}$$

Dette gjør at $E[X_{ij}] = \Pr\{z_i \; sammenlignes \; med \; z_j\}$. Altså forventningsverdien til X_{ij} er lik sannsynligheten for at $z_i \; sammenlignes \; med \; z_j$. Dermed kan vi finne forventningsverdien til totalt antall sammenligninger:

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \; sammenlignes \; med \; z_j\}$$

Derfor må vi finne $\Pr\{z_i \; sammenlignes \; med \; z_j\}$. Når elementene deles inn i to subarrays basert på et pivotelement, vil elementene som er i den ene subarrayen aldri sammenlignes med elementene som er i den andre. Disse vil sammenlignes med pivotelementet og blir deretter sendt inn i separate rekursive kall. Pivotelementet vil sammenlignes med alle elementer. Derfor vil z_i og z_j kun sammenlignes dersom det første elementet som velges som pivotelement fra Z_{ij} er enten z_i eller z_j. Dette betyr at $\Pr\{z_i \; sammenlignes \; med \; z_j\}$ vil være lik sannsynligheten for at dette skjer. Det er like sannsynlig at ethvert element fra Z_{ij} blir valgt som pivot og Z_{ij} inneholder $j - i + 1$ elementer. Derfor vil:

$$\begin{aligned} \Pr\{z_i \; sammenlignes \; med \; z_j\} &= \Pr\{z_i \text{ blir første pivot}\} + \Pr\{z_j \text{ blir første pivot}\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &= \frac{2}{j-i+1} \end{aligned}$$

siden de to hendelsene er gjensidig utelukkende. Vi setter dette inn i ligningen til $E[X]$:

$$E[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Dersom vi bruker $k = j - i$ og at $\sum_{k=1}^{n} 1/k = \ln n + O(1)$, får vi at:

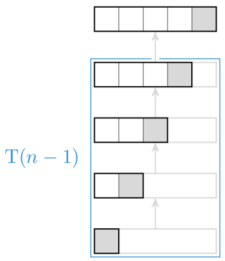
$$E[X] = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n)$$

Dermed kan vi konkludere med at dersom vi bruker RANDOMIZED-PARTITION, vil **forventet kjøretid for quicksort være $\boldsymbol{O(n + n \lg n) = O(n \lg n)}$ når elementverdiene er forskjellige.**

Appendiks B – Iterasjonsmetoden

Vi har sett at rekursjonstrær kan brukes for å løse rekurrensligninger. Dette er en mer visuell fremgangsmåte av en annen metode kalt **iterasjonsmetoden**. Disse bruker samme prinsipp, men iterasjonsmetoden jobber mer direkte på ligningene. Ideen er at man bruker rekurrensen selv til å ekspandere de rekursive terminene. Det er to fremgangsmåter, som vi illustrerer vha rekurrensrelasjonen:

$$T(0) = 0$$
$$T(n) = T(n-1) + 1$$



Her vil vi bli kvitt $T(n-1)$ på høyre side av ligningen. Hvis vi antar at $n > 1$, kan vi bruke definisjonen $T(n) = T(n-1) + 1$ til å finne ut hva $T(n-1)$ er. Definisjonen gir at $T(n-1) = T(n-2) + 1$ og dermed kan vi sette dette inn i $T(n)$, for å få at $T(n) = T(n-2) + 2$. Dette er en ekspandering av rekurrensen, og dersom vi fortsetter slik vil vi etter hvert muligens se et mønster.

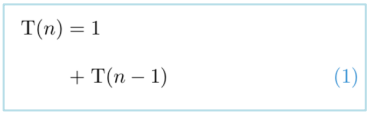
Fremgangsmåte 1: $T(n-1) + 1$

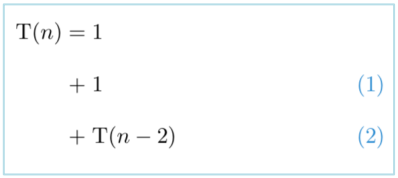
Vi regner ut hvert nivå og forsøker å se ett mønster (obs: viktig å være nøye på å ta med alle leddene fra det rekursive kallet):

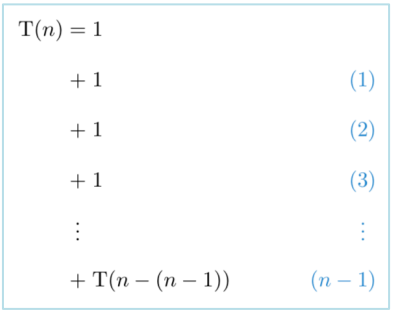
$$\begin{aligned}\text{Grunntilfellet: } T(0) &= 0\\ T(n) &= T(n-1) + 1\\ &= T(n-2) + 2\\ &= T(n-3) + 3\\ &\dots\\ &= T(n-i) + i\\ &= T(n-n) + n = n\end{aligned}$$

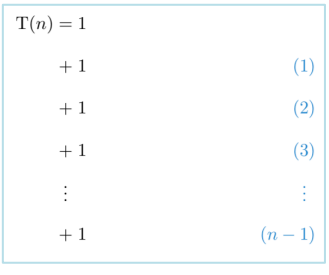
Dermed har vi funnet at $T(n) = n$ og høyden på rekurrensen er $i = n$. Først ekspanderer vi altså $T(n)$, deretter $T(n-1)$, $T(n-2)$, osv. hele tiden ved hjelp av den opprinnelige rekurrensen. Etter hvert klarer vi å uttrykke resultatet etter et vilkårlig antall ekspansjoner, i. Her kan i være 1, 2, 3, eller et hvilket som helst annet tall, så lenge vi ikke ekspanderer oss "forbi" grunntilfellet $T(0)$. Det er nettopp dette grunntilfellet vi ønsker å nå, altså vi vil gi 0 som argument til T, slik at vi kan bruke at $T(0) = 0$. I vårt eksempel vil dette være ved $i = n$, fordi da vil $n - n = 0$. Dette gir oss svaret på siste linje, nemlig at $T(n) = n$.

Fremgangsmåte 2: $T(n-1) + 1$









Her kan vi se at vi utvider rekurrensen ved å legge til et nytt nivå for hvert rekursivt kall og vi slår ikke sammen verdiene. Tallene til høyre representerer høyden til rekursjonstreet og vil være den verdien av i som gir grunntilfellet $T(1)$ (eller $T(0)$) ved bunnen. Her kan vi se at $T(n) = (n-1) * 1 + 1 = n$, fordi vi har $(n-1)$ antall 1'ere pluss 1 ved toppen.

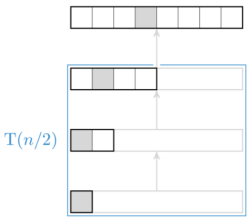
Merk: vi finner $T(n)$ ved å multiplisere antall nivåer ($i = n - 1$) med verdien ved hvert nivå (1) og legger til den øverste verdien (1).

Grunntilfellet $T(1) = 1$

På forrige side løste vi rekurrensrelasjonen $T(n) = T(n-1) + 1$ med $T(0) = 0$. Dette grunntilfellet gir $T(1) = 1$, som vi også kan bruke som grunntilfelle. **Som regel vil vi anta at $T(1) = 1$, altså at det tar konstant tid $O(1)$ å utføre algoritmen når problemet har størrelse 1.** Vi skal se at vi også kan løse rekurrensen med denne:

$$
\begin{aligned}
\text{Grunntilfellet: } T(1) &= 1 \\
T(n) &= T(n-1) + 1 \\
&= T(n-2) + 2 \\
&= T(n-3) + 3 \\
&\dots \\
&= T(n-i) + i \\
&= T\big(n-(n-1)\big) + (n-1) \\
&= 1 + (n-1) \\
&= n
\end{aligned}
$$

Her vil høyden på rekurrensen bli $i = n - 1$, altså en mindre enn når grunntilfellet er $T(0) = 0$.



Eksempel 2: Mergesort

Vi kan finne kjøretiden til mergesort vha denne metoden. Det er lettest å se mønsteret vha fremgangsmåte 2. Vi vet at mergesort har rekurrensrelasjonen:

$$
\begin{aligned}
\text{Grunntilfellet: } T(1) &= 1 \\
T(n) &= 2T(n/2) + n
\end{aligned}
$$

Øverste verdi vil være n. Verdien ved hvert nivå vil også være n pga faktoren foran det rekursive kallet. For eksempel vil $2T(n/2) = 2(2T(n/4) + n/2) = 4T(n/4) + n$:

$$
\begin{aligned}
&\mathrm{T}(n) = n \\
&\quad + 2\mathrm{T}(n/2) && (1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{T}(n) = n \\
&\quad + n \quad (= 2 \cdot \tfrac{n}{2}) && (1) \\
&\quad + 4\mathrm{T}(n/4) && (2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{T}(n) = n \\
&\quad + n && (1) \\
&\quad + n \quad (= 4 \cdot \tfrac{n}{4}) && (2) \\
&\quad + 8\mathrm{T}(n/8) && (3)
\end{aligned}
$$

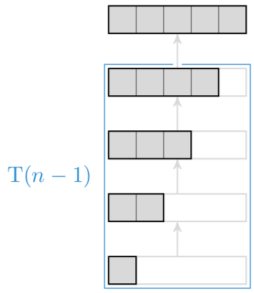
Vi kan se et mønster at neste ledd er gitt av $2^i T(n/2^i)$. For å få $T(n/n) = T(1)$, må $i = \lg n$. Dermed får vi:

$$
\begin{aligned}
&\mathrm{T}(n) = n \\
&\quad + n && (1) \\
&\quad + n && (2) \\
&\quad + n && (3) \\
&\quad \vdots && \vdots \\
&\quad + 2^{\lg n} \cdot \mathrm{T}(n/2^{\lg n}) && (\lg n)
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{T}(n) = n \\
&\quad + n && (1) \\
&\quad + n && (2) \\
&\quad + n && (3) \\
&\quad \vdots && \vdots \\
&\quad + n && (\lg n)
\end{aligned}
$$

Her har vi brukt at $T(1) = 1$. Som vi kan se på figuren vil $T(n) = n + n \lg n$, siden det er $\lg n$ antall nivåer med n kostnad og et initialt nivå med n kostand.

Dermed har vi vist at kjøretiden til mergesort er $\Theta(n + n \lg n) = \Theta(n \lg n)$.
Dette kan verifiseres med substitusjonsmetoden.



Eksempel 3 – Quicksort

Vi kan også bruke denne metoden for å finne worst-case kjøretid til Quicksort, som har rekurrensrelasjon:

$$\text{Grunntilfellet: } T(1) = 1$$
$$T(n) = T(n-1) + n$$

Det er lettest å se mønsteret vha fremgangsmåte 2:

$$\begin{aligned} \mathrm{T}(n) &= n \\ &+ \mathrm{T}(n-1) && (1) \end{aligned}$$

$$\begin{aligned} \mathrm{T}(n) &= n \\ &+ n-1 && (1) \\ &+ \mathrm{T}(n-2) && (2) \end{aligned}$$

$$\begin{aligned} \mathrm{T}(n) &= n \\ &+ n-1 && (1) \\ &+ n-2 && (2) \\ &+ \mathrm{T}(n-3) && (3) \end{aligned}$$

Vi kan se et mønster at neste ledd er gitt av $T(n-i)$. For å få $T(1)$, må $i = n-1$. Dermed får vi:

$$\begin{aligned} \mathrm{T}(n) &= n \\ &+ n-1 && (1) \\ &+ n-2 && (2) \\ &+ n-3 && (3) \\ &\vdots && \vdots \\ &+ \mathrm{T}(n-(n-1)) && (n-1) \end{aligned}$$

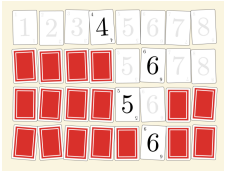
$$\begin{aligned} \mathrm{T}(n) &= n \\ &+ n-1 && (1) \\ &+ n-2 && (2) \\ &+ n-3 && (3) \\ &\vdots && \vdots \\ &+ 1 && (n-1) \end{aligned}$$

Her har vi brukt at $T(1) = 1$. Vi kjenner igjen dette som $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ fra side 31. Derfor vil løsningen være $T(n) = n(n+1)/2$.

Dermed har vi vist at worst-case kjøretid for quicksort er $\boldsymbol{\Theta(n(n+1)/2) = \Theta(n^2)}$. Dette kan verifiseres med substitusjonsmetoden.

Appendiks C – Binærsøk

Vi skal nå se på algoritmen til binærsøk, som ikke er dekket i boka. Arrayen som gis i input antas å være sortert.




```
BISECT(A, p, r, v)
1  if p ≤ r
2      q = ⌊(p + r)/2⌋
3      if v == A[q]
4          return q
5      elseif v < A[q]
6          return BISECT(A, p, q − 1, v)
7      else return BISECT(A, q + 1, r, v)
8  else return NIL
```

BISECT – rekursivt binært søk

Denne metoden vil ta inn en sortert array, start og sluttpunkt og en verdi som det skal søkes etter. Dersom starten er mindre eller lik slutten vil q settes lik halvparten av summen av disse. Dersom verdien ved denne indeksen er lik v, skal indeksen returneres. Dersom verdien er større enn v skal metoden kalles rekursivt, men nå med sluttpunkt lik $q-1$, altså halveres mengden elementer. Dette fungerer fordi subarrayen er sortert. Dersom verdien ved indeks q er større enn v vet vi at

elementene ved høyere indeks enn q også vil være større enn v. Derfor ser vi kun på elementene fram til elementet før q (dvs. fra start til $q-1$). Det samme gjelder dersom verdien ved indeks q er mindre enn v. Da vil vi sette startpunktet lik $q+1$, siden vi vet at alle elementene mindre enn q også vil være mindre enn v. Dermed blir metoden gjentatt helt til vi finner en verdi som er lik v eller subarrayen får lengde 0, slik at if-setningen ved linje 1 ikke utføres.

Dersom A inneholder v, vil BISECT*(A, 1, n, v)* returnere en indeks i, slik at $A[i] = v$. Hvis ikke returneres NIL. For å vise at pseudokoden er riktig kan vi bruke induksjon over lengden til segmentet $A[p \dots r]$. Vi har to grunntilfeller:

- $p > r$, altså segmentet er tomt og metoden returnerer NIL pga første if-setning
- $p = r$ og $v = A[q]$, som vil returnere q.

Begge disse er korrekte. Neste steg er å vise at neste iterasjon blir riktig dersom nåværende iterasjon er riktig. Vi antar at $p \leq r$ og at BISECT gir rett svar for kortere intervaller. Siden både $A[p \dots q-1]$ og $A[q+1 \dots r]$ er kortere intervaller, så vil rett indeks returneres uansett hvilket rekursivt kall som velges. Dermed har vi vist at pseudokoden er riktig!

For å finne kjøretiden kan vi for eksempel telle antall ganger sammenligningen $p \leq r$ utføres. Vi gjør noen antagelser for å forenkle utregningen som har **ingenting å si for den asymptotiske kjøretiden**. Vi antar at $n = 2^k$ for et heltall $k \geq 0$, slik at vi alltid kan dele nøyaktig på midten, helt til vi ender med $p == q$. Vi antar også at v finnes i A, slik at rekursjonen stanser der. For å sikre at vi deler på midten, kan vi anta at den eneste forekomsten av v er $A[n]$, slik at vi alltid velger det høyre segmentet, $A[q+1 \dots r]$ og finner elementet ved bunnen. Vi får da følgende rekurrensutregning:

$$
\begin{aligned}
\text{Gruntilfellet: } T(1) &= 1 \\
T(n) &= T(n/2) + 1 \\
&= T(n/4) + 2 \\
&= T(n/8) + 3 \\
&\dots \\
&= T(n/2^i) + i \\
&= T(n/n) + \lg n = \lg n + 1
\end{aligned}
$$

Siden $T(1) = 1$. **Altså vil kjøretiden til binærsøk algoritmen være $\boldsymbol{\Theta(\lg n + 1) = \Theta(\lg n)}$** .

Det er også mulig å lage en binær søk algoritme vha iterasjoner:

```
BISECT'(A, p, r, v)
1  while p ≤ r
2      q = ⌊(p + r)/2⌋
3      if v == A[q]
4          return q
5      elseif v < A[q]
6          r = q − 1
7      else p = q + 1
8  return NIL
```

<u>BISECT' – iterativt binært søk</u>
Denne metoden vil ta inn en sortert array, start og sluttpunkt og en verdi som det skal søkes etter. Dersom starten er mindre eller lik slutten vil q settes lik halvparten av disse. Hvis verdien ved q er lik v, skal q returneres. Hvis ikke skal start- eller sluttpunktet oppdateres. Denne prosessen gjentas helt til man finner v eller til starten blir større enn slutten slik at NIL returneres.

Denne versjonen vil generelt være mer effektiv, siden man slipper ekstra kostnader ved funksjonskall. Vi kan også her bruke induksjon for å vise at pseudokoden er riktig og iterasjonsmetoden for å finne kjøretiden.

Substitusjonsmetoden – forelesning

Når vi har kommet frem til en løsning på rekurrensen kan vi verifisere den med **induksjon**, som også kalles **substitusjonsmetoden**. Denne metoden har to steg:

1. **Grunntilfellet: Vis at grunntilfellet stemmer**
2. **Induktivt steg: Anta at løsningen gjelder for rekursiv kall og vis at det gir løsningen**

Eksempel 1: $T(n) = n$

Vi har rekurrensrelasjonen:

$$T(0) = 0$$
$$T(n) = T(n-1) + 1$$

Som har løsning $T(n) = n$. Vi kan verifisere dette vha substitusjonsmetoden:

1. $T(1) = 1 = T(0) + 1 = 1$
2. Vi antar $T(n-1) = n - 1$ og vil vise at $T(n) = n$. Vi får:

$$T(n) = (n-1) + 1 = n$$

Siden begge stegene oppfylles har vi verifisert løsningen $T(n) = n$.

Eksempel 2: binærsøk

Et annet eksempel er rekurrensrelasjonen til binærsøk algoritmen:

$$T(1) = 1$$
$$T(n) = T(n/2) + 1$$

Som har løsning $T(n) = \lg n + 1$. Vi kan verifisere dette vha substitusjonsmetoden:

1. $T(1) = T(0) + 1 = 1$ (antatt: $T(0) = 0$, og $T(\lfloor n/2 \rfloor)$)
2. Vi antar $T(n/2) = \lg(n/2) + 1$ og vil vise at $T(n) = \lg n + 1$. Vi får:

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= \lg\left(\frac{n}{2}\right) + 1 + 1 \\ &= \lg n - \lg 2 + 2 \\ &= \lg n - 1 + 2 \\ &= \lg n + 1 \end{aligned}$$

Eksempel 3: mergesort

Et annet eksempel er rekurrensrelasjonen til mergesort algoritmen:

$$T(1) = 1$$
$$T(n) = 2T(n/2) + n$$

Som har løsning $T(n) = n \lg n + n$. Vi kan verifisere dette vha substitusjonsmetoden:

1. $T(1) = 2T(0) + 1 = 1$ (antatt: $T(0) = 0$, og $T(\lfloor n/2 \rfloor)$)
2. Vi antar $T(n/2) = (n/2)\lg(n/2) + (n/2)$ og vil vise at $T(n) = n \lg n + n$. Vi får:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\big((n/2)\lg(n/2) + (n/2)\big) + n \end{aligned}$$

$$
\begin{aligned}
&= n\lg(n) - n\lg 2 + n + n \\
&= n\lg(n) - n + n + n \\
&= n\lg n + n
\end{aligned}
$$

Eksempel 4: quicksort (worst-case)

Et annet eksempel er rekurrensrelasjonen til quicksort algoritmen:

$$
\begin{aligned}
\text{Grunntilfellet: } & T(1) = 1 \\
& T(n) = T(n-1) + n
\end{aligned}
$$

Som har løsning $T(n) = n(n+1)/2$. Vi kan verifisere dette vha substitusjonsmetoden:

1. $T(1) = 1 = 1(1+1)/2 = 1$
2. Vi antar $T(n-1) = (n-1)n/2$ og vil vise at $T(n) = n(n+1)/2$. Vi får:

$$
\begin{aligned}
T(n) &= T(n-1) + n \\
&= \frac{(n-1)n}{2} + n \\
&= \frac{n(n-1+2)}{2} \\
&= \frac{n(n+1)}{2}
\end{aligned}
$$

Forelesning 4 – Rangering i lineær tid

Vi kan ofte få bedre løsninger ved å styrke kravene til input eller ved å svekke kravene til output. Sortering basert på sammenligninger ($x \leq y$) er et klassisk eksempel: I verste tilfellet må vi bruke $\lg n!$ sammenligninger, men om vi antar mer om elementene eller bare sorterer noen av dem så kan vi gjøre det bedre. Læringsmålene er:

- **Forstå hvorfor sammenligningsbasert sortering har en worst-case på $\Omega(n \lg n)$**
- Vite hva en stabil sorteringsalgoritme er
- Forstå COUNTING-SORT, og hvorfor den er stabil
- **Forstå RADIX-SORT, og hvorfor den trenger en stabil subrutine**
- Forstå BUCKET-SORT
- Forstå RANDOMIZED-SELECT
- Kjenne til SELECT (ikke grundig forståelse av virkemåte)

Kapittel 8 – Sortering i lineær tid

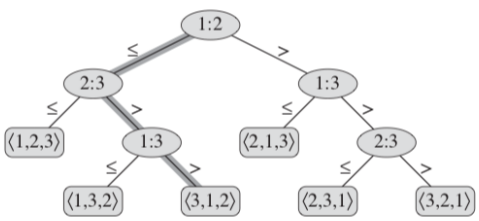
Vi har sett på flere algoritmer som kan sortere n nummer i løpet av $O(n \lg n)$ tid. Merge sort og heapsort (mer senere) oppnår dette i worst-case, mens quicksort oppnår det i average-case. Vi kan gi en input med størrelse n som får algoritmen til å kjøre på $\Omega(n \lg n)$ tid. Hos disse algoritmene vil den sorterte rekkefølgen baseres på sammenligninger av input elementene, noe som kalles **sammenligningssortering**. Vi skal se at **sammenligningssorteringer må utføre $\Omega(n \lg n)$ sorteringer i worst-case**, slik at merge sort og heapsort er asymptotiske optimale og ingen sammenlignings-sortering er raskere av mer enn en konstant faktor. Det finnes likevel andre typer sorteringer som kan kjøre på lineær tid, som vi nå skal se.

8.1 Nedre grense for sortering

I sammenligningssortering er det kun sammenligninger av elementer som gir informasjon om rekkefølgen til en input sekvens $\langle a_1, a_2, \ldots a_n \rangle$. For å bestemme rekkefølgen til to elementer a_i og a_j må vi utføre testene $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$ eller $a_i > a_j$ for å bestemme deres relative rekkefølge. Vi antar at alle input elementene er distinkte, slik at vi kan anta at alle sammenligninger kan være på formen $a_i \leq a_j$.

Valgtre modellen

Et valgtre er et fullstendig binært tre (dvs. to barnenoder) som representerer sammenligningene mellom elementer som blir utført av en sorteringsalgoritme som operer på en input. Figuren viser valgtreet som korresponderer til innsettingssortering av en inputsekvens med tre elementer $\langle 1, 2, 3 \rangle$. Utføringen av algoritmen korresponderer til å følge en rute fra roten til bladet. Hvert blad representerer en permutasjon, for eksempel $\langle 3, 2, 1 \rangle$ eller $\langle 2, 1, 3 \rangle$. Hver indre node får betegnelsen $i{:}\,j$ og representerer en sammenligning $a_i \leq a_j$. Venstre rute følges dersom $a_i \leq a_j$, mens høyre rute følges dersom $a_i > a_j$. Når vi kommer til et blad vil sorteringsalgoritmen ha bestemt rekkefølgen.



Enhver sorteringsalgoritme må kunne produsere alle permutasjoner av inputen, siden inputen kan ha enhver rekkefølge på elementverdiene, slik at sortert rekkefølge blir a_1, a_2, a_3 eller a_2, a_1, a_3, osv. Derfor må valgtreet ha $n!$ permutasjoner som hver representeres av et blad. Hver av disse bladene må også kunne nås fra roten via en nedovergående rute.

$\Omega(n \lg n)$ – en nedre grense for worst-case

Worst-case antall sammenligninger vil være den lengste ruten fra roten til bladet i valgtreet, og dette gis av høyden til valgtreet. En nedre grensen for høyden til valgtreet er derfor en nedre grense for kjøretiden til sammenligningssorteringsalgoritmen.

> **Teorem 8.1**
> **Enhver sammenligningssortering krever $\Omega(n \lg n)$ sammenligninger i worst-case tilfellet**

For å bevise dette må vi vise at høyden til valgtreet er $n \lg n$. Vi ser på et valgtre med høyde h og l blad som kan nås fra roten. Siden det er $n!$ permutasjoner som skal representeres av blad og et binært tre av høyde h ikke kan ha mer enn 2^h blad, vil $n! \leq l \leq 2^h$, slik at $n! \leq 2^h$. Ved å ta logaritme finner vi:

$$h \geq \log(n!) =^* \Omega(n \lg n)$$

* $\lg(n!) = \Theta(n \lg n)$

Dermed har vi vist at sammenligningssorteringer har en nedre grense på $\Omega(n \lg n)$. Dette gjør at **heapsort og merge sort er asymptotiske optimale sammenlignings-sorteringer** siden de har en øvre grense på $O(n \lg n)$. Altså, deres øvre grense er den nedre grensen for sorteringer av deres type.

$\mathrm{T_W}(n) = \mathrm{O}(\infty)$

$\mathrm{T_W}(n) = \Theta(?)$

$\mathrm{T_W}(n) = \Omega(n \lg n)$

Worst-case: vi har etablert at nedre grense er $\boldsymbol{\Omega(n \lg n)}$. Legg merke til at øvre grense er ∞ siden algoritmer kan bli vilkårlig dårlige. Derfor blir det ingen generell felles øvre og nedre grense

$\mathrm{T_A}(n) = \mathrm{O}(\infty)$

$\mathrm{T_A}(n) = \Theta(?)$

$\mathrm{T_A}(n) = \Omega(n \lg n)$

Average-case: samme nedre og øvre grense som worst-case, siden antall permutasjoner fortsatt er $\boldsymbol{n!}$

$\mathrm{T_B}(n) = \mathrm{O}(\infty)$

$\mathrm{T_B}(n) = \Theta(?)$

$\mathrm{T_B}(n) = \Omega(n)$

Best-case: samme øvre grense som worst-case, men nedre grense er $\boldsymbol{\Omega(n)}$ fordi da er alle elementene sortert, slik av vi trenger kun å gå igjennom tabellen en gang og se at alt er der det skal.

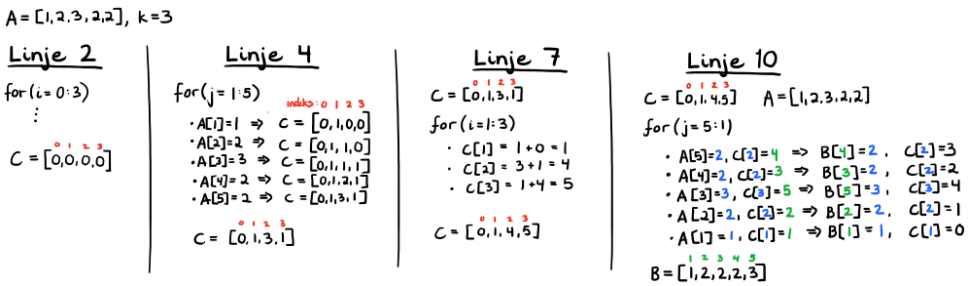
8.2 Tellesortering

Tellesortering antar at alle input elementene er et heltall i rekkevidden 0 til k (dvs. at input består av heltall fra en liten rekkevidde). **Når $\boldsymbol{k = O(n)}$ vil sorteringen kjøre på $\boldsymbol{\Theta(n)}$ tid**. Ved hvert inputelement x vil tellesortering avgjøre antall elementer som er mindre enn x og bruker dette for å plassere x direkte inn i sin posisjon i output arrayen. For eksempel dersom 17 elementer er mindre enn x, vil x plasseres i posisjon 18 i output arrayen. Vi må ta hensyn til at elementer kan ha samme verdi, slik at de ikke blir plassert i samme posisjon.

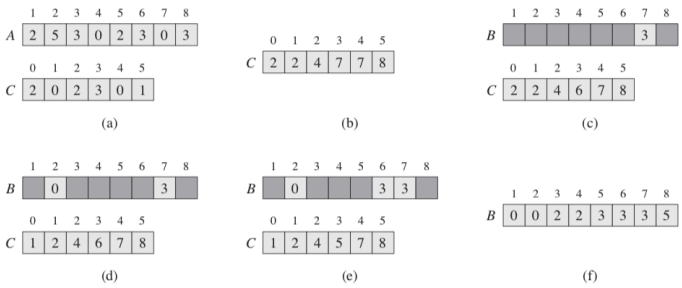
```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

COUNTING-SORT – stabil sortering

Denne metoden vil sortere input elementene vha antall elementer som er mindre eller lik element x. Denne metoden tar inn en array $A[1..n]$ som skal sorteres, en array $B[1..n]$ som skal holde den sorterte outputen og k som er øvre grense for heltallene i A. Metoden vil lage en array $C[0..k]$ med indeks 0 til k. Legg merke til at indeksene i C representerer verdiene i A, slik at dersom $A = [1, 2, 3, 2, 2]$ vil $C[2] = 3$ siden det er tre 2'ere. For-løkken ved linje 2 vil fylle C med 0'ere. **For-løkken ved linje 4 vil sørge for at $C[i]$ er antall elementer lik i.** Dette gjøres ved å sette $C[A[j]]$ lik den forrige verdien pluss 1. I vårt eksempel vil $A[5] = 2$, slik at $C[2]$ øker med 1. Etter denne for-løkken vil $C = [0, 1, 3, 1]$ (se figur). **For-løkken ved linje 7 vil sørge for at $C[i]$ er antall elementer mindre eller lik i.** Dette gjøres ved å sette $C[i]$ lik verdien ved indeks i pluss verdien ved $i - 1$ (merk at vi starter ved $j = 1$). Siden dette gjøres for alle indeksene, vil verdien ved $i - 1$ være summen av alle de tidligere verdiene. I vårt eksempel vil $C[2] = 1 + 3 = 4$, slik at $C = [0, 1, 4, 5]$ (se figur).



For-løkken ved linje 10 vil gå fra enden av A til 1 (sikrer stabilitet) og vil fylle B. Dette gjøres ved å plassere elementet fra A i posisjonen gitt av verdien i C for dette elementet. Verdien i C vil deretter reduseres med 1, siden det nå er en mindre av dette elementet. Merk: linje 12 er kun nødvendig dersom elementene kan være like. For eksempel for element $A[5] = 2$ vil $C[2] = 4$, slik at dette elementet blir plassert ved posisjon 4 i B. Deretter blir $C[2]$ redusert med 1, slik at neste 2'er blir plassert ved posisjon 3 i B. Dermed vil B være den sorterte versjonen av A.



I figur (a) vil $C[i]$ være antall elementer lik i, mens i figur (b) vil $C[i]$ være antall elementer mindre eller lik i. Neste steg blir å fylle B. $A[8] = 3$ skal plasseres ved posisjon 7, siden $C[3] = 7$. Deretter blir $C[3]$ redusert med én. Denne prosessen gjentas helt til B er fylt med alle elementene fra A, men i sortert orden.

Merk: dersom alle elementer er distinkte behøver vi ikke å redusere verdiene i C.

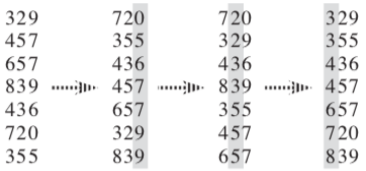
Tellesortering analyse

For-løkken i linje 2-3 tar tiden $\Theta(k)$, for-løkken i linje 4-5 tar tiden $\Theta(n)$, for-løkken i linje 7-8 tar tiden $\Theta(k)$ og for-løkken i linje 10-12 tar tiden $\Theta(n)$. Den totale tiden er derfor $\Theta(k + n)$. **Tellesortering blir som regel brukt i tilfeller der $k = O(n)$ (dvs. k er mindre eller lik n), slik at kjøretiden er $\Theta(n)$.** Tellesortering vil slå den nedre grensen på $\Omega(n \lg n)$ fordi det er ikke en sammenligningssortering. Denne algoritmen vil ikke sammenligne noen av elementene i inputen, men bruke heller elementverdiene for å plassere elementene i den sorterte arrayen.

En viktig egenskap ved tellesortering er at den er **stabil, som vil si at tall med samme verdi vil ha samme relative rekkefølge i output som i input**. Altså, rekkefølgen til duplikater vil ikke endres av sorteringen, slik at den som kommer først i input vil også komme først i output. Dette er viktig i tellesortering, fordi den brukes som subrutine i radikssortering som krever stabil sortering for å fungere (mer senere).

8.3 Radikssortering

Radikssortering vil sortere tall i en array A basert på signifikante siffer, der det **begynner med det minst signifikante sifferet**. For eksempel for [170, 93, 2] vil den først se på enerens plass [17<u>0</u>, <u>2</u>, 9<u>3</u>], deretter vil den se på tierens plass [<u>0</u>2, 1<u>7</u>0, <u>9</u>3] og til slutt på hundredes plass [<u>0</u>02, <u>0</u>93, <u>1</u>70]. **For å sortere de signifikante sifrene bruker radikssortering en annen stabil sorteringsalgoritme**, for eksempel tellesortering, merge sort eller innsettingssortering Antall signifikante siffer er d, så arrayen må sorteres d ganger for at den skal bli fullstendig sortert.




```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

<u>RADIX-SORT – stabil sortering basert på signifikante siffer</u>

Denne metoden vil sortere tallene i A og vi antar at alle disse tallene har d signifikante siffer. For-løkken vil sortere tallene fra minste signifikant siffer (1) til største (d). Ved hvert signifikant siffer vil tallene sorteres vha en annen sorteringsalgoritme. Denne prosessen vil fortsette helt til tallene har blitt sortert basert på alle de signifikante sifrene. Da vil A være en sortert array.

A = [14, 23, 13, 35, 15]

1) · Stabil : (23,13), (14), (35,15)
· ustabil : (13,23), (14), (35,15)

10) · Stabil : (13,14,15), (23), (35)
· ustabil : (15,14,13), (23), (35)

Stabil sorteringsalgoritme

For at radikssorteringen skal gi korrekt sortering, er det et krav at **sorteringsalgoritmen som brukes for å sortere de signifikante sifrene er stabil**. Altså, at duplikate elementer har samme rekkefølge i output som de har i input. Grunnen til dette er at dersom vi skal sortere (15, 13, 14) vil disse ved første signifikant siffer (5, 3, 4) sorteres som (13, 14, 15). Ved andre signifikant siffer (1, 1, 1) vil disse være identiske. Derfor er det kritisk at sorteringsalgoritmen er stabil, slik at rekkefølgen fra sorteringen av første signifikant siffer blir bevart og output blir (13, 14, 15). Noen stabile sorteringsalgoritmer er:

- **Innsettingssortering** – flytter kun på elementet dersom det er større og ikke når de er like
- **Merge sort** – ved fusjoneringen bruker vi $L \leq R$, der elementer i L opprinnelig var før elementer i R
- **Tellesortering** – elementer blir plassert fra ende til start, slik at det bakerste duplikatet i A blir plassert ved høyeste posisjon i B
- **Boblesortering** – elementer vil kun "boble" opp dersom de er større enn forrige element, og ikke hvis de er like.

Heapsort og quicksort er ikke stabile. Når vi skal velge en stabil sorteringsalgoritme, vil vi som regel **velge den raskeste**. Eks: dersom det står mellom merge sort og innsettingssortering, vil vi velge merge sort siden den er raskere.

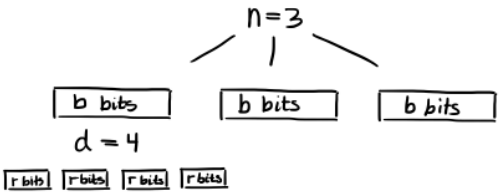
Radikssortering er selv en stabil sortering. En ulempe med radikssortering basert på tellesortering er at den **ikke sorterer *in place***, noe de fleste

sammenligningssorteringsalgoritmene gjør. Når bruk av minne er viktig, vil vi derfor foretrekke en *in-place* algoritme slik som quicksort.

Radikssortering analyse

Hvis arrayen inneholder n tall med d sifre som hver kan ha k mulige verdier (0 til $k-1$), vil **kjøretiden til radikssortering være $\Theta(d(n+k))$** hvis den stabile sorteringen bruker $\Theta(n+k)$ tid. Dette fordi kjøretiden er $\Theta(n+k)$ for hver av de d sifrene. Dersom $k = \Theta(\log n)$ vil kjøretiden til radikssorteringen bli $\Theta(dn)$.

Radikssortering – lineær kjøretid *ikke i forelesning*



Når d er en konstant og $k = O(n)$, kan radikssortering kjøre på lineær tid. Vi har n antall ord som hver består av b bits. Et slikt ord kan deles opp i d tall som hver består av $r \leq b$ bits. Dermed vil $d = b/r$ og $k = 2^r$ siden r bits vil kunne ha 2^r ulike verdier. Kjøretiden til radikssorteringen blir derfor $\Theta\big((b/r)(n+2^r)\big)$, dersom den stabile sorteringen bruker tiden $\Theta(n+k)$. Dersom $b = O(\lg n)$, kan vi velge $r = b \approx \lg n$, noe som gjør at $\Theta\big((b/r)(n+2^r)\big) = \Theta\left((b/b)\big(n+2^{\lg n}\big)\right) = \Theta(n)$. Altså vil radikssorteringen ha lineær kjøretid, slik at det virker det som om den er bedre enn quicksort, som har forventet kjøretid $\Theta(n \lg n)$. De konstante faktorene gjemt i Θ-notasjonen kan likevel være svært forskjellige. Det kan hende at radikssortering bruker færre passeringer over de n elementene enn quicksort, men hver passering kan ta signifikant lengre tid. Hvilken vi foretrekker vil derfor avhenge av egenskapene til implementasjonen, underliggende maskinvare og input dataen.

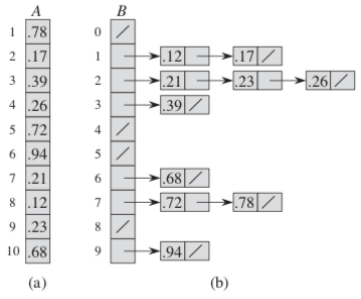
8.4 Bøttesortering

Bøttesortering antar at input blir tatt fra en uniform og uavhengig distribuering (dvs. alle elementer er like sannsynlig) og har en **average-case kjøretid på $O(n)$**. Bøttesorteringen deler intervallet $[0, 1)$ inn i n like store delintervaller, kalt **bøtter**, og vil så fordele de n input tallene inn i bøttene. Vi kan forvente at få elementer havner i samme bøtte siden de er uniformt og uavhengig fordelt. Hver bøtte vil være en linked list og vi vil bruke en annen sorteringsalgoritme (eks: innsettingssortering) for å sortere elementene i hver bøtte. Til slutt vil vi slå sammen alle bøttene til den endelige sorterte arrayen.

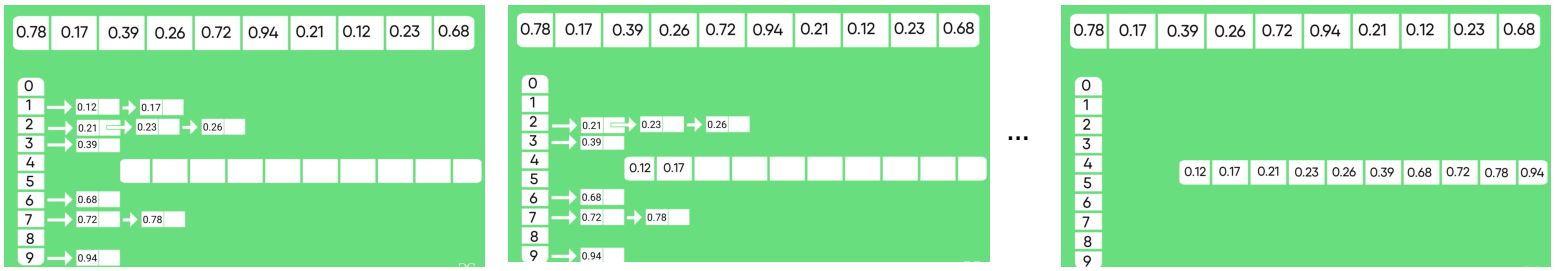
```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4      make B[i] an empty list
5  for i = 1 to n
6      insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8      sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

<u>BUCKET-SORT – sorterer elementer fra 0 til 1</u>

Denne metoden tar inn en array av n elementer som har en verdi mellom 0 og 1. Vi lager en matrise $B[0..n-1]$. For-løkken ved linje 3 vil sørge for at hver posisjon i B er en tom linked liste, som skal representere en bøtte. For-løkken ved linje 5 vil fylle bøttene i B basert på elementverdiene. Hvilken bøtte elementet skal plasseres i er gitt av verdien multiplisert med n. For eksempel kan $A[i] = 0.78$ og $n = 10$, slik at $\lfloor nA[i] \rfloor = \lfloor 7.8 \rfloor = 7$, og dermed vil 0.78 bli plassert i linked listen 7. Et annet eksempel er $A[i] = 0.21$ og $n = 10$, slik at $\lfloor nA[i] \rfloor = \lfloor 2.1 \rfloor = 2$, og dermed vil 0.21 bli plassert i linked listen 2. Til slutt vil alle elementene i A være plassert i linked listene. For-løkken ved linje 7 sørger for at alle linked listene i B



blir sortert vha innsettingssortering. Til slutt vil vi slå sammen alle linked listene. Fordi elementene blir plassert basert på verdi, vil linked listene være sortert i forhold til hverandre (se figur). Derfor kan output arrayen lages ved å direkte fylle inn $B[0]$, $B[1]$, … $B[n-1]$.



Bøttesortering analyse

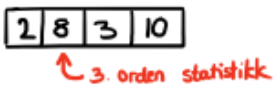
$$T_W(n) = \Theta(n^2)$$
$$T_A(n) = \Theta(n)$$
$$T_B(n) = \Theta(n)$$

For å analysere kjøretiden, kan vi observere at alle linjer bortsett fra linje 8 har worst-case kjøretid $O(n)$. Kjøretiden til bøttesorteringen vil derfor avhenge av tiden det tar å kalle innsettingssorteringen på de n bøttene. Figuren til høyre viser kjøretiden for INSERTION-SORT på n bøtter med forventet lengde $\Theta(1)$. Kjøretiden til bøttesorteringen blir dermed:

- **Worst-case:** alle n elementer blir plassert i samme bøtte, slik at kjøretiden blir $\Theta(n^2)$
- **Average-case:** de n elementene blir fordelt over bøttene, slik at innsettingssortering bruker $O(1)$ tid for å sortere hver bøtte. Dermed vil kjøretiden til bøttesorteringen bli $\Theta(n)$.

Merk: dersom k er liten (for eksempel $k = 3$) vil vi ikke kunne anta at sorteringen av elementene i hver bøtte går på $O(1)$ tid, og dermed vil average-case kjøretid bli $\Theta(n^2)$.

Kapittel 9 – Medianer og orden statistikk



Hos et sett av n elementer vil ***i*'ende orden statistikk** være det i'ende minste elementet. For eksempel vil minimum være første orden statistikk ($i = 1$), mens maksimum er $n$te orden statistikk ($i = n$). En **median** er punktet halvveis ved settet. Når n er odde vil medianen være unik ved $i = (n+1)/2$. Når n er jevn vil det være to medianer ved $i = n/2$ og $i = n/2 + 1$. Uansett paritet vil **nedre median** være ved $i = \lfloor (n+1)/2 \rfloor$ og **øvre median** er ved $i = \lceil (n+1)/2 \rceil$. Ordet "median" vil som regel referere til nedre median. Vi skal nå se på problemet ved å velge i'ende orden statistikk fra et sett med n distinkte tall. Vi spesifiserer **seleksjonsproblemet**:

- **Input** – et sett A med n distinkte tall og et heltall i, der $1 \leq i \leq n$.
- **Output** – elementet $x \in A$ som er større enn nøyaktig $i-1$ andre elementer i A, dvs. x er elementet ved indeks i når elementene er sortert i stigende størrelse.

Vi kan løse dette problemet i $O(n \lg n)$ tid ved å sortere tallene vha heapsort eller merge sort og deretter indekse det i'ende elementet. Vi skal likevel se på algoritmer som har worst-case kjøretid $O(n)$.

9.1 Minimum og maksimum

En øvre grense for antall sammenligninger som trengs å finne minimum i et sett med n elementer vil være $n-1$, siden vi undersøker hvert element (unntatt det første) og holder styr på det minste elementet sett så langt.

```
MINIMUM(A)
1  min = A[1]
2  for i = 2 to A.length
3      if min > A[i]
4          min = A[i]
5  return min
```

MINIMUM – finner minste element i en array av n elementer

Denne metoden tar inn en array av n elementer og vil returnere det minste elementet. Vi lagrer første element som minimum og går deretter gjennom alle elementene (mer fra 2 til n: $n-1$). Dersom ett av disse er mindre enn minimum vil minimum settes lik dette elementet. Slik fortsetter det til vi har gått gjennom hele arrayen. Til slutt blir det minste elementet returnert.

Vi kan også finne maksimum med $n-1$ sammenligninger. $n-1$ er også nedre grense for å finne minimum. Vi kan tenke på dette som en turnering blant elementene, der den minste av to elementer vil vinne en kamp og går videre til neste. Hvert element bortsett fra vinneren må tape minst en kamp, og derfor må det $n-1$ sammenligner til for å bestemme minimum. Dette betyr at MINIMUM algoritmen er optimal mht. antall sammenligninger som utføres.

Finne minimum og maksimum samtidig

For å finne minimum og maksimum samtidig, kan vi finne de uavhengig av hverandre vha $n-1$ sammenligninger hver, så totalt antall sammenligninger blir $2n-2=\Theta(n)$.

Vi kan også finne minimum og maksimum vha $3\lfloor n/2 \rfloor$ sammenligninger. Minimum og maksimum sett så lang blir tatt vare på. I stedet for å sammenligne hvert element med begge disse ved kostnad på 2 sammenligninger per element, kan vi behandle elementer i par. Dette gjøres ved å ta to elementer som først sammenlignes med hverandre. Deretter sammenligner vi det minste med nåværende minimum og det største med nåværende maksimum, slik at det blir 3 sammenligninger for alle par med elementer ($\lfloor n/2 \rfloor$). Dersom n er odde vil vi sette både minimum og maksimum lik verdien til det første elementet, mens dersom n er jevn vil vi utføre en sammenligning av de første to elementene for å bestemme minimum og maksimum. I begge tilfeller vil resten av elementene behandles som par. Hvis n er odde vil vi utføre $3\lfloor n/2 \rfloor$ sammenligninger, mens hvis n er jevn vil vi utføre én initial sammenligning fulgt av $3(n-2)/2$ sammenligninger, som totalt blir $3n/2-2$. **I begge tilfeller vil antall sammenligninger være maksimalt $3\lfloor n/2 \rfloor$.**

9.2 Seleksjon i forventet lineær tid

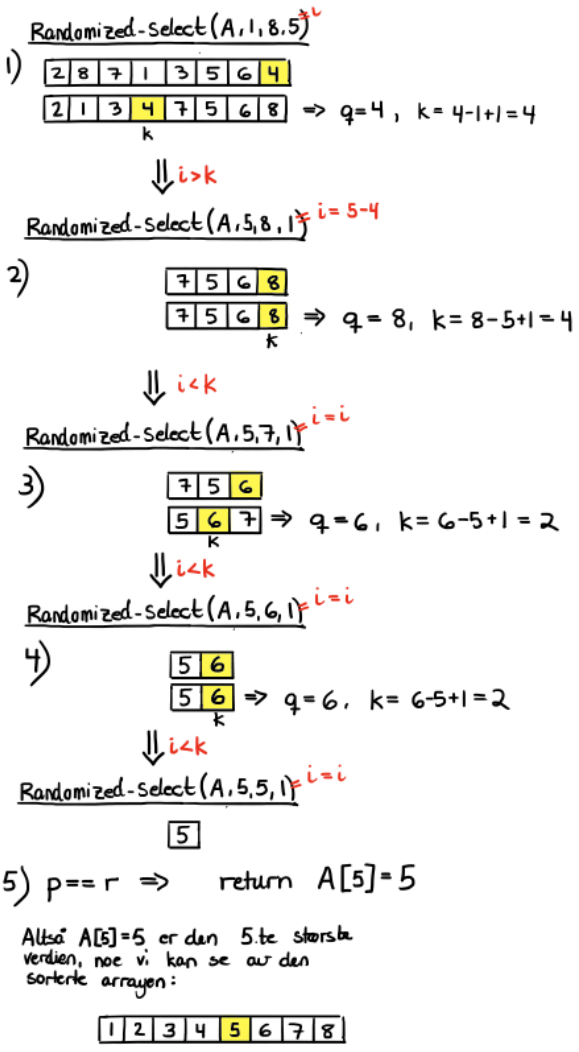
Seleksjonsproblemet virker mer komplisert enn å finne minimum, men asymptotisk kjøretid ved begge problemene er $\Theta(n)$. Seleksjonsproblemet kan løses av en divide-and-conquer algoritme kalt RANDOMIZED-SELECT. I likhet med quicksort vil vi **partisjonere input arrayen rekursivt**, men RANDOMIZED-SELECT vil kun **arbeide på en side av partisjonen** (husk: quicksort arbeider på begge sidene). Dette gjør at **RANDOMIZED-SELECT får en average-case kjøretid på $\Theta(n)$** istedenfor $\Theta(n \lg n)$ dersom elementene er distinkte.

RANDOMIZED-SELECT bruker metoden RANDOMIZED-PARTITION (s. 51), så derfor er det en randomisert algoritme siden oppførselen delvis bestemmes av det tilfeldige tallet.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k          // the pivot value is the answer
6      return A[q]
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

RANDOMIZED-SELECT – finner *i*'ende største element

Denne metoden tar inn en array, et start og sluttpunkt og indeks for elementet som skal finnes. Linje 1 sjekker grunntilfellet der subarrayen $A[p..r]$ kun inneholder ett element, slik at i'enden minste element må være $A[p]$. Hvis ikke denne utføres vil vi kalle på RANDOMIZED-PARTITION som vil dele A inn i to (muligens tomme) subarrays: $A[p..q-1]$ og $A[q+1..r]$, slik at alle elementer i den første er mindre enn $A[q]$ og alle elementer i den andre er større enn $A[q]$. Her vil q være indeksen til pivotelementet i den originale arrayen. I linje 4 regner vi ut k som skal være indeksen til pivotelementet i subarrayen vi ser på (Dvs. $A[p..r]$ med nåværende verdier av p og r, se figur til høyre). k vil altså være antall elementer som er mindre enn pivotelementet i subarrayen. Vi har tre situasjoner:



1. Dersom $i == k$ vil vi ha det heldige tilfellet at det tilfeldig valgte pivotelementet er elementet vi søker etter, og vi returnerer i så fall dette elementet.

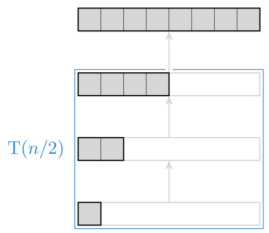
2. Dersom $i < k$ vil elementet vi søker etter være mindre enn pivotelementet. Derfor må vi ta et rekursivt kall der vi ser etter i'ende største element i $A[p..q-1]$, altså den nedre subarrayen.

3. Dersom $i > k$ vil elementet vi søker etter være større enn pivotelementet. Derfor må vi ta et rekursivt kall der vi ser etter $(i-k)$'ende største element i $A[q+1..r]$. Merk: vi ser på $i = i - k$ fordi når vi ser på subarrayen til høyre vil vi allerede ha k elementer som er mindre, så derfor står det igjen $i - k$ elementer for at et element skal ha i elementer som er mindre.

På figuren til høyre kan vi se et eksempel. Her har vi ikke fått tilfelle 1, så da vil metoden fortsette helt til subarrayen får kun ett element og $p == r$. Prosedyren kunne ha blitt avsluttet tidligere dersom det tilfeldig valgte pivotelementet ble $A[5] = 5$, fordi da ville $i = k = 5$ og første tilfellet hadde blitt utført.

RANDOMIZED-SELECT analyse

Worst-case kjøretid for RANDOMIZED-SELECT er $\Theta(n^2)$, fordi dette er tilfellet der partisjoneringen alltid er rundt største eller minste gjenværende element. Samtidig er det svært usannsynlig at dette skjer fordi metoden er randomisert.



Ved denne metoden vil partisjoneringen ta tiden $\Theta(n)$ og vi har et rekursivt kall på halvparten av elementene (se figur). Derfor vil rekurrensen bli:

$$T(1) = 1$$
$$T(n) = T(n/2) + n$$

Vi løser denne vha iterativ metode fremgangsmåte 2 (s. 54):

$$\begin{aligned} \mathrm{T}(n) &= n \\ &\quad + \mathrm{T}(n/2) && (1) \end{aligned}$$

$$\begin{aligned} \mathrm{T}(n) &= n \\ &\quad + n/2 && (1) \\ &\quad + \mathrm{T}(n/4) && (2) \end{aligned}$$

$$\begin{aligned} \mathrm{T}(n) &= n \\ &\quad + n/2 && (1) \\ &\quad + n/4 && (2) \\ &\quad + \mathrm{T}(n/8) && (3) \end{aligned}$$

Vi kan se et mønster at neste ledd er gitt av $T(n/2^i)$. For å få $T(1)$, må $i = \lg n$. Dermed får vi:

$$\begin{aligned} \mathrm{T}(n) &= n \\ &\quad + n/2 && (1) \\ &\quad + n/4 && (2) \\ &\quad + n/8 && (3) \\ &\quad \vdots && \vdots \\ &\quad + \mathrm{T}(n/2^{\lg n}) && (\lg n) \end{aligned}$$

$$\begin{aligned} \mathrm{T}(n) &= n \\ &\quad + n/2 && (1) \\ &\quad + n/4 && (2) \\ &\quad + n/8 && (3) \\ &\quad \vdots && \vdots \\ &\quad + 1 && (\lg n) \end{aligned}$$

Dersom n er en toerpotens, dvs. $n = 2^k$, vil $T(n) = 2n - 1$.

Dette fordi $n = 2^k$, gir:

$$\begin{aligned} T(n) &= 2^k + \frac{2^k}{2} + \frac{2^k}{4} + \cdots + 1 \\ &= 2^k + 2^{k-1} + 2^{k-2} + \cdots + 1 \\ &= 2^{k+1} - 1 \\ &= 2 \cdot 2^k - 1 \\ &= 2n - 1 \end{aligned}$$

Dette kan verifiseres med substitusjonsmetoden (s. 58):

1. **Grunntilfellet:** $T(1) = 1 = 2 \cdot 1 - 1 = 1$
2. **Induktivt steg:** Vi antar at $T(n/2) = 2(n/2) + n/2$ og vil vise at $T(n) = 2n - 1$. Vi får:

$$\begin{aligned} T(n) &= T(n/2) + n \\ &= (2(n/2) - 1) + n \\ &= n - 1 + n \\ &= 2n - 1 \end{aligned}$$

Dermed har vi vist at RANDOMIZED-SELECT har average-case kjøretid $\Theta(n)$. Figuren til høyre viser kjøretidene ved de ulike tilfellene. Legg merke til at det er samme som Quicksort, siden begge avhenger av PARTITION metoden.

$$\begin{aligned} \mathrm{T}_{\mathrm{W}}(n) &= \Theta(n^2) \\ \mathrm{T}_{\mathrm{A}}(n) &= \Theta(n) \\ \mathrm{T}_{\mathrm{B}}(n) &= \Theta(n) \end{aligned}$$

9.3 Seleksjon med worst-case lineær kjøretid

Vi skal nå se på en seleksjonsalgoritme der også worst-case kjøretid er $O(n)$. Algoritmen SELECT vil finne det ønskede elementet ved å rekursivt partisjonere input arrayen, men nå vil **vi garantere en god oppdeling**! **Kjernen i Select er å finne et godt pivotelement, noe den gjør vha partisjon metoden.**

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2       return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6       return A[q]
7  elseif i < k
8       return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

```
SELECT(A, p, r, i)
1  if p == r
2       return A[p]
3  q = GOOD-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6       return A[q]
7  elseif i < k
8       return SELECT(A, p, q − 1, i)
9  else return SELECT(A, p + 1, r, i − k)
```

Her kan vi se at SELECT fungerer på samme måte som RANDOMIZED-SELECT, bortsett fra at den kaller på GOOD-PARTITION istedenfor RANDOMIZED-PARTITION (og de rekursive kallene er oppdatert).

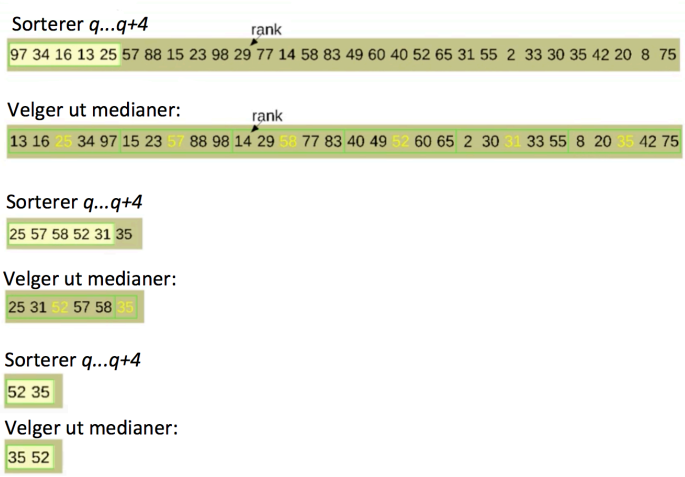
```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Vi har tidligere sett på metoden PARTITION som bruker $A[r]$ som pivotelement. Select algoritmen er basert på en modifisert versjon av denne.

```
GOOD-PARTITION(A, p, r)
1  n = r − p + 1
2  m = ⌈n/5⌉
3  create B[1 . . m]
4  for i = 0 to m − 1
5      q = p + 5i
6      sort A[q . . q + 4]
7      B[i] = A[q + 3]
8  x = SELECT(B, 1, m, ⌊m/2⌋)
9  return PARTITION-AROUND(A, p, r, x)
```

GOOD-PARTITION – garantert god oppdeling

Denne metoden vil garantere en god partisjonering pga et godt pivotelement. Vi lar n være antall elementer i subarrayen. Deretter deler vi subarrayen inn i $\lceil n/5 \rceil$ grupper med 5 elementer hver (siste gruppe vil ha gjenværende n mod 5, dvs. resten). Deretter må vi finne medianen til hver av disse gruppene, og disse skal lagres i array B. For-løkken ved linje 4 vil finne disse medianene ved å først lokalisere gruppen, deretter sortere den og til slutt hente ut element 3 fra gruppen (vil være medianen). Denne blir lagt til arrayen B, som etter for-løkken derfor vil inneholde m medianer. Ved linje 8 bruker vi SELECT metoden for å finne medianen av disse m medianene, altså vil GOOD-PARTITION brukes på arrayen av medianer. Dette blir gjentatt helt til vi står igjen med x som vil være medianen av medianene. Deretter sender vi x som input til PARTITION-AROUND metoden, og resultatet fra denne skal returneres.



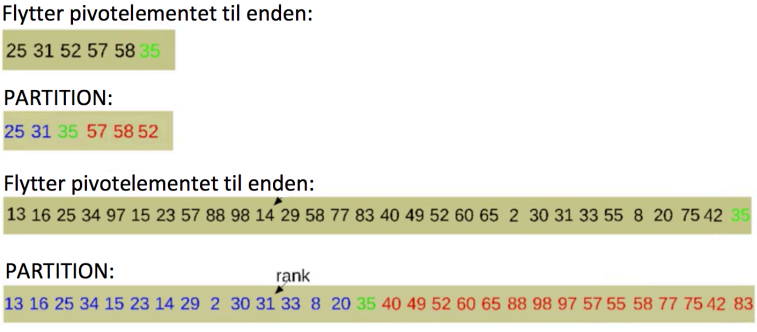
Figurene viser et eksempel. Vi begynner med arrayen A og lager grupper med 5 elementer. Hver gruppe sorteres slik at vi kan finne medianen til hver gruppe. Merk: siden det er så få elementer vil sorteringen ta lineær tid. Det dannes en array av medianer, og prosessen gjentas helt til vi står igjen med en median. I dette tilfellet er det et jevnt antall medianer så derfor vil medianen være den laveste: $x = 35$.

```
PARTITION-AROUND(A, p, r, x)
1  i = 1
2  while A[i] ≠ x
3      i = i + 1
4  exchange A[r] and A[i]
5  return PARTITION(A, p, r)
```

PARTITION-AROUND – partisjon med pivotelement som input

Denne metoden tar inn pivotelementet x, som er medianen av medianene og partisjonen skal utføres rundt denne. Vi vet at PARTITION metoden bruker elementet ved r som pivotelement, så for å få x til å bli pivotelementet, må vi flytte det til posisjon r. Deretter kan vi kalle på den vanlige PARTITION metoden som vil bruke $x = A[r]$ som pivotelement og dele opp arrayen som før. Dermed har vi sikret en god oppdeling fordi vi har brukt et godt pivotelement.

Figuren viser fortsettelsen av eksempelet over. Her kan vi se at det valgte pivotelementet blir flyttet til enden og deretter blir det utfør en partition etter denne. Videre blir resten av SELECT prosedyren kjørt, altså det sjekkes om det i'ende elementet er lik, større eller mindre pivotelementet. x blir returnert dersom $i = k$, ellers vil SELECT kalles rekursivt.

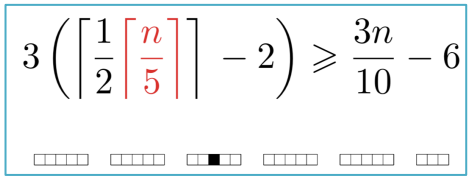


SELECT analyse

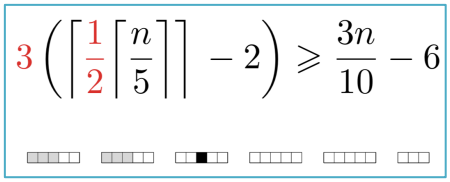
For å analysere kjøretiden til SELECT må vi først bestemme nedre grense for antall elementer som er større enn pivotelementet x. Denne vil være:

$$3\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil - 1\right) \geq \frac{3n}{10} - 6$$

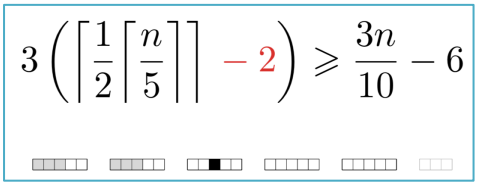
Forklaringen er som følger:



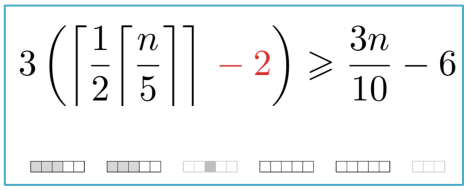
Vi har delt inn i $\lceil n/5 \rceil$ grupper



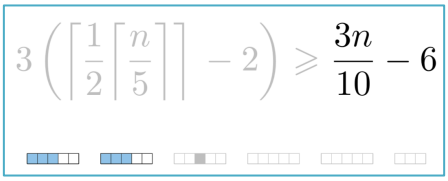
Minst halvparten bidrar med 3 verdier mindre enn pivot



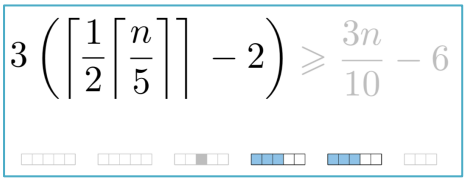
Unntatt én om $\lceil n/5 \rceil > n/5$ (dvs. den siste gruppen som er fylt med n mod 5)



og unntatt pivotgruppen (kan hende den kun består av tre elementer inkludert pivotelementet).



Minst så mange elementer vil være mindre enn x



Minst så mange elementer vil være større enn x

Dette gjør at vi er garantert en viss prosent på hver side av pivot, slik at worst-case kjøretid blir $T(n) = \Theta(n)$, altså vil kjøretiden være lineær. Dette kan bevises med substitusjon (bonusmateriale). Rekurrensen er:

$$T(n) \leq T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$$

der $T(\lceil n/5 \rceil)$ representerer rekursiv bruk av SELECT for å finne medianen av $\lceil n/5 \rceil$ gruppemedianer, $T(7n/10 + 6)$ representerer kall i største "halvdel" med $n - (3n/10 - 6)$ elementer og $O(n)$ representerer PARTITION, gruppering, sortering av grupper, osv. Vi lar a være konstanten fra O-notasjonen, slik at:

$$T(n) \leq T(\lceil n/5 \rceil) + T(7n/10 + 6) + an$$

Vi ønsker å vise at $T(n) \leq cn$ med substitusjon, og antar denne grensen for de rekursive kallene: $T(\lceil n/5 \rceil) \leq c\lceil n/5 \rceil$ og $T(7n/10 + 6) \leq c(7n/10 + 6)$. Dermed vil:

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + \big(an - (cn/10 + 7c)\big) \\ &\leq cn \end{aligned}$$

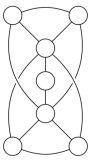
hvis c er stor nok, slik at $an \leq (cn/10 + 7c)$.Feks kan $c \geq 20a$, hvis $n \geq 140$. Dermed har vi vist at SELECT vil ha kjøretid $\Theta(n)$.

Forelesning 5 – Rotfaste trestrukturer

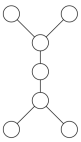
Rotfaste trær gjenspeiler rekursiv dekomponering. **I binære søketrær er alt i venstre deltre mindre enn rota, mens alt i høyre deltre er større**, og det gjelder rekursivt for alle deltrær! Hauger (max-heaps) er enklere: alt er mindre enn rota. Det begrenser funksjonaliteten, men gjør dem billigere å bygge og balansere. Læringsmålene er:

- **Forstå hvordan heaps fungerer, og hvordan de kan brukes som prioritetskøer (PARENT, LEFT, RIGHT, MAX-HEAPIFY, BUILD-MAX-HEAP, HEAPSORT, MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, HEAP-MAXIMUM, og tilsvarende for min-heaps)**
- Forstå HEAPSORT
- Forstå hvordan rotfaste trær kan implementeres
- **Forstå hvordan binære søketrær fungerer (INORDER-TREE-WALK, TREE-SEARCH, ITERATIVE-TREE-SEARCH, TREE-MINIMUM, TREE-MAXIMUM, TREE-SUCCESSOR, TREE-PREDECESSOR, TREE-INSERT, TRANSPLANT, TREE-DELETE (ikke grundig forståelse av to siste)**
- Vite at forventet høyde for et tilfeldig binært søketre er $\Theta(\lg n)$
- Vite at det finnes søketrær med garantert høyde på $\Theta(\lg n)$

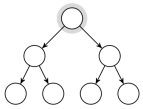
Trær - forelesning



Dette er urettet graf som består av en mengde noder/vertekser V og en mengde kanter E. Grafen kan beskrives som $G = (V, E)$. En kant vil være en kobling mellom et par $\{u, v\}$ med noder. Vi sier at denne er syklisk, siden det er en syklisk kobling mellom nodene.

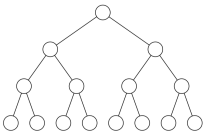


Dette er et fritt tre, som er en sammenhengende, asyklisk og urettet graf. Dette treet har en sti som kobler sammen hvert par, og det er én kant unna usammenhengende eller syklisk. For slik trær vil $|E| = |V| - 1$.

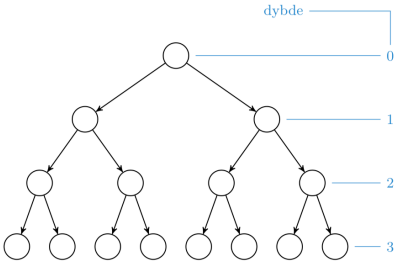


Dette er et rotfast tre, som ofte er "rettede" grafer med rettede stier vekk fra rota. Et rotfast tre er et fritt tre med en angitt rotnode som ikke har foreldrenoder.

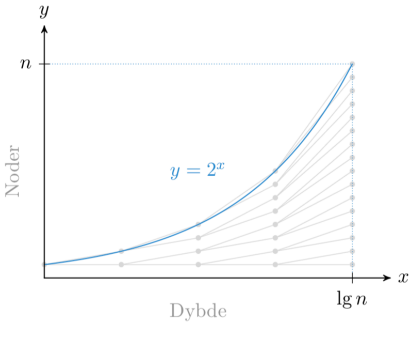
I et ordnet tre har barna en ordning, mens i et posisjonstre har hvert barn en posisjon.



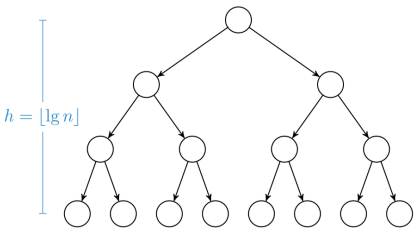
Dette er et komplett binærtre, som vil si at hver node har to barnenoder og bunnivået er fullstendig fylt. Et binærtre er et posisjonstre der hver node har to barneposisjoner, og de kan tolkes som ordnede trær med ekstra informasjon.



Dybden er antall kanter unna rota (figur til venstre). Figuren til venstre viser et binært tre, og som vi kan se vil **antall noder ved dybde x være gitt av 2^x. Dybden ved bladene til det binære treet vil være lg n.**



Høyden er den maksimale dybden, altså dybden fra rota til bladene. For et binært tre vil dybden være $\lg n$, fordi totalt antall noder er $n = 2^x \Rightarrow x = \lg n$. En annen måte å tenke er at størrelsen ved nivå i vil være $n/2^i$. Ved bunnen vil størrelsen til delproblemene være 1, noe som er tilfellet ved $i = \lg n$.

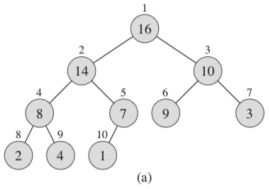


Binærtre er **fullt** når alle interne noder har to barnenoder, det er **balansert** når alle løvnoder/blad har ca. samme asymptotisk dybde og det er **komplett** når alle løvnoder/blad har nøyaktig samme dybde.

Kapittel 6 Heapsort

Heapsort er en annen sorteringsalgoritme. I likhet med mergesort har den kjøretid $O(n \lg n)$, men forskjellen er at den sorterer *in place* (i likhet med innsettingssortering), fordi kun et konstant antall array elementer blir lagret på utsiden av input arrayen ved ethvert tidspunkt. Heapsort bruker en datastruktur som kalles heap for å kontrollere informasjonen, og denne strukturen lager en effektiv prioritetskø.

6.1 Heap = haug



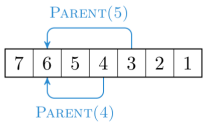
(Den binære) heap datastrukturen er et array objekt som kan ses på som et nesten fullstendig binært tre (figur a). Hver node i treet korresponderer til et element i arrayen (figur b). Treet er fullstendig fylt, bortsett fra muligens det laveste nivået som fylles fra venstre. En array A som representerer en heap vil ha to attributter: $A.length$ som gir størrelsen til arrayen (kan være tomme plasser) og *A.heap-size* som representerer antall elementer i heapen som er lagret i arrayen ($= A.length$ hvis arrayen er fullstendig fylt) . Altså, vil kun elementene i $A[1 \ldots A.heap - size]$, der $0 \leq A$.heap-size $\leq A.length$, være gyldige elementer fra heapen. Roten til treet er $A[1]$.

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

For en node med indeks i har vi følgende metoder:

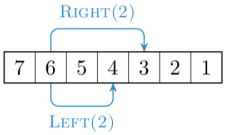
PARENT – finner foreldrenoden

Denne metoden finner indeks til foreldrenoden, som er gitt av $\lfloor i/2 \rfloor$



LEFT – finner venstre barnenode

Denne metoden finner indeks til venstre barnenode, som er gitt av $2i$

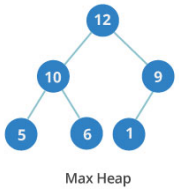


RIGHT – finner høyre barnenode

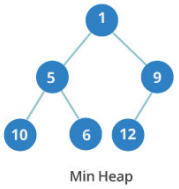
Denne metoden finner indeks til høyre barnenode , som er gitt av $2i + 1$

Det er to typer binære heaps:

- **Max-heaps – foreldrenodene er større enn barnenodene**, slik at det største elementet er lagret i roten og elementverdien reduseres nedover heapen. Vi sier at max-heap egenskapen er at for enhver node i bortsett fra roten vil $A[PARENT(i)] \geq A[i]$. **Vi bruker denne for heapsort**
- **Min-heaps – foreldrenodene er mindre enn barnenodene**, slik at det minste elementet er lagret i roten og elementverdien økes nedover heapen. Vi sier at min-heap egenskapen er at for enhver node i bortsett fra roten vil $A[PARENT(i)] \leq A[i]$.



Max Heap



Min Heap

Dersom vi ser på en heap som et tre, vil **høyden til en node i heapen være antall kanter på den lengste banen nedover fra noden til et blad**. Høyden til heapen er lik høyden til roten. En heap av n elementer er basert på et fullstendig binært tre, og vil derfor ha høyde $\Theta(\lg n)$ (to forgreininger, så bunnen er ved $n/2^i = 1$, $i = \lg n$). Grunnleggende operasjoner på heaps har kjøretid som er maksimalt proporsjonal med høyden til treet, altså $O(\lg n)$. Noen grunnleggende prosedyrer på heaps er:

- **MAX-HEAPIFY ($O(\lg n)$)** – opprettholder max-heap egenskapen
- **BUILD-MAX-HEAP ($O(n)$)** – lager en max-heap fra en ikke-ordnet input array
- **HEAPSORT ($O(n \lg n)$** – sorterer en array *in place*

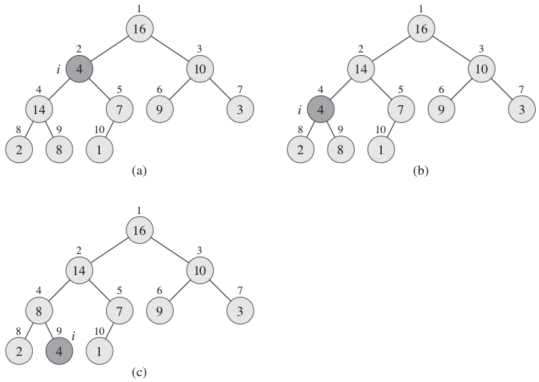
6.2 Opprettholde max-heap egenskapen

Metoden **MAX-HEAPIFY** brukes for å opprettholde max-heap egenskapen, nemlig at foreldrenodene er større enn barnenodene. Den antar at de binære trærne med rot $LEFT(i)$ og $RIGHT(i)$ er max-heaps, men at $A[i]$ kan være mindre enn barnenodene. MAX-HEAPIFY lar $A[i]$ synke ned til rett plass i max-heapen, slik at deltreet med rot i følger max-heap egenskapen.

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

MAX-HEAPIFY – opprettholder max-heap egenskapen

Denne metoden tar inn en array A og en indeks i, og den er basert på å finne indeksen til det største elementet av foreldrenoden og barnenodene. Metoden begynner med å hente indeksene til venstre og høyre barnenode. Dersom venstre barnenode er innenfor heapen og verdien til denne er større enn verdien til $A[i]$, vil *largest* lagres som indeksen til venstre barnenode. Hvis $A[i]$ er størst vil *largest* settes lik i. Dersom høyre barnenode er innenfor heapen og verdien til denne er større enn verdien til $A[largest]$, vil *largest* lagres som indeksen til høyre barnenode. Dermed vil *largest* være indeksen til største element. Hvis dette ikke er i vil $A[i]$ byttes med $A[largest]$, altså vil foreldrenoden bytte plass med den største av barnenodene. Deretter blir metoden kalt rekursivt, helt til den opprinnelige foreldrenoden er plassert slik at den er større enn barnenodene og max-heap egenskapen er oppfylt.



Figuren til høyre viser et eksempel på hvordan MAX-HEAPIFY fungerer. Noden med verdi 4 flyttes nedover helt til den ikke har noen barnenoder som er større.

MAX-HEAPIFY bruker tiden $\Theta(1)$ på å fikse forholdet mellom elementene $A[i]$, $A[LEFT(i)]$ og , $A[RIGHT(i)]$. Vi antar at det rekursive kallet utføres på deltreet til en av barnenodene. Worst-case tilfellet vil være når bunnivået i treet er nøyaktig halvfullt, og i dette tilfellet vil størrelsen til barnenodens deltre være $2n/3$ (worst-case: barnenoden vi går til har største deltre som inneholder $2/3$ av nodene). Derfor vil MAX-HEAPIFY bruke tiden $T(2n/3)$ på det rekursive kallet. Den totale kjøretiden blir:

$$T(n) \leq T(2n/3) + \Theta(1)$$

Master teoremet gir at **MAX-HEAPIFY har kjøretid $T(n) = O(\lg n)$.** Alternativt kan vi si at kjøretiden til MAX-HEAPIFY på en node med høyde h er $O(h)$.

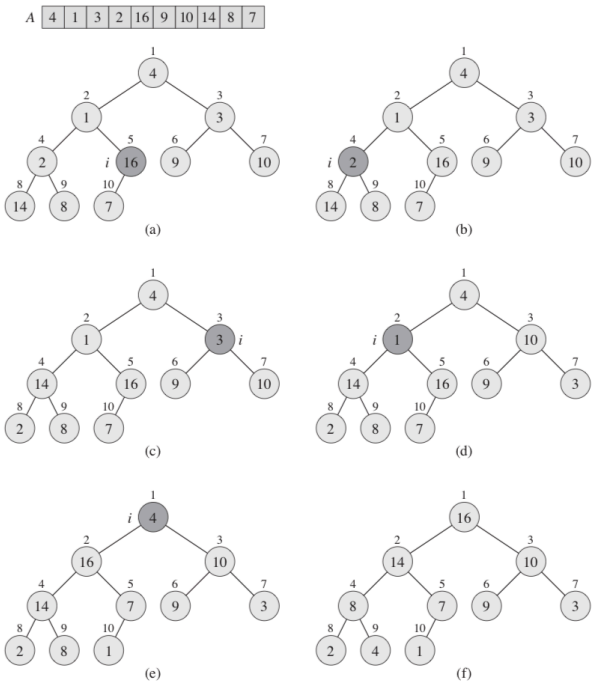
6.3 Bygge en heap (haug)

Metoden BUILD-MAX-HEAP brukes for å lage en max-heap fra en ikke-ordnet input array $A[1..n]$, der $n = A.length$. Dette gjøres ved å bruke MAX-HEAPIFY fra bunnen og oppover.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

BUILD-MAX-HEAP – lager en max-heap av en matrise *A*

Denne metoden vil ta inn en matrise A og lage en max-heap ved å fikse alle deltrærne. Metoden begynner med å sette *A.heap-size* = *A.length*, slik at hele A blir en heap. For-løkken ved linje to begynner med $i = \lfloor A.length/2 \rfloor$, som vil være foreldrenoden til den siste noden i arrayen (dvs. noden lengst ned til høyre i heapen. Arrayen og denne indeksen blir sendt som input til MAX-HEAPIFY som dermed vil sørge for at det "siste" deltreet blir en max-heap. Deretter vil for-løkken hoppe til neste node. Denne prosessen gjentas helt til roten blir sendt som input til MAX-HEAPIFY, som dermed vil gjøre at hele heapen blir en max-heap.



Figuren viser hvordan BUILD-MAX opererer. Her vil for-løkken begynne ved $i = \lfloor 10/2 \rfloor = 5$, altså ved elementet $A[5] = 16$. Siden barnenoden til denne er mindre enn 16, vil dette deltreet allerede være en max-heap. Derfor vil BUILD-MAX hoppe til neste node ved $i = 4$. Her ser vi at $A[4] = 2$ er mindre enn barnenoden, så derfor vil disse bytte plass i MAX-HEAPIFY. Tilslutt vil vi ha en max-heap.

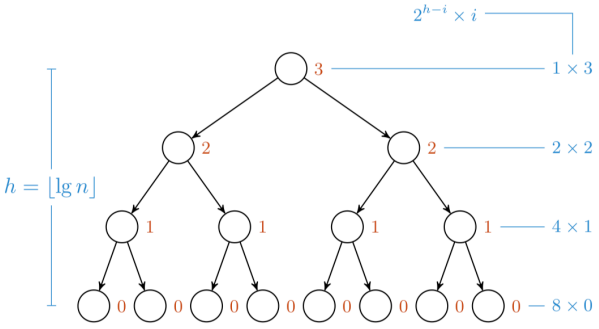
BUILD-MAX-HEAP bevis

For å vise at BUILD-MAX-HEAP fungerer riktig ser vi på loop invarianten: "Ved starten av hver iterasjon av for-løkken vil hver node $i + 1, i + 2, \ldots n$ være en rot for en max-heap". Vi må vise tre ting ved loop invarianten:

1. **Initialisering** – før første iterasjon vil $i = \lfloor n/2 \rfloor$, så alle nodene $i + 1, i + 2, \ldots n$ vil være blad med størrelse 1 som allerede er en max-heap.
2. **Vedlikehold** – barnenodene til node i vil ha høyere indeks enn i, så derfor gir loop invarianten at disse er røtter til max-heaps. Dette er kravet for MAX-HEAPIFY, så denne metoden vil sørge for at også i blir en rot for en max-heap og at $i + 1, i + 2, \ldots n$ fortsatt er røtter for max-heaps.
3. **Terminering** – ved terminering vil $i = 0$, slik at loop invarianten gir at alle nodene $1, 2, \ldots n$ er en rot for en max-heap, så derfor vil hele heapen være en max-heap.

BUILD-MAX-HEAP analyse

Hvert kall til MAX-HEAPIFY bruker $O(\lg n)$ tid og BUILD-MAX-HEAP gjør $O(n)$ slike kall. **Derfor vil kjøretiden til BUILD-MAX-HEAP være $O(n \lg n)$.** Dette er en korrekt øvre grense, men den er ikke asymptotisk tett.



Vi finner en tettere grense ved å bruke at kjøretiden til MAX-HEAPIFY på en node avhenger av høyden til denne noden i treet. Merk at vi bruker betegnelsen h på den totale høyden til treet (dvs. dybde fra rot til blad) og i for høyden til en bestemt node (dvs. $i = 0, 1, \ldots h$). En heap med n elementer vil ha $h = \lfloor \lg n \rfloor$ og antall noder ved høyden i vil være $n/2^i = 2^h/2^i = 2^{h-i}$. Kjøretiden til MAX-HEAPIFY på en node ved høyde i vil være $O(i) = i$. Derfor vil total kostnad per høyde være $2^{h-i} \cdot i$ (se figur). For å finne den totale kostnaden til BUILD-MAX-HEAP må vi summere kostnadene ved hver høyde av treet:

$$T(n) = \sum_{i=0}^{h} 2^{h-i} \cdot i = \sum_{i=0}^{h} \frac{2^h}{2^i} \cdot i = 2^h \sum_{i=0}^{h} \frac{i}{2^i} = n \cdot \sum_{i=0}^{h} \frac{i}{2^i} = \Theta(n) \cdot \sum_{i=0}^{h} \frac{i}{2^i}$$

Her vil $\sum_{i=0}^{h} 1/2^i$ representere gjennomsnittlig arbeid per node. Faktoren i gir at MAX-HEAPIFY vokser lineært, mens 2^i gir at antall kall til MAX-HEAPIFY synker eksponentielt. Dersom vi bruker at $\sum_{i=0}^{\infty} kx^k = x/(1-x)^2$ med $x = (1/2)^i$, får vi at:

$$\sum_{i=0}^{h} \frac{i}{2^i} \leq \sum_{i=0}^{\infty} i \cdot \left(\frac{1}{2}\right)^i = \frac{1/2}{1/4} = 2 = \Theta(1)$$

Dermed vil **kjøretiden til BUILD-MAX-HEAP bli $T(n) = \Theta(n)$**. Vi kan altså bygge en max-heap på lineær tid. BUILD-MIN-HEAP fungerer på samme måte, bortsett fra at MAX-HEAPIFY er byttet ut med MIN-HEAPIFY.

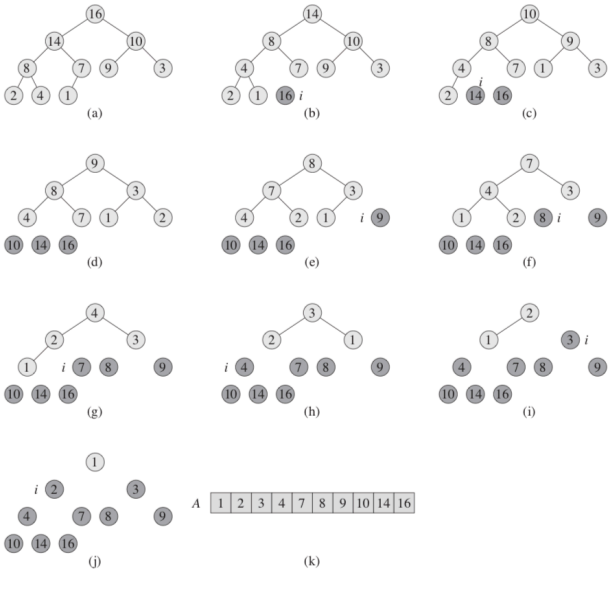
6.4 Heapsort algoritmen

Heapsort algoritmen bruker både BUILD-MAX-HEAP og MAX-HEAPIFY for å sortere input arrayen $A[1..n]$.

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

HEAPSORT – sorterer en array vha max-heap egenskapen

Denne metoden kaller på BUILD-MAX-HEAP for å lage en max-heap av input arrayen A. Deretter vil for-løkken gå igjennom alle elementene i max-heapen. Siden det største elementet i en max-heap er plassert ved roten $A[1]$, må for-løkken telle nedover fra n. For hvert element vil roten byttes med $A[i]$ slik at de store elementene blir plassert til høyre og størrelsen til heapen reduseres med 1. Dette byttet av elementer gjør at et nytt element har blitt rot, og vi kaller på MAX-HEAPIFY for å sørge for at det lages en max-heap på nytt. Deretter kan for-løkken gjentas for den nye største roten.



Figuren viser et eksempel. Vi begynner med $A[1] = 16$ som blir byttet ut med $A[n] = 1$. Dermed blir 1 plassert ved roten, men MAX-HEAPIFY gjør at dette elementet blir flyttet nedover og vi får max-heapen på figur b. Legg merke til at $A[n] = 16$. Etter neste steg vil $A[n-1] = 14$. Slik fortsetter prosessen helt til $A[1] = 1$ og dette er eneste node.

HEAPSORT algoritmen har kjøretid $O(n \lg n)$ siden kallet til BUILD-MAX-HEAP tar tiden $O(n)$ og hver av de $n-1$ kallene til MAX-HEAPIFY tar tiden $O(\lg n)$. Altså:

$$T(n) = O(n) + O\big((n-1)\lg n\big) = O(n \lg n)$$

Figuren viser kjøretiden til heapsort sammenlignet med de andre sorteringsalgoritmene. Loop invarianten ved HEAPSORT er at: ”alt utenfor heapen er større og sortert”, og den blir tilfredsstilt ved de tre tilfellene.

Algoritme	WC	AC/E	BC
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heap sort	$\Theta(n \lg n)$	—	$\Theta(n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)^*$	$\Theta(n \lg n)$
Counting sort	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$
Radix sort	$\Theta(d(n+k))$	$\Theta(d(n+k))$	$\Theta(d(n+k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)^\dagger$	$\Theta(n)$

6.5 Prioritetskøer

En god implementasjon av quicksort vil ofte være bedre enn heapsort. Heap datastrukturen har likevel mange bruksområder, for eksempel som effektive prioritetskøer. **En prioritetskø er en datastruktur som inneholder et sett *S* med elementer, som hver har en assosiert verdi kalt *key***. Linked-list og dynamisk tabell kan også brukes som prioritetskø med konstant innsettingsstid og lineær tid for å finne eller ta ut maksimum. En max-prioritetskø er basert på en max-heap.

En max-prioritetskø støtter følgende operasjoner:

```
HEAP-MAXIMUM(A)
1  return A[1]
```

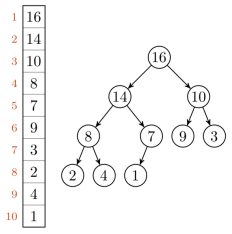
HEAP-MAXIMUM – finner største element i max-heap *A*

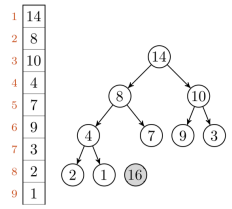
Det største elementet vil være først i en max-heap, så derfor vil denne metoden returnere $A[1]$. Dette tar konstant tid, slik at **kjøretiden er $\Theta(1)$.**

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

HEAP-EXTRACT-MAX – fjerner det største elementet i max-heap *A*

Denne metoden vil returnere det største elementet og fjerne dette fra max-heapen som deretter må oppdateres. Dersom heapen er tom skal en feilmelding utløses. Hvis ikke vil *max* settes lik roten, og roten settes lik det siste elementet i max-heapen. Deretter vil størrelsen til heapen reduseres med 1. Roten er altså byttet ut med det siste og derfor minste elementet i heapen, så vi må kalle på MAX-HEAPIFY for å sørge for at max-heap egenskapen oppfylles. Til slutt vil vi returnere *max*. Linjene 1-5 og 7 vil ta konstant tid, mens MAX-HEAPIFY bruker tiden $O(\lg n)$. **Derfor vil kjøretiden være $O(\lg n)$.**

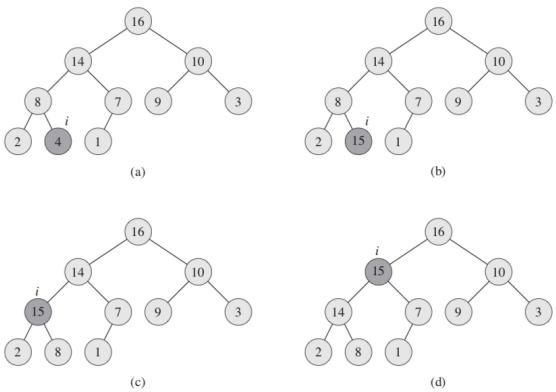




```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2        error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5        exchange A[i] with A[PARENT(i)]
6        i = PARENT(i)
```

HEAP-INCREASE-KEY – øker verdien til element

Denne metoden vil øke $A[i]$ til verdien gitt av *key*, og deretter oppdatere max-heapen. Dersom *key* er mindre enn elementet ved $A[i]$ vil en feilmelding utløses. Hvis ikke vil $A[i]$ settes lik *key*. Deretter vil metoden gå fra denne noden mot roten helt til den finner riktig plass for den nye verdien Dersom foreldrenoden har lavere verdi enn den nye verdien, vil disse bytte plass. Dette fortsetter helt til vi er ved roten eller foreldrenoden er større enn den nye verdien. Dermed blir max-heap egenskapen opprettholdt. Kjøretiden er $O(\lg n)$ siden dette er høyden.

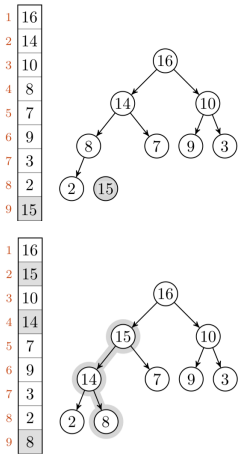


Figuren viser et eksempel der $A[9] = 4$ økes til $key = 15$, og noden flyttes opp til posisjon $i = 2$

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

MAX-HEAP-INSERT – setter inn en verdi

Denne metoden vil sette verdien gitt av *key* inn i max-heapen A. Dette gjøres ved å øke størrelsen til heapen med 1 og sette verdien ved enden lik $-\infty$. Deretter kaller den på HEAP-INCREASE-KEY på den siste posisjonen med *key*-verdien som input. Dette gjør at den siste verdien blir endret til *key* og HEAP-INCREASE-KEY sørger for at max-heap egenskapen blir opprettholdt (derfor setter vi ikke inn verdien direkte!). **Kjøretiden er $O(\lg n)$.**



Figuren viser et eksempel der $key = 15$ blir satt inn i $A[9]$ og deretter flyttet oppover for å oppfylle max-heap egenskapen.

Algoritme	Kjøretid
MAX-HEAPIFY	$O(\lg n)$
HEAP-MAX	$\Theta(1)$
HEAP-EXTRACT-MAX	$O(\lg n)$
HEAP-INCREASE-KEY	$O(\lg n)$
MAX-HEAP-INSERT	$O(\lg n)$
BUILD-MAX-HEAP	$\Theta(n)$

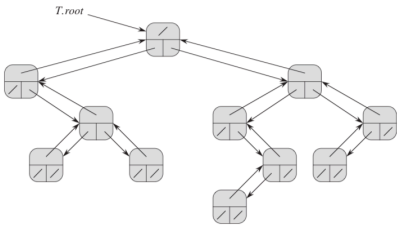
Som vi kan se på tabellen vil alle operasjoner på en prioritetskø basert på en heap ha kjøretid $O(\lg n)$. Merk: **grunnen til at en heap kan brukes for en prioritetskø er at den kan implementere disse metodene!**

10.4 – Representasjon av rotfaste trær

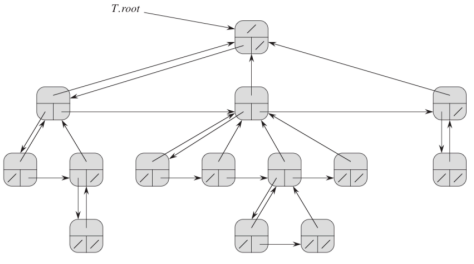
Vi skal se hvordan rotfaste trær kan representeres av en lenket datastruktur. I binære trær vil hver node ha maksimalt to barnenoder, mens i rotfaste trær kan noder ha et vilkårlig antall barnenoder. Hver node representeres av et objekt og vi antar at hver av nodene inneholder en *key* attributt. Resten av attributtene er pekere til andre noder.

Binære trær

Figuren viser hvordan vi bruker attributtene ***p*, *left* og *right*** for å lagre pekere til hhv. foreldrenoden, venstre og høyre barnenode til hver node i det binære treet T. Dersom $x.p = NIL$ vil x være roten. Dersom x ikke har noe venstre barnenode vil $x.left = NIL$, og tilsvarende for høyre barnenode. $\boldsymbol{T.root}$ vil peke mot roten til hele treet, så treet er tomt dersom $T.root = NIL$.



Rotfaste trær med ubegrenset forgreining



For å representere et rotfast tre, der hver node har maksimalt k antall barnenoder vil vi erstatte $left$ og $right$ attributtene med $child_1$, $child_2$, … $child_k$. Dette vil ikke fungere når antall barnenoder er ubegrenset (kan ikke tildele attributter på forhånd) og når k er stor vil mye plass kastes bort (de fleste nodene vil ha et lite antall barnenoder). Figuren viser en alternativ tilnærming som kun bruker $O(n)$ plass for et tre med n noder. Som før vil $x.p$ peke mot foreldrenoden og $T.root$ peker mot roten til treet. Pekerne mot barnenodene er derimot erstattet av:

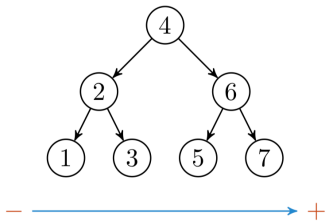
1. ***x*.left-child** – peker mot barnenoden som er lengst til venstre
2. ***x*.right-sibling** – peker mot søskennoden som er rett til høyre for x

Altså alle barnenodene peker mot sin foreldrenode, men foreldrenoden peker kun mot venstre barnenode og når de andre via denne. Dersom x ikke har noen barnenoder vil *x.left-child = NIL,* og dersom x er barnenoden helt til høyre for foreldrenoden vil *x.right-sibling = NIL.*

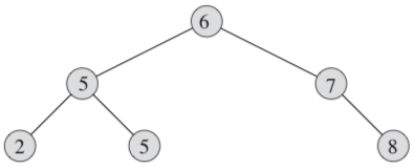
Kapittel 12 – Binære søketrær

Et søketre må støtte følgende operasjoner: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT og DELETE, og det kan derfor brukes som både ordbok og prioriteringskø. Kjøretiden til grunnleggende operasjoner på et binært søketre vil være proporsjonal med høyden til treet. **For et fullstendig binærtre med n noder vil worst-case kjøretid være $\Theta(\lg n)$, mens for et tre med en lineær kjede av n noder vil worst-case kjøretid være $\Theta(n)$.** Forventet høyde for et tilfeldig bygd binærtre er $O(\lg n)$, så derfor vil average-case kjøretid på slike trær være $\Theta(\lg n)$. Vi kan ikke alltid garantere tilfeldig bygde binærtre, man vi kan designe binære søketrær som har god garantert ytelse ved worst-case på grunnleggende operasjoner.

12. 1 Hva er et binært tre?



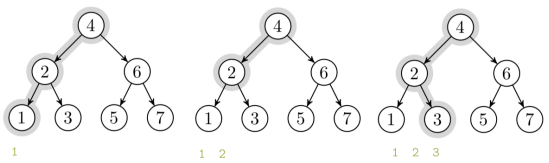
Et binært søketre er ordnet i et binært tre som kan representeres med en lenket datastruktur der hver node er et objekt. Hver node vil ha en *key*-verdi og attributtene $left$, $right$ og p som peker mot venstre og høyre barnenode og foreldrenoden. Nodene i et binært søketre er alltid ordnet slik at **binær-søketre egenskapen** oppfylles: **Hvis y er en node i venstre deltre til x vil $y.key \leq x.key$, mens hvis y er en node i høyre deltre til x vil $y.key > x.key$** (figur til høyre). Dvs. noder med *key* verdi som er lavere enn foreldrenoden vil plasseres til venstre, mens noder med *key* verdi som er høyere vil plasseres til høyre (huskeregel: høyeste verdier til høyre). På figuren til venstre kan vi se at roten har $x.key = 6$, slik at *key* verdiene 2, 5, 5 plasseres i venstre deltre og *key* verdiene 7 og 8 plasseres til høyre. **Denne egenskapen gjelder for alle noder i treet,** for eksempel vil 5 plasseres til høyre for 5, mens 2 plasseres til venstre.



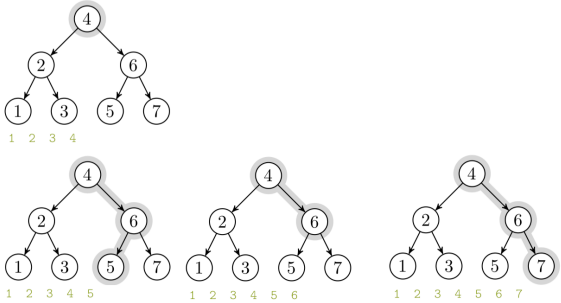
Binær-søketre egenskapen lar oss skrive ut alle *key*-verdiene i et binært søketre i sortert rekkefølge vha en enkel rekursiv algoritme, kalt en **ikke-ordnet tre traversering** (*inorder tree walk*). Denne vil skrive ut verdiene i venstre deltre, deretter roten til deltreet og til slutt verdiene i høyre deltre.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

<u>INORDER-TREE-WALK – printer *key* verdiene i binært søketre</u>
Denne metoden tar inn en node x og vil printe *key* verdiene til venstre deltre, til x og til høyre deltre vha rekursive kall. De rekursive kallene sørger for at alle verdiene i deltreet med x som rot blir printet og siden første rekursiv kall har $x.left$ som input, vil printingen bli i sortert rekkefølge. **Kjøretiden er $\Theta(n)$ for et binært søketre av n noder** (den er $\Omega(n)$ fordi den må innom alle nodene, og $T(n) = O(n)$ kan bevises vha substitusjonsmetoden).



Figuren viser hvordan INORDER-TREE-WALK kan brukes for å printe hele det binære søketreet dersom $x = T.root$. Den ser først på venstre deltre, deretter roten og tilslutt høyre deltre.



12.2 Operasjoner på binært søketre

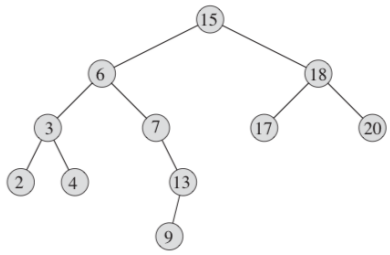
Et binært søketre støtter følgende operasjoner: MINIMUM, MAXIMUM, SEARCH, SUCCESSOR og PREDECESSOR, og **disse har kjøretid $O(h)$ der h er høyden til treet**.

Merk: kan også bruke iterativ metode med while-løkke

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

TREE-SEARCH – søker etter en node med en bestemt *key*

Denne metoden tar inn en peker mot roten og en *key*-verdi k, og vil returnere en peker mot en node som har $key = k$ dersom det eksisterer. Hvis ikke vil den returnere *NIL*. Metoden vil begynne søket ved roten, og vil returnere denne noden dersom $x = NIL$ eller $x.key = k$ (roten har *key*-verdien vi søker etter). Hvis ikke vil den sammenligne k med $x.key$. Hvis k er mindre, vil verdien ligge i venstre deltre til x og derfor kalles metoden rekursivt på $x.left$. Hvis k er større, vil verdien ligge i høyre deltre til x og derfor kalles metoden rekursivt på $x.right$. Dette gjentas helt til if-setningen ved linje 1 blir utført som følger av at vi går tom for noder eller vi finner en node med $key = k$. **Kjøretiden til TREE-SEARCH er $O(h)$**, fordi nodene som undersøkes vil danne en enkel bane nedover fra roten.



For å søke etter $k = 13$ på figuren vil algoritmen følge banen 15 → 6 → 7 → 13 og den returnerer denne noden.

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

TREE-MINIMUM – finner node med minste *key*-verdi

Denne metoden finner noden med minste *key*-verdi ved å følge venstre barnepeker helt til vi møter NIL. Metoden returnerer en peker mot denne noden. Binær-søketre egenskapen garanterer at dette vil være det minste elementet. **Kjøretiden til TREE-MINIMUM er $O(h)$**, fordi nodene som undersøkes vil danne en enkel bane nedover fra roten.

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

TREE-MAXIMUM – finner node med største *key*-verdi

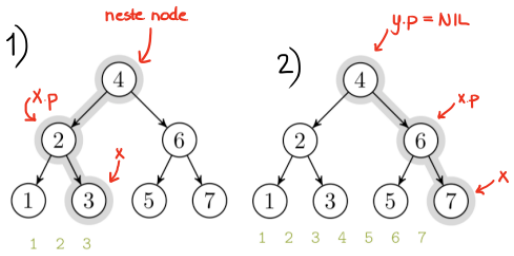
Denne metoden finner noden med største *key*-verdi ved å følge høyre barnepeker helt til vi møter NIL (dvs. motsatt av TREE-MINIMUM). Metoden returnerer en peker mot denne noden. Binær-søketre egenskapen garanterer at dette vil være det største elementet. **Kjøretiden til TREE-MINIMUM er $O(h)$**, fordi nodene som undersøkes vil danne en enkel bane nedover fra roten.

Neste metode er TREE-SUCCESSOR som vil finne neste element i den sorterte rekkefølgen som INORDER-TREE-WALK vil lage. Husk at denne prosessen vil printe ut høyre subtre etter x og vil da starte med de minste verdiene i høyre subtre, slik at verdiene returneres i sortert rekkefølge. Strukturen til et binært søketre gjør altså at vi slipper å sammenligne *key*-verdier for å finne etterfølgeren til x.

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

TREE-SUCCESSOR – finner neste node i sortert rekkefølge

Denne metoden vil finne etterfølgeren i den sorterte rekkefølgen fra INORDER-TREE-WALK ved å bruke strukturen til et binært søketre. Det er to tilfeller: (1) høyre subtre til x er ikke tomt, slik at etterfølgeren vil være det minste elementet i dette subtreet. Vi finner og returnerer dette elementet vha TREE-MINIMUM.



(2) Høyre subtre til x er tomt, altså x er det største element i deltreet, slik at neste node vil være rota til $x.p$ eller NIL (se figur). Når vi ser på det største elementet i et deltre, betyr det at vi har sett på alle elementene i dette deltreet. Dersom det er et venstre deltre (figur 1) vil det fortsatt være flere elementer igjen av hele treet, mens dersom det er et høyre deltre (figur 2) vil alle elementene ha blitt printet. Vi lar $y = x.p$ og entrer while-løkken. Denne løkken vil fortsette å se på foreldrenoden helt til $y = NIL$ (dvs. vi har nådd $T.root$) eller $x = y.left$ (dvs. x er i venstre deltre og vi har nådd roten til dette deltreet). Metoden vil dermed returnere NIL eller roten til deltreet.

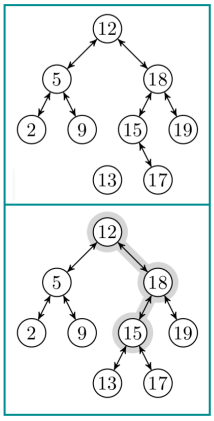
Kjøretiden til TREE-SUCCESSOR er $O(h)$, siden vi enten vil følge en enkel bane nedover fra roten eller oppover mot roten. **Prosedyren TREE-PREDECESSOR er symmetrisk med TRE-SUCCESSOR og har også kjøretid $O(h)$.**

12.3 Innsetting og fjerning

Innsetting og fjerning vil føre til at det binære søketreet endres, fordi datastrukturen må modifiseres slik at binært-søketre egenskapen fortsatt gjelder.

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == NIL
10      T.root = z          // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

<u>TREE-INSERT – vil sette inn en node i et binært søketre</u>
Denne metoden vil ta inn et binært søketre T og en node z som den vil sette inn ved rett plass i T bestemt av $z.key$. Metoden begynner ved roten til treet og bruker while-løkken for å finne en bane nedover på jakt etter en NIL som kan erstattes med z. For hver iterasjon vil y settes lik x, slik at den representerer forrige node. Hvis $z.key$ er mindre enn $x.key$ vil vi gå inn i venstre deltre, mens hvis $z.key$ er større vil vi gå inn i høyre deltre. Dermed følger vi en bane avhengig av *key*-verdien til z. Dette vil fortsette helt til $x = NIL$, slik at y er en node som ikke har noen barnenoder. Deretter lar vi y være foreldrenoden til z. Dersom treet var tomt vil vi sette roten lik z. Hvis ikke må vi avgjøre om z er venstre eller høyre barn til y ved å sammenligne *key*-verdiene. Dermed vil z være plassert på en slik måte at binært-søketre egenskapen er oppfylt. **Kjøretiden til TREE-INSERT er $O(h)$**, fordi det dannes en enkel bane nedover fra roten.

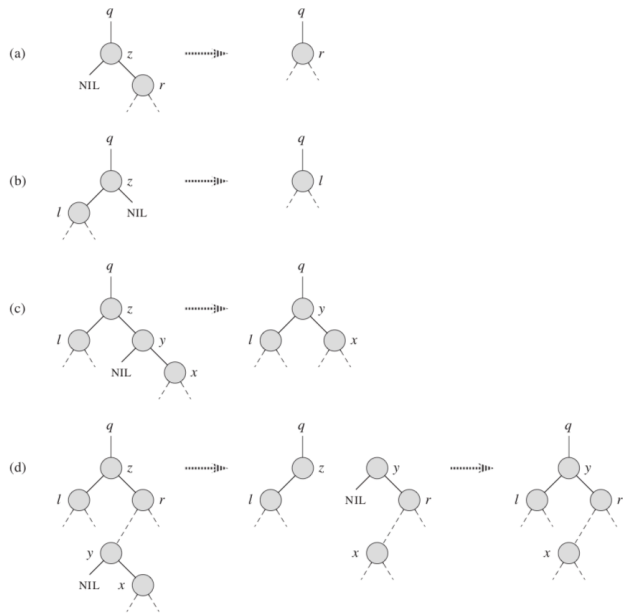


Figuren viser at vi setter inn z med $z.key = 13$. Siden denne er større enn 12 vil vi bevege oss inn i høyre deltre, og siden den er mindre enn 18 vil vi bevege oss inn i venstre deltre. Siden den er mindre enn 15 vil vi bevege oss til venstre, slik at $x = NIL$ og dermed $y.key = 15$. Siden $z.key = 13 < 15$ vil z bli venstre barn til y. Pilene indikerer at vi også har foreldrepekere.

Det kreves ikke grundig forståelse av slette-prosedyren hos binære søketrær, så vi ser kun på et overblikk. TREE-DELETE blir mer komplisert fordi dersom du fjerner z fra det binære søketreet er du nødt til å oppdatere barnenode pekerne til $z.p$ og foreldrenode pekerne til $z.left$ og $z.right$ dersom z har barnenoder. Dette kan altså kreve at vi flytter hele deltrær, noe vi gjør vha en separat algoritme kalt TRANSPLANT. Denne metoden vil kunne erstatte deltreet med rot u med deltreet med rot v. Transplantering vil kun fikse ting som har med foreldrenoden å gjøre, så TREE-DELETE må håndtere barna.

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

```
TREE-DELETE(T, z)
1  if z.left == NIL
2      TRANSPLANT(T, z, z.right)
3  elseif z.right == NIL
4      TRANSPLANT(T, z, z.left)
5  else y = TREE-MINIMUM(z.right)
6      if y.p ≠ z
7          TRANSPLANT(T, y, y.right)
8          y.right = z.right
9          y.right.p = y
10     TRANSPLANT(T, z, y)
11     y.left = z.left
12     y.left.p = y
```



Her kan vi se algoritmen for TRANSPLANT og TREE-DELETE og noen av de ulike tilfellene. **Kjøretiden til TREE-MINIMUM er $O(h)$**, fordi alle linjene i TREE-DELETE bruker konstant tid, bortsett fra kallet til TREE-MINIMUM som bruker tiden $O(h)$.

Algoritme	Kjøretid
INORDER-TREE-WALK	$\Theta(n)$
TREE-SEARCH	$O(h)$
TREE-MINIMUM	$O(h)$
TREE-SUCCESSOR	$O(h)$
TREE-INSERT	$O(h)$
TREE-DELETE	$O(h)$

Som vi kan se på tabellen vil alle operasjoner på et binært søketre ha kjøretid $O(h)$, bortsett fra INORDER-TREE-WALK som har kjøretid $\Theta(n)$.

12.4 Tilfeldig bygde binære søketrær – forelesning Ikke pensum

Vi har sett at operasjonene på binære søketrær avhenger av høyden til treet, men denne høyden vil variere. Average høyde til et binært søketre er lite kjent når innsetting og sletting brukes for å lage treet. **Dersom vi ser kun på innsetting vil vi få at forventet høyde til et tilfeldig bygd binært søketre vil være $O(\lg n)$**, slik at worst-case høyde blir lineær! Dermed vil kjøretiden til operasjonene bli $O(\lg n)$. Dersom dette hadde vært mulig hadde vi altså brutt grensen på sorteringshastigheten! Tilfeldig bygd vil si at *key*-verdiene blir satt inn i tilfeldig rekkefølge i et initialt tomt tre og alle input-permutasjonene ($n!$) er like sannsynlig.

Vi bruker ofte heap istedenfor binær søketrær fordi en heap er alltid perfekt balansert og operasjonene er alltid $O(n \lg n)$, mens et binært søketre vil avhenge av balansering, innsetting og sletting, som ofte tar lengre tid. Heap brukes når vi vil finne min og maks, mens binær søketrær er bedre når du ønsker at alt skal være sortert.

Forelesning 6 – Dynamisk programmering

Dynamisk programmering er på en måte en generalisering av splitt og hersk prosedyren, der delproblemer kan overlappe. I stedet for et tre av delproblem-avhengigheter, har vi en rettet asyklisk graf. Vi finner og lagrer del-løsninger i en rekkefølge som stemmer med avhengighetene. Læringsmålene er:

- **Forstå ideen om en delproblemgraf**
- **Forstå designmetoden dynamisk programmering**
- **Forstå løsning ved memoisering (top-down)**
- Forstå løsning ved iterasjon (bottom-up)
- Forstå hvordan man rekonstruerer en løsning fra lagrede beslutninger
- Forstå hva optimal delstruktur er
- Forstå hva overlappende delproblemer er
- Forstå eksemplene stavkutting og LCS
- Forstå løsningen på 0-1-ryggsekkproblemet (appendiks D, KNAPSACK, KNAPSACK').

Kapittel 15 – Dynamisk programmering

Dynamisk programmering vil i likhet med splitt-og-hersk metoder løse problemer ved å kombinere løsninger på delproblemer. Forskjellen er at dynamisk programmering brukes når **delproblemene overlapper**, altså når delproblemer har samme deldelproblemer. I dette tilfellet vil splitt-og-hersk metoden gjøre mer arbeid enn nødvendig, fordi den vil løse de delte deldelproblemene flere ganger. En dynamisk programmeringsalgoritme vil løse hvert deldelproblem en gang og lagrer svaret i en tabell, slik at den unngår å må gjenta arbeidet for å løse identiske deldelproblem.

Dynamisk programmering blir ofte brukt på **optimaliseringsproblem**, som kan ha mange ulike løsninger med ulike verdier, der vi vil finne løsningen med optimal verdi (maks eller min). Dette kalles en optimal løsning, og det kan være flere for et problem. Når vi skal utvikle en dynamisk programmeringsalgoritme, følger vi disse stegene:

1. **Karakteriser strukturen til en optimal løsning**
2. **Rekursivt definer verdien til en optimal løsning**
3. **Regn ut verdien til en optimal løsning, vanligvis bottom-up**
4. **Lag en optimal løsning fra den utregnede informasjonen.**

Hvis vi kun trenger verdien til en optimal løsning og ikke løsningen, kan steg 4 utelates.

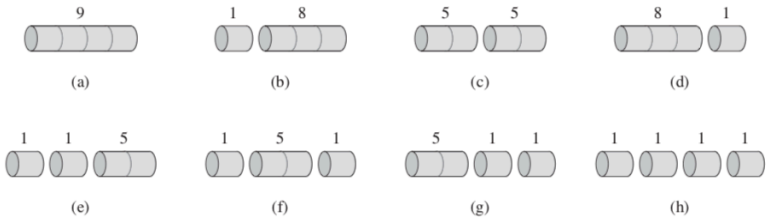
15.1 Stavkutting – eksempel på dynamisk programmering

Vi kan bruke dynamisk programmering for å bestemme den beste måten å kutte stålstenger. Dersom i er lengden på en stav, antar vi at prisen p_i for denne lengden er kjent. Tabellen til høyre viser noen av disse prisene.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Stavkutte-problemet

Gitt en stav med lengde n og en tabell med priser p_i for $i = 1, 2, \ldots, n$, bestem maksimum inntekt (r_n) som kan nås ved å kutte opp staven og selge delene. Merk at hvis p_n er høy nok, kan det hende at den optimale løsningen er å ikke kutte staven.



Figuren viser hvordan staven kan kuttes når $n = 4$. Her kan vi se at den optimale løsningen er å dele opp staven i to like store deler med lengde $i = 2$, fordi da får vi optimal inntekt på $r_4 = 10$.

Vi bruker betegnelsen $7 = 2 + 2 + 3$ for å indikere at en stav med lengde 7 har blitt delt inn i to deler av lengde 2 og én del av lengde 3. **En optimal løsning vil kutte staven inn i deler av lengde $i_1, i_2 \dots i_k$ der $k \leq n$, slik at summen av lendene blir $i_1 + i_2 + \cdots + i_k = n$ og totalprisen $r_n = p_{i_1} + p_{i_2} + \cdots p_{i_k}$ blir maksimert.** Vi er altså nødt til finne lengdene $i_1, i_2, \dots i_k$ som gjør at r_n blir maksimert. Dersom vi tar utgangspunkt i prisene på tabellen vil vi kunne finne løsningene som maksimerer r_i for $i = 1, 2, \dots 10$ vha observasjon:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- $r_1 = 1$ fra løsningen $n = 1$
- $r_2 = 5$ fra løsningen $n = 2$ (her er også $n = 1 + 1$ en løsning, men gir lavere inntekt)
- $r_3 = 8$ fra løsningen $n = 3$ (her er også $n = 2 + 1$ en løsning, men gir lavere inntekt)
- $r_4 = 10$ fra løsningen $n = 4 = 2 + 2$ (dette er løsningen som gir høyest inntekt)
- $r_5 = 13$ fra løsningen $n = 2 + 3$
- $r_6 = 17$ fra løsningen $n = 6$
- $r_7 = 18$ fra løsningen $n = 1 + 6$ eller $n = 2 + 2 + 3$
- $r_8 = 22$ fra løsningen $n = 2 + 6$
- $r_9 = 25$ fra løsningen $n = 3 + 6$
- $r_{10} = 30$ fra løsningen $n = 10$

For r_n vil det altså finnes flere løsninger, men den optimale løsningen vil maksimere r_n. En generell måte å finne denne løsningen er:

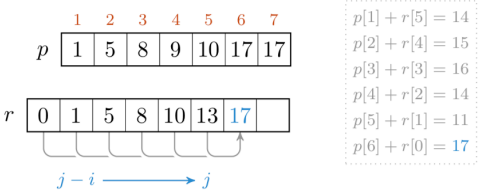
$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

Dvs. r_n vil være den største av inntektene som de ulike oppdelingene gir. Her vil p_n være inntekten når staven ikke deles opp, mens de andre argumentene er maksimum inntekt oppnådd når staven deles opp i to deler av størrelse i og $n - i$, og disse blir optimalt oppdelt videre for å oppnå inntekten r_i og r_{n-i}. Vi løser altså problemet av størrelse n ved å løse samme problem av mindre størrelse. Når vi kutter en stav i to deler, kan vi se på disse som to uavhengige instanser av stavkutte-problemet som derfor kan løses hver for seg. Derfor sier vi at stavkutte-problemet har **optimal delstruktur**, siden den optimale løsningen på problemet vil være gitt av kombinasjonen av de optimale løsningene på delproblemene. For eksempel finner vi $r_9 = \max(p_9, r_1 + r_8, \dots, r_3 + r_6, \dots r_8 + r_1) = \max(24, 21, \dots, 25, \dots, 21) = 25$, der vi har brukt at de optimale løsningene på $r_3 = p_3 = 8$ og $r_6 = p_6 = 17$.

En annen måte å finne maksimal inntekt, er ved å se på:

$$\boldsymbol{r_n = max_{1 \leq i \leq n}(p_i + r_{n-i})}$$

Her vil dekomponeringen bestå av å kutte av en venstre del med lengde i, slik at høyre del har lengde $n - i$. Det er kun høyre del som kan deles opp videre. r_n vil være den kombinasjonen av oppkuttinger som gir høyres pris. I eksempelet over vil $r_7 = p_2 + r_5 = p_2 + p_2 + r_3 = p_2 + p_2 + p_3 = 5 + 5 + 8 = 18$. Figuren viser hvorfor $r_6 = 17$. Her vil den optimale løsningen være gitt av den optimale løsningen til kun ett delproblem, nemlig resten r_{n-i}.



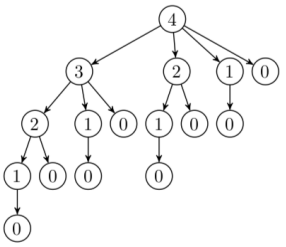
Rekursiv top-down implementasjon

Følgende metode finner maksimum inntekt ved stavkutting vha ligningen $r_n = max_{1 \le i \le n}(p_i + r_{n-i})$. Dette er en top-down rekursiv implementasjon:

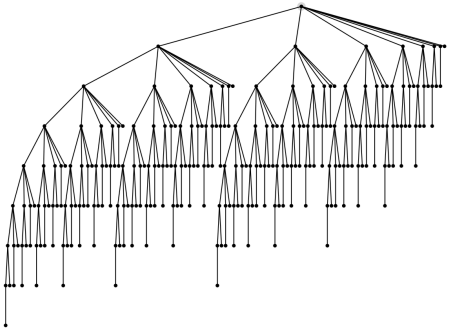
```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

CUT-ROD – kutter staven slik at inntekt maksimeres

Denne metoden bruker ligningen $r_n = max_{1 \le i \le n}(p_i + r_{n-i})$ for å finne maksimal inntekt q ved de gitte prisene i input arrayen $p[1 \ldots n]$ og lengden n. Dersom lengden er 0 vil fortjenesten være 0. Hvis ikke vil for-løkken prøve alle mulige oppdelinger. Den maksimale inntekten q vil endres dersom denne oppdelingen gir en større maksimal inntekt enn den største frem til nå. q initialiseres til $-\infty$ for å sikre at den blir endret ved første iterasjon. Vha det rekursive kallet vil $p[i] + CUT\text{-}ROD(p, n-i)$ bli den største inntekten som er mulig dersom du kutter av venstre del ved i, fordi $CUT\text{-}ROD(p, n-i)$ vil regne ut maksimal inntekt for høyre del av staven. Etter for-løkken vil derfor q være den maksimale inntekten for en stav med lengde n, og denne returneres.



CUT-ROD er svært ineffektiv, fordi den kaller på seg selv rekursivt med de samme parameter verdiene. Altså vil den løse samme delproblem flere ganger. Figuren viser tilfellet ved $n = 4$, og for hver av nodene må algoritmen regne ut maksimal inntekt (derfor forgrening under hver node > 0).



Kjøretiden til CUT-ROD vil være gitt av rekurrensen:

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

CUT-ROD undersøker alle 2^{n-1} måter å dele opp staven, så derfor er det logisk at kjøretiden blir 2^n

der 1 representerer kallet ved roten, mens $\sum_{j=0}^{n-1} T(j)$ er de rekursive kallene. Dersom vi løser denne vil vi få $T(n) = 2^n$, så kjøretiden vil være eksponentiell i n og vokser derfor veldig raskt. Figuren til venstre viser samme funksjon med $n = 8$!

Dynamisk programmering for optimal stavkutting

Vi skal omdanne CUT-ROD til en effektiv algoritme vha. dynamisk programmering. Etter å ha observert at en rekursiv løsning er ineffektiv, vil vi ordne det slik at hvert delproblem kun løses én gang ved at vi lagrer løsningen. Dersom vi trenger en løsning på nytt, kan vi slå den opp istedenfor å regne den på nytt. Dynamisk programmering vil derfor bruke ekstra minne for å spare beregningstid, og det kan spares svært mye tid (fra eksponentiell til polynomial)! Dynamisk programmering bruker altså en "time-memory tradeoff", så derfor vil det kun være nyttig hvis vi trenger noen løsninger mer enn én gang.

Det er to måter å implementere en dynamisk programmeringsalgoritme:

1. **Top-down med memoisering** – prosedyren lages rekursivt som før, men endres slik at den **lagrer resultatet til hver delproblem** (eks: i array eller hash tabell). Dersom prosedyren har løst delproblemet tidligere, vil den returnere den lagrede verdien istedenfor å regne den ut på nytt. Hvis ikke vil den regne ut løsningen på den vanlige måten. Den rekursive prosedyren har blitt memoisert, som vil si at den "husker" resultatet som den har regnet ut tidligere.

2. **Bottom-up metode** – løsningen av et delproblem avhenger kun av å **løse "mindre" delproblem**. Vi sorterer og løser delproblemene etter størrelse (minst først). Løsningen på hvert delproblem blir lagret, slik at hvert delproblem blir løst kun én gang. Når vi løser et bestemt delproblem vil vi allerede ha løst alle de mindre delproblemene som løsningen avhenger av og disse løsningene er lagret.

Disse to tilnærmingene vil gi algoritmer med samme asymptotiske kjøretid, men bottom-up metoden vil ofte ha bedre konstante faktorer.

Legg merke til at disse algoritmene bruker samme fremgangsmåte som vi har sett på tidligere (rekursjon/induksjon). Forskjellen er egentlig kun at løsningene mellomlagres!

Vi ser på disse to tilnærmingene for CUT-ROD:

Top-down med memoisering

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

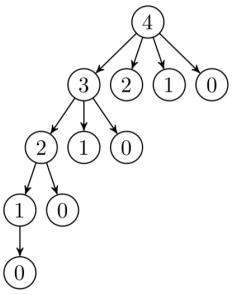
MEMOIZED-CUT-ROD – rekursiv løsning med lagring

Denne er identisk med CUT-ROD, bortsett fra at delløsningene blir lagret i arrayen $r[0..n]$. For-løkken vil sørge for at arrayen fylles med $-\infty$, som vil fungere som en markør på at delproblemene ikke er løst tidligere (hvis $r[i] \geq 0$ vil vi ha løst delproblemet før). Deretter kaler vi på MEMOIZED-CUT-ROD-AUX.

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

MEMOIZED-CUT-ROD-AUX – rekursiv løsning

Denne metoden vil finne den maksimerte inntekten ved kutting av stav med lengde n og prisene $p[1..n]$. Dersom problemet allerede er løst for n vil metoden returnere denne løsningen som er lagret ved $r[n]$. Dersom lengden er 0 vil inntekten være 0. Hvis ikke vil vi måtte finne løsningen som før, altså ved å gå igjennom alle oppdelingene og finne beste løsning ved å sammenligne med tidligere beste løsning. Legg merke til at den beste løsningen lagres ved $r[j]$. Dersom et senere rekursivt kall får samme input vil derfor første if-setning utløses, slik at vi slipper å utføre beregningen på nytt. Dermed vil vi finne $r[n]$ mye raskere, og denne returneres til slutt.



Figuren viser tilfellet ved $n = 4$, og dersom vi sammenligner denne med figuren på forrige side kan vi se at antall forgreininger er mye mindre i dette tilfellet. Grunnen til dette er at etter å ha funnet optimal løsning for node 2, vil algoritmen lagre løsningen, slik at neste gang den møter en node 2 vil den slippe å regne ut løsningen på nytt (dvs. slipper forgreiningen som følger av de rekursive kallene).

Bottom-up metode

Denne metoden er basert på iterasjon istedenfor rekursjon.

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = −∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```

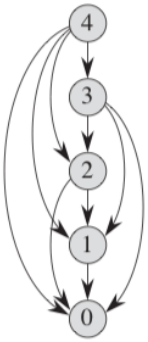
BOTTOM-UP-CUT-ROD – finner optimum fra bunnen og opp

Denne metoden vil finne den maksimerte inntekten ved kutting av stav med lengde n og prisene $p[1..n]$. Dette gjør den ved å bruke den naturlige rekkefølgen til delproblemene, altså at et delproblem av størrelse i er mindre enn et delproblem av størrelse j dersom $i < j$. Løsningene vil bli lagret i arrayen $r[0..n]$, der $r[i]$ vil være maksimal fortjeneste ved lengde i. Ved lengde 0 vil fortjenesten være 0, så derfor lagrer vi denne løsningen. For-løkken ved linje 3 vil bestemme venstre del av oppdelingen, mens for-løkken ved linje 5 vil dele opp denne venstre delen i mindre deler for å finne maksimal fortjeneste for denne. Legg merke til

at vi bruker $r[j-i]$ istedenfor et rekursivt kall, altså henter vi løsningen som er lagret istedenfor å regne ut løsningen på nytt. For eksempel for $j = 1$ og $i = 1$ vil $q = \max(-\infty, p[1] + r[0]) = \max(-\infty, 1 + 0) = 1$, der vi har hentet løsningen $r[0] = 0$. Denne løsningen blir lagret ved $r[1]$ slik at den kan brukes av senere iterasjoner. Dermed sikrer vi at delproblem blir løst kun én gang. De to for-løkkene vil løse delproblemene fra bunnen av siden indeksene begynner på 1 og går oppover. Til slutt vil vi finne $r[n]$ som så kan returneres.

Analyse av bottom-up og top-down

Både top-down og bottom-up metodene vil altså lagre løsningene på delproblemene. Top-down memoisering unngår å løse samme delproblem flere ganger ved å sjekke om løsningen er lagret før det rekursive kallet, mens bottom-up vil løse delproblemene fra bunnen av og deretter bruke det lagrede resultatet for å løse større delproblem.
Både bottom-up og top-down memoisering vil ha kjøretid $\boldsymbol{\Theta(n^2)}$, fordi bottom-up har dobbel for-løkke og top-down har n delproblem som løses vha n iterasjoner, men kun én gang hver. Dette er vesentlig mye bedre enn kjøretiden til CUT-DOWN på 2^n når n blir stor.



Delproblem graf Ikke forelesning

Grafen for delproblemet lar oss forstå settet av delproblem som er involvert i et dynamisk programmeringsproblem og hvordan disse delproblemene avhenger av hverandre. Størrelsen til delproblem grafen $G = (V, E)$ kan brukes for å bestemme kjøretiden til algoritmen. Hvert delproblem løses én gang og antall delproblem er lik antall vertekser i grafen. Tiden for å løse et delproblem vil som regel være proporsjonal med antall utgående kanter. Derfor vil kjøretiden være gitt av produktet mellom antall vertekser og antall utgående kanter. Figuren viser delproblem grafen for stavkutte problemet med $n = 4$. Her kan vi se at antall vertekser er n og antall kanter per verteks er maksimum n. Derfor vil kjøretiden til stavkutte-problemet være $O(n^2)$.

En rettet kant (x, y) indikerer at vi trenger en løsning på delproblem y for å løse delproblem x. Ved bottom-up vil delproblemene løses ved en reversert topologisk rekkefølge av delproblem grafen (dvs. fra bunnen og opp). Den vil ikke se på et delproblem før den har løst alle delproblemene som denne avhenger av. Top-down metoden kan ses på som et dybde-første søk (mer senere).

Rekonstruering av en løsning

Algoritmene vi har sett på vil returnere verdien til den optimale løsningen (q: maksimal inntekt), men de vil ikke returnere den faktiske løsningen, altså listen av lengder som staven kuttes opp i. For at dette skal være mulig må algoritmen i tillegg til å lagre den optimale verdien, også lagre valget som førte til denne.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] and s[0..n] be new arrays
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          if q < p[i] + r[j - i]
7              q = p[i] + r[j - i]
8              s[j] = i
9      r[j] = q
10 return r and s
```

EXTENDED-BOTTOM-UP-CUT-ROD – finner optimal løsning
Denne metoden vil finne den optimale inntekten *OG* de optimale lengden ved kutting av stav med lengde n og prisene $p[1..n]$. Prosedyren er identisk med BOTTOM-UP-CUT-ROD, bortsett fra at den lager arrayen $s[0..n]$ som skal lagre den optimale størrelsen i på første del som kuttes av når man løses delproblemet j. For eksempel for $j = 1$ vil $i = 1$ gi maksimal inntekt, så derfor vil $s[1] = 1$. Metoden vil returnere både r og s.

Figuren viser resultatet når $n = 10$. For eksempel har vi sett at $r[7] = 18$ for løsningen $n = 1 + 6$, slik at $s[7] = 1$

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Dynamisk programmering er **nyttig** når vi har overlappende delproblemer og **korrekt** når vi har optimal delstruktur!

15.3 Elementer ved dynamisk programmering

To krav for å kunne bruke dynamisk programmering er: (1) **optimal delstruktur** og (2) **overlappende delproblem**. Vi har sett at de fire stegene for å utvikle en dynamisk programmeringsalgoritme er:

1. Karakteriser strukturen til en optimal løsning – er det optimal delstruktur?
2. Rekursivt definer verdien til en optimal løsning
3. Regn ut verdien til en optimal løsning, vanligvis bottom-up
4. Lag en optimal løsning fra den utregnede informasjonen.

Vi skal nå se nærmere på de to kravene.

Optimal delstruktur

Første steg for å løse et problem med dynamisk programmering er å karakterisere strukturen til en optimal løsning. **Et problem har optimal delstruktur dersom en optimal løsning på problemet er basert på optimale løsninger på delproblemene.** Dersom problemet har optimal delstruktur kan det hende at dynamisk programmering kan brukes (eller grådig algoritme, mer senere). For å oppdage optimal delstruktur, kan vi følge dette mønsteret:

1. Vis at en løsning på problemet innebærer et valg, for eksempel å velge et initialt kutt på staven. Dette valget etterlater en eller flere delproblem som må løses.
2. Anta at for et gitt problem blir du gitt valget som fører til en optimal løsning
3. Gitt valget, bestem hvilke delproblem som følger og hvordan man kan best karakterisere det resulterende rommet av delproblem (prøv å holde rommet så enkelt som mulig og utvid hvis nødvendig, eks: rommet for stavkutte-problemet inneholdt problemer for å optimalt kutte en stav av lengde i).
4. Vis at løsningene på delproblemene innenfor en optimal løsning også må være optimale vha en "klipp-ut-lim-inn" teknikk. Dersom vi har en optimal løsning A, vil vi anta at den er basert på ikke-optimale løsninger på delproblemene. Deretter vil vi "klippe ut" de ikke optimale-løsningene og "lime inn" optimale løsninger. Dersom dette vil gi en bedre optimal løsning enn A, vil vi ha en motsigelse på at A er en optimal løsning. Derfor har vi vist at en optimal løsning A må være basert på optimale delløsninger.

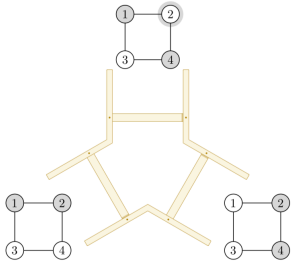
Merk: for å finne en optimal løsning vil $n = 4$ kreve kun ett delproblem av størrelse $4 - i$, men totalt sett vil det være fire delproblem

Variasjoner i optimal substruktur skyldes to faktorer:

- Totalt antall delproblemer
- Antall valg per delproblem

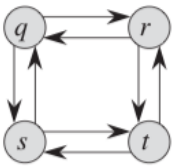
Kjøretiden til en dynamisk programmeringsalgoritme vil avhenge av produktet av disse to faktorene. I stavkutte-problemet hadde vi $\Theta(n)$ delproblemer og n valg for hvert delproblem. For eksempel for $n = 4$ vil vi ha 4 delproblem: $n = 3$, $n = 2$, $n = 1$ og $n = 0$ og for hver av disse har vi n muligheter for i. Dermed vil kjøretiden bli $O(n^2)$. Subproblem grafen gir en alternativ måte å utføre samme analyse (forrige side).

Når et problem har optimal delstruktur, kan det også hende at den optimale løsningen kan finnes vha en grådig algoritme. **En dynamisk programmeringsalgoritme vil finne optimale løsninger på delproblemene og deretter ta et informert valg, mens en grådig algoritme vil først utføre et grådig valg som ser best ut ved det tidspunktet og deretter løse resulterende delproblem**. Dermed slipper den å løse alle mulige delproblem, noe som fungerer i noen tilfeller!



Vi skal se på to problemer for en rettet graf $G = (V, E)$ og verteksene $u, v \in V$. Her vil en ha optimal delstruktur, mens den andre har ikke det:

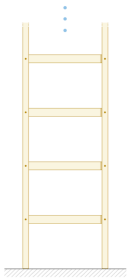
1. **Ikke-vektet korteste bane** – finn en bane fra u til v som består av færrest mulig kanter. En slik bane må være enkel, siden det å fjerne en syklus fra en bane vil gi en bane med færre kanter (med enkel mener vi at samme verteks ikke blir valgt flere ganger). Dette problemet har optimal delstruktur, som vi nå skal vise. Dersom $u \neq v$ vil enhver bane p fra u til v inneholde en mellomliggende verteks w (kan være u eller v). Dermed kan $u \overset{p}{\to} v$ deles inn i $u \overset{p_1}{\to} w$ og $w \overset{p_2}{\to} v$, og antall kanter i p vil være summen av antall kanter i p_1 og p_2. Vi påstår at dersom p er optimal bane (dvs. kortest) fra u til v, så vil p_1 være optimal bane fra u til w. Vi kan bevise dette vha et "klipp-ut-lim-inn" argument: vi antar at p er optimal og at p_1 er ikke-optimal. Det finnes en delbane p_1' som er optimal, dvs. kortere enn p_1. Dersom vi klipper ut p_1 og limer inn p_1' vil vi få banen $u \overset{p_1'}{\to} w \overset{p_2}{\to} v$, som er kortere enn p og dermed motsier at p er optimal. Derfor må p_1 være optimal når p er det. Det samme gjelder p_2. Derfor vil dette problemet ha optimal delstruktur, slik at vi kan finne korteste bane fra u til v ved å velge w og delbanene $u \to w$ og $w \to v$ slik at disse blir kortest mulig.



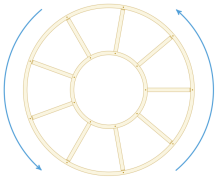
2. **Ikke-vektet lengste bane** – finn en enkel bane fra u til v som består av flest mulig kanter. Her må vi inkludere kravet "enkel" fordi ellers kan vi traversere en syklus så mange ganger vi vil for å lage baner med et vilkårlig stort antall kanter. Selv om dette ligner på problemet over, vil det ikke ha optimal delstruktur, noe vi ønsker å vise. Figuren til høyre viser et eksempel. Vi ser på banen $q \to r \to t$ som er lengste, enkle bane fra q til t. Her kan vi se at $q \to r$ vil ikke være lengste bane fra q til r, fordi det vil være $q \to s \to t \to r$. Altså, dersom p er en optimal bane (dvs. lengst) fra u til v, vil ikke nødvendigvis p_1 være en optimal delbane fra u til w. Årsaken til dette er at delproblemene for å finne lengste enkle bane **ikke er uavhengige.** Med dette mener vi at løsningen på et delproblem vil påvirke løsningen på et annet delproblem av det samme problemet. For eksempel kan problemet av å finne lengste enkle bane fra q til t deles inn i to delproblem: finne lengste enkle bane fra q til r og fra r til t. For det første delproblemet vil løsningen være $q \to s \to t \to r$. Dermed har vi valgt s og t, og kan ikke lenger bruke disse i løsningen på et andre delproblemet fordi da vil vi ikke få en enkel bane. Dermed kan vi ikke finne løsningen til problemet.

Merk: korteste-bane problemet vil ha uavhengige delproblem fordi ingen verteks annen enn w vil kunne ligge i begge delbanene (vil da ikke være korteste delbaner). Derfor vil dette problemet kunne ha optimal delstruktur.

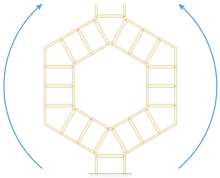
Velfunderthet *Ikke pensum , men forståelse på delproblem*



Enhver avhengighetskjede ender med et grunntilfelle. Vi kan godt bygge oss oppover i det uendelige, men ikke nedover! Dette fordi grunntilfellet kreves for at en rekursiv løsning skal terminere og at en iterativ løsning skal ha et sted å starte.



Sykliske avhengigheter (figur opp til høyre) vil også være en uendelig avhengighetskjede, som ikke har noe start. Derfor kan ikke delproblem være av denne formen. Vi kan ha forgreininger som samles (figur ned til høyre). Det viktige er å unngå sykliske avhengigheter (dvs. rettede sykler).

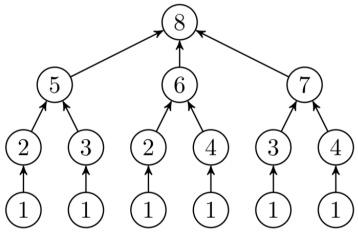


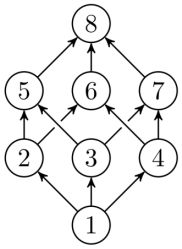
Overlappende delproblem

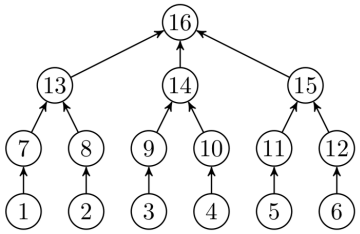
Det andre kravet for at dynamisk programmering skal kunne brukes er at rommet av delproblem må være "lite" slik at en rekursiv algoritme vil løse samme delproblem flere ganger, istedenfor å alltid generere nye delproblem. En grov tilnærming:

Vi bruker **dynamisk programmering** når den rekursive algoritmen besøker samme problem flere ganger, altså optimaliseringsproblemet har **overlappende delproblem**.

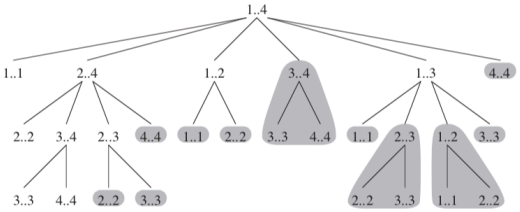
Vi bruker **splitt-og-hersk** når den rekursive algoritmen genererer nye delproblem ved hvert steg av rekursjonen







Dynamisk programmering utnytter overlappende delproblem ved å lagre løsningen, slik at den kan hentes fram ved konstant tid ved behov. Et vanlig tilfelle der vi har overlappende delproblem vil være når en algoritme gjentatt bruker løsningen på mindre delproblem for å finne løsningen på større delproblemer. På figuren kan vi se at verdiene fra lavere rader blir brukt for å løse høyere rader. De mørke grå feltene representerer utregningene som kan unngås ved å lagre løsningene underveis. **Dynamisk programmering kan øke effektiviteten dersom rekursjonstreet inneholder samme delproblem flere ganger og totalt antall ulike delproblem er lite.** Denne økningen kan være svært stor!



Rekonstruksjon av en optimal løsning

Dersom vi ønsker å finne den optimale løsningen må vi rekonstruere denne fra den optimale verdien vi har funnet, noe som kan ta lang tid. Vi kan derfor spare en signifikant mengde arbeid ved å lagre valget som ble gjort ved den optimale verdien i en array underveis, slik at vi slipper rekonstrueringen!

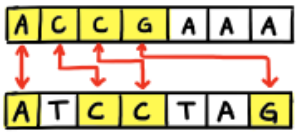
Memoisering

En alternativ tilnærming til dynamisk programmering er en top-down rekursiv strategi med memoisering, altså vil løsningene lagres underveis. En memoisert rekursiv algoritme vil ha en array som lagrer løsningene til hvert delproblem. Dersom algoritmen møter samme delproblem senere vil den søke opp løsningen istedenfor å regne den ut på nytt. Verdiene initialiseres til $\pm\infty$ for å markere om det har blitt funnet en løsning for et bestemt delproblem. **Dersom alle delproblem må løses en gang vil som regel bottom-up algoritmen være bedre enn top-down memoisering,** fordi den har lavere konstant faktor. **Hvis det er noen delproblem som ikke må løses vil det være bedre å bruke top-down memoisering**, fordi den vil kun løse delproblemene som det kreves løsning på.

Fire steg for å finne dynamisk programmeringsalgoritme:
1. Karakteriser strukturen til en optimal løsning
2. Rekursivt definer verdien til en optimal løsning
3. Regn ut verdien til en optimal løsning, vanligvis bottom-up
4. Lag en optimal løsning fra den utregnede informasjonen.

15.4 *Longest Common Subsequence* (LCS)

Et DNA kan representeres som en streng der hver base gis av forbokstaven: {A, G, C, T}. For eksempel kan DNAet til en organisme være $S_1 = ACCGAAA$, mens DNAet til en annen er $S_2 = ATCCTAG$. En måte å definere likheten mellom to DNA strenger er å finne en tredje DNA streng S_3 som er i både S_1 og S_2. Jo lengre S_3 er, desto større er likheten mellom S_1 og S_2. I vårt tilfelle vil $S_3 = ACCG$. Merk: **de like elementene trenger ikke å være på samme plass i S_1 og S_2, men rekkefølgen må være den samme (se figur)!**



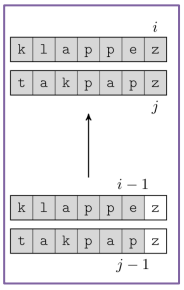
Vi definerer dette som **longest-common-subsequence (LCS) problemet**. Vi har gitt to sekvenser $X = \langle x_1, x_2, \dots x_m \rangle$ og $Y = \langle y_1, y_2, \dots y_n \rangle$ og ønsker å finne en felles subsekvens med maksimum lengde. Vi skal se hvordan de fire stegene (se boks opp til høyre) kan brukes for å lage en dynamisk programmeringsalgoritme s om løser LCS problemet.

Steg 1 – karakteriserer strukturen til LCS

LCS vil ha optimal delstruktur, som vi nå skal bevise. Gitt en sekvens $X = \langle x_1, x_2, \dots, x_m \rangle$, definerer vi $i$ende prefiks til X som $X_i = \langle x_1, x_2, \dots, x_i \rangle$. For eksempel hvis $X = \langle A, B, C, B, D, A, B \rangle$ vil $X_4 = \langle A, B, C, B \rangle$. Vi lar $Z = \langle z_1, z_2, \dots z_k \rangle$ være en LCS av X og Y. Da får vi følgende:

- **Hvis $x_m = y_n$, så må $z_k = x_m = y_n$ og Z_{k-1} er en LCS av X_{m-1} og Y_{n-1}.** Altså, hvis det siste elementet i X og Y er identisk, vil vi kunne fjerne disse fra settene og Z_{k-1} vil fortsatt være LCS. Bevis:
  - **$z_k = x_m = y_n$:** hvis $z \neq x_m$ kan vi feste $x_m = y_n$ på enden av Z for å få LCS med lengde $k + 1$. Dette er en motsigelse, siden Z er LCS og har lengde k.
  - **Z_{k-1} er fortsatt LCS**: vi har at Z_{k-1} er en prefiks med lengde $k - 1$. Vi antar at det eksisterer en felles subsekvens W med lengde større enn $k - 1$. Siden $x_m = y_n$ skal vi kunne feste disse på enden av denne subsekvensen, men da vil den få lengde $k + 1$, som er en motsigelse.
- **Hvis $x_m \neq y_n$, så vil $z_k \neq x_m$ bety at Z er en LCS av X_{m-1} og Y.** Altså hvis det siste elementet i X er ulik det siste elementet i Y og Z vil vi kunne fjerne dette og Z vil fortsatt være LCS. Bevis:
  - Hvis W er LCS for X_{m-1} og Y med lengde større enn k, må W også være LCS for X og Y siden $x_m \neq y_n$. Dette er en motsigelse fordi Z er LCS.
- **Hvis $x_m \neq y_n$, så vil $z_k \neq y_n$ bety at Z er en LCS av X og Y_{n-1}.** Altså hvis det siste elementet i Y er ulik det siste elementet i X og Z vil vi kunne fjerne dette og Z vil fortsatt være LCS. Bevis:
  - Hvis W er LCS for X og Y_{n-1} med lengde større enn k, må W også være LCS for X og Y siden $x_m \neq y_n$. Dette er en motsigelse fordi Z er LCS.

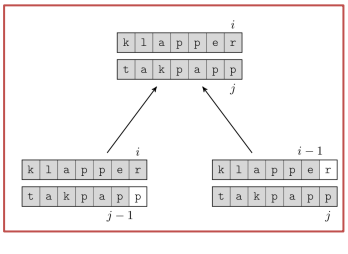
Dette gir oss at en LCS til to sekvenser inneholder en LCS av prefiksene til de to sekvensene. **LCS problemet har derfor en optimal delstruktur.**



Steg 2 – En rekursiv løsning for lengden til LCS

Tilfellene over gir at vi må enten undersøke et eller to delproblem:

- **Hvis $x_m = y_n$ trenger vi kun å løse ett delproblem**, nemlig å finne en LCS til X_{m-1} og Y_{n-1}, siden vi kan feste $x_m = y_n$ til LCS for å finne LCS til X og Y.
- **Hvis $x_m \neq y_n$ må vi løse to delproblem**, nemlig å finne en LCS til X_{m-1} og Y og finne en LCS til X og Y_{n-1} (vi vet ikke om $z_k \neq x_m$ eller om $z_k \neq y_n$). Den av disse som er lengst vil være LCS til X og Y.



Vi kan lett se at **disse delproblemene overlapper**. Fordi delproblemene av å finne LCS til X og Y_{n-1} og til X_{m-1} og Y vil inkludere deldelproblemer av å finne LCS til X_{m-1} og Y_{n-1}. Mange delproblemer vil altså ha samme deldelproblemer. Vi lar $c[i,j]$ være lengden til en LCS av sekvensene X_i og Y_j. **Den optimale delstrukturen til LCS problemet gir følgende rekursiv løsning:**

$$c[i,j] = \begin{cases} 0 & \text{hvis } i = 0 \text{ eller } j = 0 \\ c[i-1,j-1]+1 & \text{hvis } i,j > 0 \text{ og } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{hvis } i,j > 0 \text{ og } x_i \neq y_j \end{cases}$$

Dersom i eller j er 0, vil LCS være 0. Hvis $x_i = y_j$ må den rekursive formelen finne LCS til X_{i-1} og Y_{j-1} og legge til 1, siden vi kan feste $x_i = y_j$ til enden for å finne LCS til X og Y. Dersom i eller j er 0, vil LCS være 0. Dersom $x_i \neq y_j$ må vi løse de to delproblemene og sette LCS lik den største av disse.

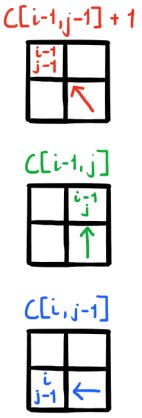
Steg 3 – regne ut lengden til LCS

Siden LCS problemet kun har $\Theta(mn)$ distinkte delproblem, kan vi bruke dynamisk programmering for å finne løsningen, altså lengden til LCS. Vi skal se på en bottom-up metode for å finne den optimale løsningen på LCS problemet.

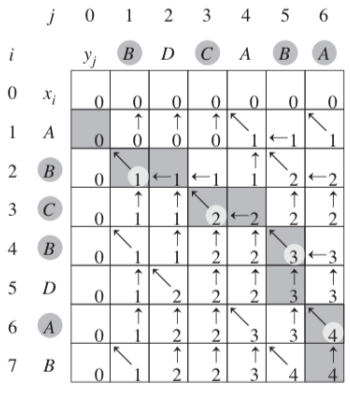
```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

LCS-LENGTH – finner lengden til LCS

Denne metoden vil finne den optimale løsningen på LCS problemet, altså lengden til LCS for to sekvenser X og Y. Metoden vil lagre lengden til delproblemene i tabellen c. Den vil også ha en tabell $b[1..m, 1..n]$ som vil hjelpe oss med å konstruere en optimal løsning, ved at $b[i,j]$ peker mot tabelloppføringen som ble brukt for å finne $c[i,j]$. Figuren til høyre viser de tre tilfellene for verdier i b.



Vi bruker en bottom-up metode, og må derfor begynne med å sette $c[i,0] = c[0,j] = 0$, altså dersom $X = 0$ eller $Y = 0$ vil lengden til LCS være 0. Deretter kan vi begynne å løse delproblemene fra bunnen av og opp. En dobbelt for-løkke brukes for å gå igjennom alle elementene i X og Y. For hver iterasjon vil vi se på $c[i-1,j-1] + 1$ dersom $x_i = y_j$ og $\max(c[i,j-1], c[i-1,j]$ dersom $x_i \neq y_j$. Legg merke til at vi lagrer lengden til LCS i c og vi lagrer pekeren i b. Til slutt vil vi nå $c[m,n]$, slik at vi finner lengden til LCS for X og Y. Metoden vil returnere c og b. Kjøretiden til metoden er $\Theta(mn)$, siden det tar $\Theta(1)$ tid å regne ut hver tabelloppføring og det er $m \cdot n$ oppføringer.



Figuren viser tabellen produsert av LCS-LENGTH på sekvensene X = $\langle A,B,C,B,D,A,B\rangle$ og Y = $\langle B,D,C,A,B,A\rangle$. Her kan vi se at LCS vil være $c[7,6] = 4$ som representerer $\langle B,C,B,A\rangle$.

Steg 4 – rekonstruere en LCS

Tabellen b lar oss raskt konstruere en LCS for X og Y, altså finne elementene i LCS og ikke bare lengden til LCS. Dette gjør vi ved å starte i $b[m,n]$ og deretter følge pilene (mørke grå område på figuren over). "↖" indikerer at $x_i = y_j$ er et element i LCS, "←"

indikerer at $x_i = y_{j-1}$ er et element i LCS og "↑" indikerer at $x_{i-1} = y_j$ er et element i LCS. Dermed vil vi finne elementene LCS for X og Y i revers rekkefølge.

Dermed har vi laget en dynamisk programmeringsalgoritme for å finne både lengden til LCS og elementene i LCS for to sekvenser X og Y. Denne har kjøretid $\Theta(mn)$, som er en betraktelig stor forbedring fra brute force som ville ha brukt eksponentiell tid!

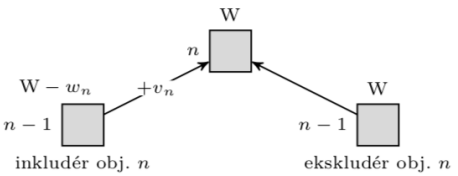
Appendix D – Ryggsekkproblemet

Det såkalte **ryggsekkproblemet** kommer i flere varianter, og vi skal se på to av dem i kapittel 16. Dette problemet går ut på at en tyv raner en butikk og finner n varer. i'ende vare er verdt v_i kroner og veier w_i gram. Tyven ønsker å ta så verdifull last som mulig, men ryggsekken hans klarer kun å bære W gram. Hvilke varer bør han ta? Den fraksjonelle varianten er lett å løse, fordi da tar man med seg så mye som mulig av den dyreste gjenstanden og fortsetter nedover på lista som er sortert etter kilopris. Det er derimot vanskeligere å løse 0-1 varianten, for da må man ta med seg en hel gjenstand eller la den ligge (0: la den ligge, 1: ta med). Siden løsningen er litt bortgjemt i teksten, skal vi nå se litt nærmere på den.

Dekomponeringen er basert på et ja-nei-spørsmål på formen "Skal vi ta med gjenstand i?". For hver av de to mulighetene sitter vi igjen med et delproblem som vi løser rekursivt. Som vanlig tenker vi at dette er siste trinn og antar at vi har gjenstander $1, 2, \ldots i$ tilgjengelig. Vi har da følgende alternativer:

- **Ja, vi tar med gjenstand *i***. Vi løser så problemet for gjenstander $1, 2, \ldots, i-1$, men kapasiteten til ryggsekken er redusert med w_i. Til slutt legger vi til v_i
- **Nei, vi tar ikke med gjenstand *i***. Vi løser så problemet for gjenstander $1, 2, \ldots, i-1$, men kan fortsatt bruke hele kapasiteten. Vi får ikke legge til v_i til slutt.

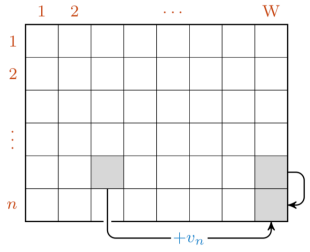
Figuren illustrerer situasjonen. Her vil hver rute representere en delløsning og pilene er avhengigheter. Vi skal nå se hvordan dette problemet kan løses rekursivt og vha dynamisk programmering (oppgave 16.2-2).



Rekursiv løsning av KNAPSACK

```
KNAPSACK(n, W)
1  if n == 0
2        return 0
3  x = KNAPSACK(n − 1, W)
4  if W < w_n
5        return x
6  else y = KNAPSACK(n − 1, W − w_n) + v_n
7        return max(x, y)
```

KNAPSACK – rekursiv løsning av 0-1 ryggsekkproblem
Denne metoden vil returnere den største verdien tyven kan ta. Dersom antall varer er 0, vil ikke tyven kunne ta noen varer. Hvis ikke lar vi x være den maksimale verdien vi får av de andre varene ved å ikke velge vare n. Dersom vekten til vare n er større enn vekten ryggsekken kan bære, vil vi velge vare x. Vi lar y være den maksimale verdien vi får av de andre varene når vi velger vare n. Til slutt vil vi returnere den største av x eller y. Merk: det rekursive kallet sørger for at vi går igjennom alle varene helt til vi når $n = 0$ eller til $w_i > W$. Linje 6 sørger for at verdiene for de ulike varene blir summert. **Kjøretiden vil være eksponentiell,** fordi antall rekursive kall vi vokse eksponentielt som følger av at det er to valg per kall.



Oppgave 6.2-3 Dynamisk programmering av KNAPSACK

For å redusere kjøretiden kan vi bruke dynamisk programmering ved å legge til memoisering eller skrive den om til en iterativ bottom-up løsning:

```
KNAPSACK(n, W)
 1  let K[0..n, 0..W] be a new array
 2  for j = 0 to W
 3      K[0, j] = 0
 4  for i = 1 to n
 5      for j = 0 to W
 6          x = K[i − 1, j]
 7          if j < w_i
 8              K[i, j] = x
 9          else y = K[i − 1, j − w_i] + v_i
10              K[i, j] = max(x, y)
```

KNAPSACK' – bottom-up løsning på ryggsekkproblem

Denne metoden vil returnere den største verdien tyven kan ta, ved å lagre delløsninger i en tabell K. $K[i,j]$ vil altså være maksimal verdi som kan oppnås for varene $1, 2, .. i$ ved kapasitet j. Siden det er en bottom-up metode, må vi begynne med å bestemme 0'verdiene. For hver kapasitet vil vi sette verdien lik 0 når det er ingen objekter. Deretter vil vi løse delproblemene fra bunnen og opp. En dobbelt for-løkke vil gå igjennom alle varene og alle kapasitetene fra bunnen av. x vil settes lik total verdi som kan oppnås når vare i ikke blir valgt. Dersom vekten til vare i er større enn kapasiteten, vil $K[i,j]$ settes lik x fordi vi har ikke kapasitet til å velge vare i. Hvis ikke vil y settes lik total verdi som kan oppnås når vare i blir valgt og $K[i,j]$ settes lik den største av x og y.

Figuren viser et eksempel på den resulterende tabellen K der kolonnene er kapasitetene og radene er antall varer. Her vil w og v være hhv vekten og verdiene til de n varene, der antall varer er $n = 6$ og total kapasitet er $W = 5$. Løsningen vil være i siste rute, dvs. $K[6,5] = 15$. For å rekonstruere den optimale løsningen, altså finne hvilke varer som er med i løsningen, kan vi bruke samme teknikk som i LCS (dvs. lagre valget som gjøres av $max(x,y)$ ved hver iterasjon).

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	1	1	1	1
2	0	1	5	6	6	6
3	0	4	5	9	10	10
4	0	4	5	9	10	10
5	0	4	6	9	11	12
6	0	4	6	10	12	15

w	v
1	1
2	5
1	4
3	3
1	2
2	6

Men, dette er ikke polynomisk

Selv om det virker som om vi har funnet en algoritme med polynomisk kjøretid, er det binære ryggsekkproblemet (0-1-knapsack) et såkalt NP-hardt problem. Det er ingen som har funnet noen polynomisk løsning på problemet! Kjøretiden til begge algoritmene er $\Theta(nW)$, siden det er nW delproblemer og vi utfører en konstant mengde arbeid per delproblem. Dette er en polynomisk funksjon av n og W, men det er ikke nok til å si at algoritmene har polynomisk kjøretid. Spørsmålet er "som funksjon av hva?". Hvis eksplisitt sier "som funksjon av n og W" er alt i orden, men om vi ikke oppgir noen eksplisitte parametre, så antas kjøretiden å være en funksjon av input-størrelsen (dvs. hvor stor plass en instans tar). Hvordan vi måler denne størrelsen avhenger av problemet, men når vi regner på om ting kan løses i polynomisk tid holder vi oss til antall bits i input. Dersom W har m antall bits, vil kjøretiden bli:

$$T(n,m) = \Theta(n2^m)$$

Som vi kan se vil ikke dette være en polynomisk kjøretid, men heller en eksponentiell. I stedet kaller vi kjøretiden for pseudopolynomisk, siden den blir polynomisk hvis vi lar et tall fra input (her: W) være med som parameter til kjøretiden: $\Theta(nW)$.

Forelesning 7 – Grådige algoritmer

Grådige algoritmer består av en serie med valg, og hvert valg tas lokalt. Algoritmen gjør alltid det som ser best ut her og nå, uten noe større perspektiv. Slike algoritmer er ofte enkle, men utfordringen ligger i å finne ut om de gir rett svar. Læringsmålene er:

- **Forstå designmetoden grådighet**
- **Forstå grådighetsegenskapen (*the greedy-choice property*)**
- Forstå eksemplene aktivitet-utvelgelse og det fraksjonelle ryggsekkproblemet
- Forstå HUFFMAN og Huffman-koder

Kapittel 16 – Grådige algoritmer

For mange optimaliseringsproblem vil dynamisk programmering være for mye, og det er bedre å bruke en enklere og mer effektiv algoritme. **En grådig algoritme vil gjøre valget som ser best ut ved det øyeblikket. Dvs. den tar et lokalt optimalt valg i håp om at dette fører til en global optimal løsning.** Grådige algoritmer vil gi optimale løsninger til mange problemer, men ikke alle! Bruksområder for grådig algoritmer inkluderer blant annet minimum-spenne-trær og Dijkstras algoritme (mer senere).

16.1 Aktivitet-utvelgelse problemet

Anta at vi har et sett $S = \{a_1, a_2, \dots, a_n\}$ med n aktiviteter som krever eksklusiv bruk av en felles ressurs, eks forelesningssal. Hver aktivitet a_i har en starttid s_i og en sluttid f_i, der $0 \leq s_1 < f_i < \infty$ og dermed et tidsintervallet $[s_i, f_i)$. To aktiviteter er kompatible dersom de har tidsintervall som ikke overlapper, altså for $[s_i, f_i)$ og $[s_j, f_j)$ vil a_i og a_j være kompatible dersom $s_i \geq f_j$ eller $s_j \geq f_i$. I aktivitet-utvelgelse problemet ønsker vi å velge et størst mulig subsett av gjensidig kompatible aktiviteter. Vi antar at aktivitetene er ordnet slik at sluttidene er i økende rekkefølge: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Denne tabellen viser ett eksempel på et sett av aktiviteter S. Vi skal se hvordan dette problemet kan løses med dynamisk programmering og med grådig algoritme.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Den optimale delstrukturen

Vi begynner med å vise at aktivitet-utvelgelse problemet har optimal delstruktur, altså at en optimal løsning på problemet er basert på optimale løsninger på delproblemene. Vi lar S_{ij} være settet av aktiviteter som starter etter aktivitet a_i og før aktivitet a_j. Vi tar utgangspunkt i S_{ij} og **antar at et maksimum sett av kompatible aktiviteter er A_{ij}** som inkluderer en aktivitet a_k. Ved å inkludere a_k vil vi få to delproblem:

1. Finne kompatible aktiviteter i S_{ik}, betegnet som A_{ik}
2. Finne kompatible aktiviteter i S_{kj}, betegnet som A_{kj}

A_{ik} inneholder aktivitetene fra A_{ij} som slutter før a_k starter, mens A_{kj} inneholder aktivitetene fra A_{ij} som starter etter a_k slutter. Altså vil $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, og **antall kompatible aktiviteter i S_{ij} vil være maksimalt $|A_{ij}| = |A_{ik}| + 1 + |A_{kj}|$.**

"Klipp-ut-lim-inn" teknikken kan brukes for å vise at dersom A_{ij} er en optimal løsning, må også A_{ik} og A_{jk} være optimale løsninger på de to delproblemene S_{ik} og S_{kj}. Dersom vi finner en bedre løsning for S_{kj} slik at $|A'_{kj}| > |A_{kj}|$, vil S_{ij} inneholde $|A_{ik}| + 1 + |A'_{kj}| > |A_{ik}| + 1 + |A_{kj}| = |A_{ij}|$ kompatible aktiviteter. Dette er en motsigelse på antagelsen om at A_{ij} er maksimum sett av kompatible aktiviteter. Samme argument kan brukes for S_{ik}. Hvis A_{ij} er en optimal løsning, må derfor A_{ik} og A_{kj} også være optimale løsninger, og vi kan si at **aktivitet-utvelgelses problemet har optimal delstruktur**.

Dermed kan vi bruke dynamisk programmering for å løse aktivitet-utvelgelses problemet. **Dersom $c[i,j]$ er verdien til en optimal løsning for settet S_{ij} (dvs. maks antall aktiviteter), vil vi få rekurrensen: $c[i,j] = c[i,k] + c[k,j] + 1$**. Denne er 0 dersom antall aktiviteter i S_{ij} er 0. Hvis ikke vil vi finne den største verdien mulig. Siden løsningen inkluderer valget av a_k er vi nødt til å undersøke løsningen ved alle verdiene a_k kan ha. Derfor vil:

$$c[i,j] = \begin{cases} 0 & S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} (c[i,k] + c[k,j] + 1) & S_{ij} \neq \emptyset \end{cases}$$

Videre kan vi bruke en rekursiv algoritme med memoisering eller bottom-up metode for å finne den optimale løsningen. Men da har vi ikke benyttet oss av en viktig egenskap ved dette problemet!

Å gjøre et grådig valg

Det grådige valget i aktivitet-utvelgelse problemet går ut på å legge til en aktivitet i den optimale løsningen før alle delproblemene er løst. Denne aktiviteten vil være den som har **tidligst sluttid**, slik at ressursen blir tilgjengelig for så mange andre aktiviteter som mulig. **Siden aktivitetene er sortert i økende orden etter sluttid, vil det grådige valget være aktivitet a_1** (det finnes også andre muligheter).

Dersom vi gjør det grådige valget vil vi stå igjen med ett delproblem som må løses, nemlig å **finne aktiviteter som starter etter a_1 er slutt**. Vi lar $S_k = \{a_i \in S \mid s_i \geq f_k\}$ være settet av aktiviteter som starter etter aktivitet a_k, slik at dersom vi tar det grådige valget vil S_1 være det eneste delproblemet som må løses ($a_k = a_1$). **Aktiviteten med tidligst sluttid (a_1) vil alltid være en del av en mulig optimal løsning** (bevis s. 418 i boka). Den optimal delstrukturen gir derfor at **den optimale løsningen til hele problemet vil bestå av a_1 og den optimale løsningen på delproblemet S_1**. Altså, hvis vi velger tidsintervallet som slutter først vil resten fortsatt løses optimalt.

I stedet for å bruke dynamisk programmering kan vi altså **velge aktiviteten som er ferdig først, kun ta vare på aktiviteter som er kompatible med denne og deretter gjenta prosessen helt til ingen aktiviteter er igjen.** En algoritme som løser aktivitets-utvelgelse problemet kan derfor arbeide top-down, ved å velge en aktivitet som den putter i den optimale løsningen og deretter løse delproblemet som går ut på å velge aktiviteter fra de som er kompatibel med de som allerede er valgt. **Grådige algoritmer har som regel top-down designet av å gjøre et valg og deretter løse et delproblem.**

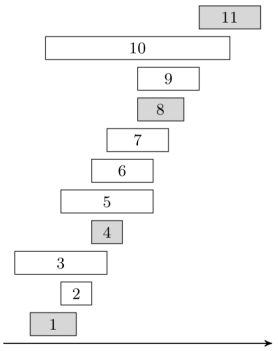
En rekursiv grådig algoritme

Vi skal se hvordan en rekursiv, grådig algoritmen kan brukes for å løse aktivitet-utvelgelse problemet:

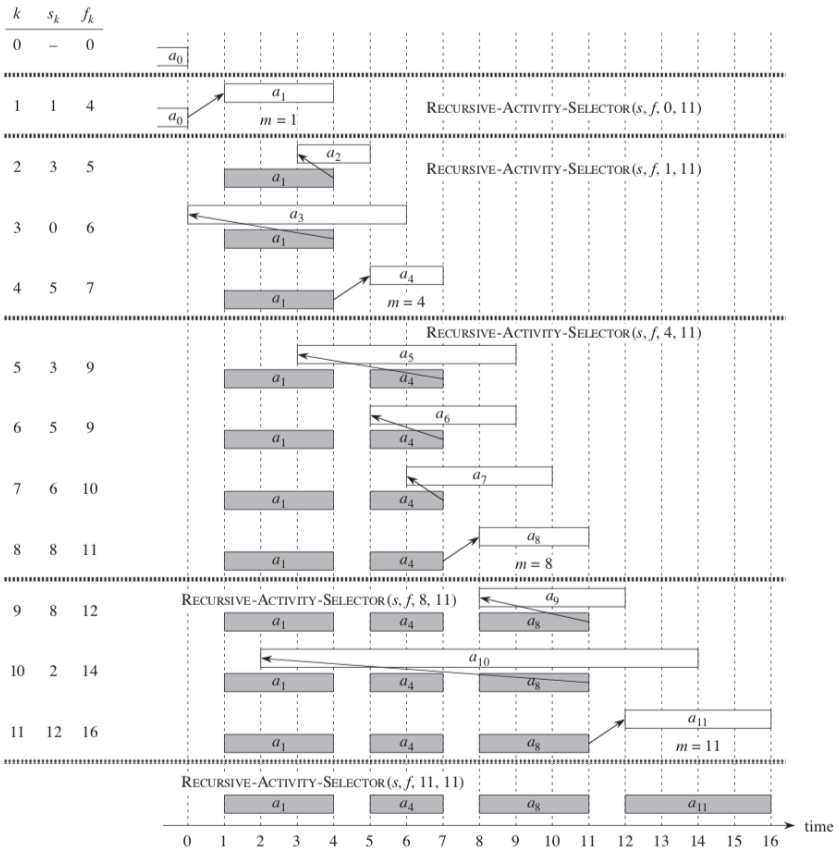
```
REC-ACT-SEL(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]
3        m = m + 1
4  if m ≤ n
5        S = REC-ACT-SEL(s, f, m, n)
6        return {a_m} ∪ S
7  else return ∅
```

REC-ACT-SEL – rekursiv, grådig løsning av aktivitet-utvelgelse

Denne metoden vil ta inn tidsintervaller $[s_1, f_1)$, … $[s_n, f_n)$, indeksen k som definerer delproblemet S_k og størrelsen n til det originale problemet. Delproblemet S_k går ut på å finne aktiviteter som starter etter a_k slutter. Metoden vil returnere det største mulige settet av kompatible aktiviteter, dvs. aktiviteter med ikke-overlappende tidsintervaller. Vi antar at tidsintervallene er sortert etter økende sluttid. For å løse hele problemet må vi bruke $k = 0$ og legger til en fiktiv aktivitet a_0 som har $f_0 = 0$ (slutter ved tidspunkt 0). m settes lik $k + 1$, slik at vi ser på aktiviteten som kommer etter a_k. Merk: dette er en grådig algoritme siden neste aktivitet vi velger vil være den som har tidligst sluttid. While-løkken vil se på neste aktivitet, så lenge det er flere aktiviteter igjen, helt til a_m starter etter a_k. Dermed er det to mulige utfall: (1) det er en aktivitet a_m som starter etter a_k eller (2) vi går tom for aktiviteter. Ved utfall 1 må vi finne resten av løsningen rekursivt og deretter kombinere dette med a_m (merk: union brukes for å kombinere to subsett). Ved utfall 2 må vi returnere et tomt sett, siden det er ingen aktiviteter som starter etter a_k. Det rekursive kallet vil altså fortsette helt til det ikke er flere aktiviteter igjen og $S = \emptyset$. Deretter fortsetter den oppover til det første kallet, slik at metoden vil returnere et sett med kompatible aktiviteter.



Kjøretiden er $\boldsymbol{\Theta(n)}$ dersom aktivitetene er sortert etter sluttid, siden hver aktivitet blir undersøkt én gang. Figuren viser et eksempel der metoden returnerer $\{a_1, a_4, a_8, a_{11}\}$. Den grådige algoritmen vil velge a_1, deretter neste aktivitet som starter etter a_1 slutter, osv. helt til den går tom for aktiviteter.



En iterativ grådig algoritme

Denne rekursive algoritmen kan omdannes til en iterativ algoritme, som er mye enklere:

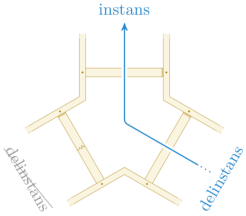
```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

GREEDY-ACTIVITY-SELECTOR – iterativ, grådig løsning

Denne metoden vil kun ta inn tidsintervaller $[s_1, f_1)$, … $[s_n, f_n)$, og returnere det største mulige settet av kompatible aktiviteter. Antall aktiviteter settes lik antall startverdier, og den grådige algoritmen begynner med å velge det første elementet a_1 som en del av den optimale løsningen. Deretter vil den gå igjennom alle aktivitetene, og dersom aktivitet a_m begynner etter aktivitet a_k som sist ble lagt til A, vil vi legge til a_m i den optimale løsningen. Tilslutt vil vi returnere arrayen A som vil inneholde det største mulige settet av kompatible aktiviteter. Merk: her har vi antatt at aktivitetene er ordnet etter økende sluttid, slik at f_k alltid vil være den største sluttiden i A. **Kjøretiden er $\boldsymbol{\Theta(n)}$**, dersom aktivitetene er ordnet etter økende sluttid.

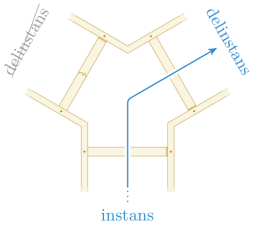
Bevis av resultat fra grådig algoritme

For å vise at den grådige algoritmen gir en optimal løsning kan vi vise at grådighet er minst like bra som alle andre algoritmer for hvert trinn, og dermed følger det at den gir en optimal løsning. Vi kan også bruke induksjon for å vise at siste sluttidspunkt i grådig aktivitetsutvalg er minst like tidlig som enhver annen løsning.



16.2 Elementer ved den grådige algoritmen

En grådig algoritme vil finne en optimal løsning på et problem ved å gjøre en sekvens av valg. Ved hvert avgjøringspunkt vil algoritmen gjøre valget som virker best i det øyeblikket. Altså, vi deler opp instansen i delinstanser og løser kun den mest lovende (figur til venstre). Alternativt kan vi se på "veien videre" i stedet for "veien hit", slik at den optimale løsningen kun vil bygge videre i den mest lovende retningen (figur til høyre). Disse to perspektivene er ekvivalente. En grådig algoritme vil ikke alltid gi en optimal løsning, men noen ganger vil det fungere. En grådig algoritme har følgende steg:



1. Vis at problemet har optimal delstruktur
2. Utvikle en rekursiv løsning
3. Vis at et grådig valg vil føre til at ett delproblem står igjen
4. Vis at det alltid er trygt å ta et grådig valg
5. Lag en rekursiv algoritme som implementerer den grådige strategien
6. Omform den rekursive algoritmen til en iterativ algoritme

På side 95 viste vi at aktivitet-utvelgelse problemet har optimal delstruktur ved å velge a_k slik at S_{ij} deles inn i to delproblemer S_{ik} og S_{kj}. Alternativt (med utgangspunkt i grådig algoritme) kunne vi ha vist at valget av a_k ville føre til ett delproblem S_k. Deretter kunne vi ha vist den optimale delstrukturen ved å vise at en optimal løsning på S_k er gitt av et grådig valg av a_m kombinert med en optimal løsning på settet S_m.

En grådig algoritme kan være designet etter følgende steg:

1. Skriv optimaliseringsproblemet på formen der vi gjør et valg og står igjen med ett delproblem som må løses
2. Vis at det alltid finnes en optimal løsning på det originale problemet som gjør det grådige valget, slik at det grådige valget er trygt.

3. Bevis den optimale delstrukturen ved å vise at det grådige valget vil etterlate et delproblem, slik at en optimal løsning på dette delproblemet kombinert med det grådige valget vil gi en optimal løsning på det originale problemet.

Vi må også vise at problemet har grådighetsegenskapen og optimal delstruktur, som vi nå skal se nærmere på.

Grådighetsegenskapen

Grådighetsegenskapen er at vi kan lage en global optimal løsning ved å gjøre lokale, optimale (grådige) valg. Det er altså at vi kan velge det som ser best ut her og nå uten å skyte oss i foten. Merk at denne egenskapen beskriver at det "første" grådige valget er trygt, og ikke om vi kan "fortsette" å være grådige! Ved dynamisk programmering vil vi gjøre et valg ved hvert steg og dette valget avhenger som regel av løsningene på delproblemene. I en grådig algoritme tar vi valget som virker best ved det øyeblikket og løser delproblemet som blir igjen. Dette valget kan avhenge av valg som har blitt gjort frem til nå, men kan ikke avhenge av noen fremtidige valg eller løsninger på delproblem. **Dynamisk programmering vil løse delproblem før det første valget blir gjort, mens grådige algoritmer vil gjøre det første valget før den løser delproblem.**

Det er viktig å bevise at et grådig valg ved hvert steg vil føre til en global optimal løsning. Dette blir vanligvis gjort ved å se på en global optimal løsning på et delproblem, og deretter vise at denne løsningen inkluderer et grådig valg og et mindre, lignende delproblem. For eksempel i aktivitet-utvelgelse så vi at delproblemet S_k har optimal løsning og at det grådige valget a_m (aktiviteten som har tidligst sluttid) vil være en del av en slik løsning. Dermed vil vi stå igjen med delproblemet S_m.

Optimal delstruktur

Et problem har optimal delstruktur dersom en optimal løsning på problemet inneholder optimale løsninger på delproblemene. Dette er et krav for både dynamisk programmering og grådige algoritmer. Ved grådige algoritmer må vi vise at en optimal løsning på delproblemet kombinert med det grådige valget allerede gjort, vil gi en optimal løsning på det originale problemet. Dermed kan vi bruke induksjon for å vise at det grådige valget ved hvert steg gir en optimal løsning.

Grådig vs dynamisk programmering – ryggsekkproblemet

Vi skal se på to klassiske optimaliseringsproblem for å illustrere forskjellen mellom grådig og dynamisk programmering. **Ryggsekkproblemet** (*knapsack*) går ut på at en tyv raner en butikk og finner n varer (se side 92). i'ende vare er verdt v_i kroner og veier w_i gram. Tyven ønsker å ta så verdifull last som mulig, men ryggsekken hans klarer kun å bære W gram. Hvilke varer bør han ta?

Det er to variasjoner av dette problemet: **0-1 knapsack** (tyven kan enten ta eller ikke ta varen) og **fraksjonell knapsack** (tyven kan ta fraksjoner av varer). Begge disse har optimal delstruktur og den optimale løsningen er den mest verdifulle lasten W. Ved 0-1 knapsack problemet vil vi fjerne element j, slik at den mest verdifulle lasten blir $W - w_j$ som tyven har valgt fra $n - 1$ elementer. Ved fraksjonsproblemet vil vi fjerne w fra vekten w_j til element j, slik at den mest verdifulle lasten blir $W - w$ som tyven har valg fra $n - 1$ elementer og gjenværende vekt $w_j - w$ av element j. **Problemene har**

optimal delstruktur siden dersom vi tar med noe, må resten av sekken fortsatt fylles optimalt.

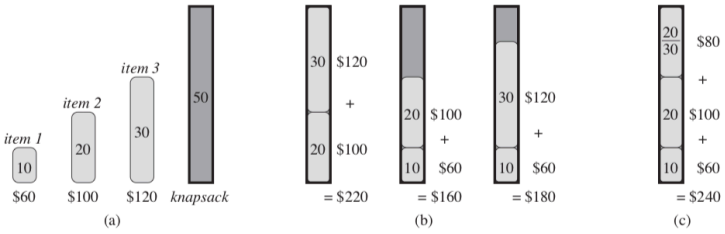
Grådig algoritme kan brukes for å løse det fraksjonelle ryggsekkproblemet, men ikke 0-1 problemet! For å løse fraksjonsproblemet vil tyven begynne med å ta så mye som mulig av elementet med størst verdi per vekt, v_i/w_i. Dersom det går tomt for dette elementet, vil han fortsette til neste element med størst verdi per vekt. Slik fortsetter prosedyren helt til han når vektgrensen W. Ved å sortere elementene etter verdi per vekt, vil denne grådige algoritmen ha kjøretid $O(n \lg n)$. **Fraksjonsproblemet har grådighets-egenskapen, fordi det finnes en optimal løsning der vi tar med mest mulig av det dyreste.**

Figuren under viser hvorfor en grådig algoritme ikke vil løse 0-1 knapsack problemet. Dette eksempelet har tre varer og en ryggsekk som kan bære $W = 50$ kg:

- Vare 1: 10 kg og 60 dollar, $v_1/w_1 = 6$ dollar/kg
- Vare 2: 20 kg og 100 dollar, $v_2/w_2 = 5$ dollar/kg
- Vare 3: 30 kg og 120 dollar, $v_3/w_3 = 4$ dollar/kg

OBS: en vare kan kun velges én gang!

Merk: en vare kan kun velges én gang! Siden verdien per vekt er størst for vare 1, vil den grådige algoritmen velge denne varen først. Dermed kan vi velge vare 2 eller vare 3, som begge innebærer at vekten blir så høy at den tredje varen ikke kan velges. Disse vil gi verdi hhv. 160 og 180 dollar. På figuren kan vi derimot se at den optimale løsningen er å velge vare 2 og vare 3 som gir verdi 220 dollar. **0-1 knapsack problemet vil ikke ha grådighetsegenskapen, siden den optimale løsningen ikke er gitt av å ta med mest mulig av det dyreste**. Legg merke til at det fraksjonelle problemet vil gi riktig svar med grådig algoritme (figur c).



Generelt vil ikke grådige algoritmer fungerer i situasjoner der vi er nødt til å sammenligne løsningen når elementet blir inkludert med løsningen til delproblemet når elementet blir ekskludert. Slike problem inkluderer mange overlappende delproblem, som er et kjennetegn til dynamisk programmering.

16.3 Huffman koder

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Huffman koder brukes for å komprimere data, som for eksempel kan være en sekvens av karakterer. Huffmans grådige algoritme bruker en tabell som gir hvor ofte hver karakter forekommer (dvs. frekvensen) for å lage en optimal måte å representere hver karakter som en binær streng. Anta at vi har en datafil på 100 000 karakterer som vi ønsker å lagre kompakt. Tabellen over gir frekvensen til de ulike karakterene. En måte å representere filen er vha **binær karakterkode**, der hver karakter representeres av en unik binær streng, som vi kaller **kodeord**. To typer:

- **Fast-lengde kode** – for å representere 6 karakterer trenger vi 3 bits, $a = 000$, $b = 001$, ... Denne metoden krever derfor 300 000 bits for å kode hele filen,
- **Variert-lengde kode** (optimal kode) – en karakter med høy frekvens får kort kodeord (dvs. få bits), mens en karakter med lav frekvens får lengre kodeord. Dersom vi ganger frekvensen med antall bits per kodeord, finner vi at koden krever:

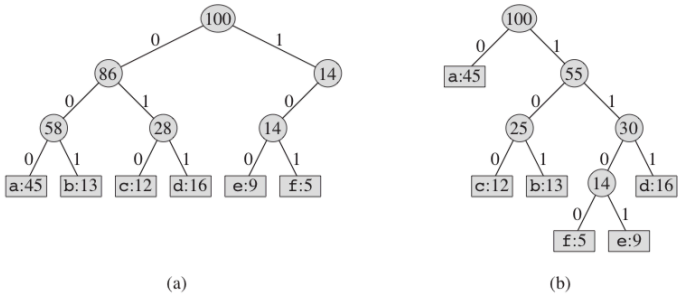
$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224\,400\ bits$$

Merk: vi ganger med 1000 siden frekvensen er *in thousands*

Prefikskoder

I en prefikskode vil ingen av kodeordene være prefiks for et annet kodeord. For eksempel vil $\{9, 55\}$ være en prefikskode, mens $\{9, 5, 55, 59\}$, vil ikke være en prefikskode siden 5 er prefiks for både 55 og 59. Legg merke til at hele kodeordet må være prefiksen, slik at $\{110, 1110\}$ vil være en prefikskode fordi 110 er ikke en prefiks til 1110. Prefikskoder vil forenkle dekodingen, som går ut på å oversette de binære strengene tilbake til karakterer. For eksempel vil $0101100 = 0 \cdot 101 \cdot 100 = abc$. Dersom kodeordene er prefikskoder vil det første kodeordet i filen vil være entydig, siden det ikke kan være prefiksen til et annet kodeord. **For å dekode filen kan vi derfor finne det første kodeordet, oversette det til en karakter og deretter gjenta prosessen for resten av filen**.

Kodeord kan representeres av et binærtre, der bladene er de gitte karakterene. **Vi finner kodeordet (eks: 101) til en karakter (eks: *b*) ved å følge banen nedover fra rota, når 0 betyr "velg venstre barn" og 1 betyr "velg høyre barn".** På figuren kan vi se binærtreet til fast-lengde kode (a) og optimal prefikskode (b). En optimal kode for en fil vil representeres av et fullstendig binærtre, så derfor vil ikke fast-lengde koden være optimal.



Dersom vi ser på et fullstendig binærtre, lar C være alfabetet som karakterene hentes fra og alle frekvensene er positive, vil den optimale prefikskoden ha nøyaktig $|C|$ blader (en for hver karakter) og nøyaktig $|C| - 1$ interne noder. Antall bits som trengs for å kode en fil er:

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

der c er en karakter fra C, $c.freq$ er frekvensen og d_T er dybden til bladet der c befinner seg, altså antall bits i kodeordet. $B(T)$ blir også kalt kostnaden til treet T.

Produksjon av Huffman koden

Huffman algoritmen er en grådig algoritme som lager en optimal prefikskode, som kalles en **Huffman kode**. Merk: en optimal prefikskode er et kodeord som representerer en karakter, har variabel-lengde og følger prefikskode egenskapen. Denne algoritmen vil bygge et binærtre, ved å begynne med et sett av $|C|$ blader og utføre $|C| - 1$ fusjoneringsoperasjoner.

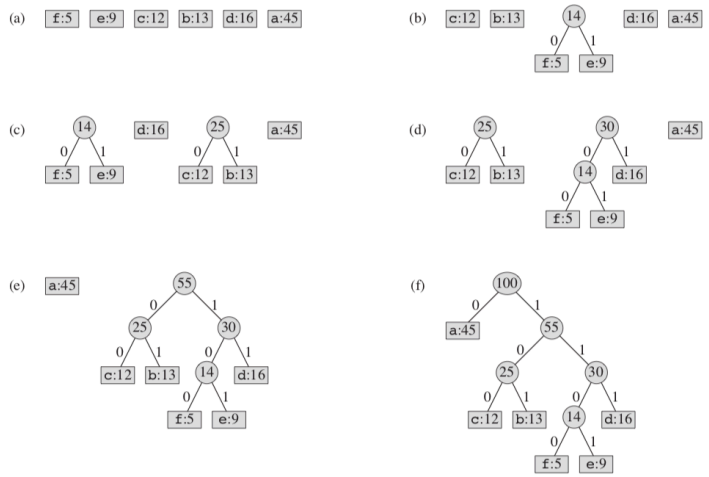
```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

HUFFMAN – vil lage en Huffman kode

Denne metoden tar inn C som er et sett med n karakterer, der hver karakter $c \in C$ er et objekt med egenskapen $c.freq$. Linje 2 vil initialisere min-prioritetskøen Q basert på frekvensen til objektene (minst frekvens først). For-løkken vil hente de to nodene x og y som har lavest frekvens og erstatte dem i køen med en ny node z som er foreldrenoden. Dette er det grådige valget (vi gjør objektene med lavest frekvens til barnenoder, slik at de kommer lengre ned i treet og får dermed lengre kodeord). Vi bruker metoden EXTRACT-

MIN for å hente ut venstre og høyre barn (merk: rekkefølgen er vilkårlig fordi det spiller ingen rolle om det er 110 eller 111 som representerer c, fordi de har samme lengde). Node z vil være et nytt objekt som skal legges til i prioritetskøen, så derfor finner vi frekvensen til z ved å summere frekvensen til x og y. Dermed kan z fusjoneres videre med de andre karakterene. Denne prosedyren blir gjentatt helt til vi kun har én node i Q, nemlig rota til treet som blir returnert. **Kjøretiden er $O(n \lg n)$**, siden det tar $O(n)$ tid å bygge en min-heap og for-løkken bruker tiden $O(n \lg n)$ ($n-1$ iterasjoner og $O(\lg n)$ per iterasjon).

Figuren viser et eksempel for karakterene $C = \{a, b, c, d, e, f\}$ med gitt frekvenser. Ved hvert steg vil det to objektene med lavest frekvens slås sammen til et nytt objekt med frekvens lik summen av de to frekvensene. Roten til treet vil derfor være summen av alle frekvensene, og det er denne som returneres. Denne gir at kodeordet til d er 111 ved å følge banen fra rota til bladet til d.



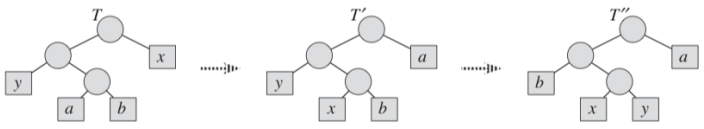
Korrekthet til Huffmans algoritme

For å vise at Huffmans grådige algoritme er riktig, må vi vise at problemet ved å bestemme optimal prefikskode har **grådighetsegenskapen** og **optimal delstruktur.**

Grådighetsegenskapen

Vi må vise at vi fortsatt kan få en optimal løsning dersom vi begynner med å slå sammen de to objektene med lavest frekvens. Dersom x og y er de to karakterene i C med lavest frekvens ($x.f$ og $y.f$), vil det eksistere en optimal prefikskode for C der kodeordene for x og y har samme lengde og den siste biten er eneste forskjell (dvs. de har samme foreldrenode z).

Vi vil bevise dette ved å ta utgangspunkt i et binærtre T som representerer en optimal prefikskode og modifisere denne slik at det blir en annen optimal prefikskode der x og y er søskennoder ved maksimum dybde. Dersom et slikt tre finnes har vi bevist utsagnet over. Vi lar a og b være to karakterer som er søsken ved maksimum dybde i T. Siden x og y har lavest frekvens vil $x.freq \leq a.freq$ og $y.freq \leq b.freq$. Som vi kan se på figuren vil vi lage treet T' ved å bytte a og x, og vi lager treet T'' ved å bytte b og y. I T'' er x og y søsken ved maksimum dybde, så det som gjenstår er å vise at dette fortatt er en optimal løsning. Formelen på forrige side, gir at forskjellen i kostnad blir:



$$B(T) - B(T') = \sum_{c \in C} c.f \cdot d_T(c) - \sum_{c \in C} c.f \cdot d_{T'}(c)$$

Vi bruker at eneste forskjell mellom T og T' er x og a. Videre kan vi bruke at $d_T(a) = d_{T'}(x)$ og $d_T(x) = d_{T'}(a)$

$$\begin{aligned}
&= x.f \cdot d_T(x) + a.f \cdot d_T(a) - x.f \cdot d_{T'}(x) - a.f \cdot d_{T'}(a) \\
&= x.f \cdot d_T(x) + a.f \cdot d_T(a) - x.f \cdot d_T(a) - a.f \cdot d_T(x) \\
&= (a.f - x.f)\big(d_T(a) - d_T(x)\big) \\
&\geq 0
\end{aligned}$$

Siden både $(a.f - x.f) \geq 0$ og $\big(d_T(a) - d_T(x)\big) \geq 0$. Altså vil kostnaden $B(T') \leq B(T)$.

Det samme gjelder for bytte av y og b, altså:

$$B(T'') \leq B(T)$$

Siden T er optimal, må dette bety at T'' også vil være en optimal prefikskode ($B(T'') = B(T)$). Dermed har vi bevist at **vi fortsatt kan få en optimal løsning dersom vi begynner med å slå sammen x og y som har lavest frekvens**. Vi har altså vist at problemet har grådighetsegenskapen, dvs. at det første grådige valget av å slå sammen objekter med lavest frekvens vil føre til en optimal prefikskode. Det første grådige valget er trygt!

Optimal delstruktur

Vi lar C' være alfabetet der x og y er fjernet og en ny karakter z er lagt til, slik at $C' = C - \{x, y\} \cup \{z\}$ (husk at union kombinerer subsett). Vi lar treet T' representere en optimal prefikskode for alfabetet C'. **Da vil treet T, som vi får fra T' ved å erstatte bladnoden z med en intern node som har x og y som barn, representere en optimal prefiks for alfabetet C.**

For å beviset dette utsagnet vil vi begynne med å finne et uttrykk for kostnaden $B(T)$ vha $B(T')$. Vi har at $d_T(x) = d_T(y) = d_{T'}(z) + 1$, siden dybden til x og y er 1 mer enn dybden til z. Derfor vil:

$$\begin{aligned} x.f \cdot d_T(x) + y.f \cdot d_T(y) &= (x.f + y.f)(d_{T'}(z) + 1) \\ &= z.f \cdot d_{T'}(z) + x.f + y.f \end{aligned}$$

der vi har brukt at $z.f = x.f + y.f$. Videre kan vi bruke at $B(T)$ og $B(T')$ vil inneholde de samme leddene bortsett fra at $B(T)$ inneholder x og y, mens $B(T')$ inneholder z. Derfor vil:

$$B(T) - B(T') = \big(x.f \cdot d_T(x) + y.f \cdot d_T(y)\big) - \big(z.f \cdot d_{T'}(z)\big) = x.f + y.f$$

Der vi har brukt resultatet fra formelen over. Altså har vi funnet at:

$$\begin{aligned} B(T) &= B(T') + x.f + y.f \\ B(T') &= B(T) - x.f - y.f \end{aligned}$$

Nå kan vi bevise utsagnet ved motsigelse. Anta at T ikke er en optimal prefikskode for C, altså vil det eksistere et optimalt tre $\mathcal{T}$ slik at $B(\mathcal{T}) < B(T)$. Vi lar $\mathcal{T}'$ være $\mathcal{T}$ der x og y er byttet ut med en felles foreldrenode z. Da vil formelen over gi at:

$$\begin{aligned} B(\mathcal{T}') &= B(\mathcal{T}) - x.f - y.f \\ &< B(T) - x.f - y.f = B(T') \end{aligned}$$

Altså betyr dette at $B(\mathcal{T}') < B(T')$, noe som er en motsigelse på antagelsen om at T' er en optimal prefikskode for C'. **Derfor må T være en optimal prefikskode for alfabetet C, og vi har vist at problemet har optimal delstruktur.**

Vi har vist at Huffmans algoritme har grådighetsegenskapen og optimal delstruktur, og dermed at den vil produsere en optimal prefikskode!

Forelesning 8 – Traversering av grafer

Vi traverserer en graf ved å besøke noder vi vet om. Vi vet i utgangspunktet bare om startnoden, men oppdager naboene til dem vi besøker. Traversering er viktig i seg selv, men danner også ryggraden til flere mer avanserte algoritmer. Læringsmålene er:

- Forstå hvordan grafer kan implementeres
- Forstå BFS, også for å finne korteste vei uten vekter
- Forstå DFS og parentesteoremet
- Forstå hvordan DFS klassifiserer kanter
- Forstå TOPOLOGICAL-SORT
- Forstå hvordan DFS kan implementeres med en stakk
- Forstå hva traverseringstrær (som bredde-først- og dybde-først-trær) er
- **Forstå traversering med vilkårlig prioritetskø**

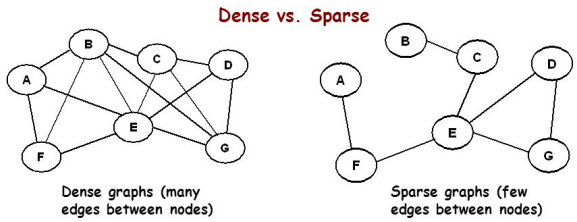
Kapittel 22 – Elementære grafalgoritmer

Vi skal se på hvordan grafer kan representeres og hvordan vi kan søke i en graf ved å systematisk følge kantene for å besøke verteksene i grafen.

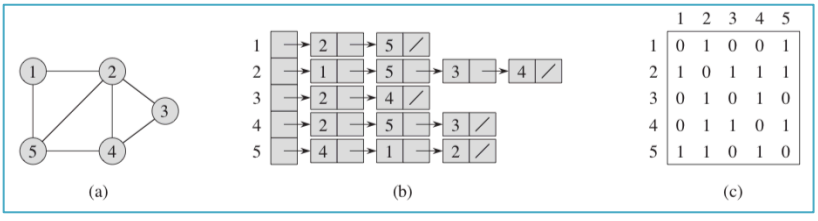
22.1 Representasjoner av grafer

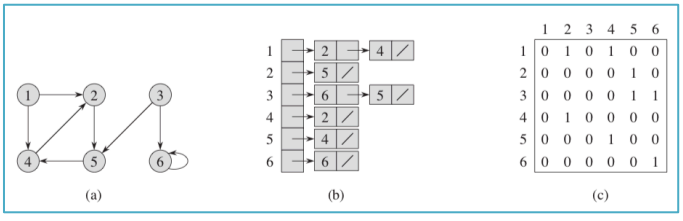
To mye brukte måter å representere en graf $G = (V, E)$ er:

- **Nabolister – ofte tregest, men tar mindre plass.** Brukes ofte for sparsomme grafer, altså når $|E| \ll |V|^2$ (dvs. antall kanter er mye mindre enn antall vertekser kvadrert = sparsom graf).
- **Nabomatriser – ofte raskest, men tar mer plass**. Brukes ofte for tette grafer, altså når $|E| \approx |V|^2$ (dvs. antall kanter er nær antall vertekser kvadrert = tett graf). Brukes også for å hurtig bestemme om det er en kant mellom to gitte vertekser.



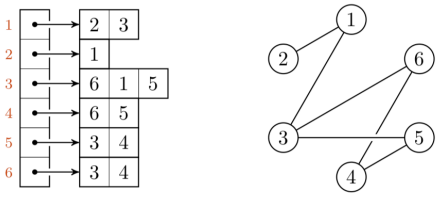
Figurene under viser en urettet og en rettet graf G (figur a) representert av en naboliste (figur b) og en nabomatrise (figur c)

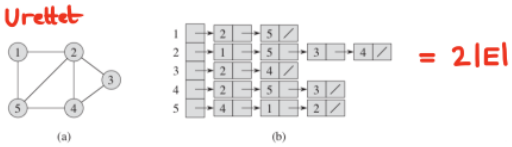


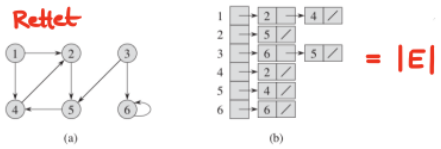


Nabolister – krever lite plass, men tregere

Naboliste representasjonen av grafen $G = (V, E)$ består av **en array av $|V|$ lenket lister**, altså en for hver verteks i V. **Listen til verteks u vil inneholde alle verteksene v, slik at det er en kant $(u, v) \in E$.** Listen til u vil altså inneholde alle naboverteksene til u.



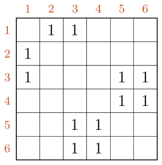


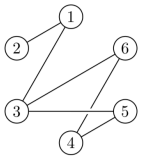


Dersom G er en urettet graf vil nabolisten til u inneholder alle vertekser som u er koblet til via en kant. Derfor vil summen av lengdene til alle nabolistene være $2|E|$, siden kanten (u, v) vil representeres av at v er i nabolisten til u, og omvendt. **Dersom G er en rettet graf vil nabolisten til u kun inneholde vertekser som kanten til u peker mot.** Derfor vil summen av lengdene til alle nabolistene være $|E|$, siden kanten (u, v) vil representeres av at v er i nabolisten til u (og ikke omvendt for rettet grafer). Nabolister kan også brukes for å representere **vektede grafer**, der vekten til en kant er gitt av vektfunksjonen w. Dette gjør vi ved å lagre vekten $w(u, v)$ til kant (u, v) med verteks v i nabolisten til u.

Fordelen med nabolister er at de er kompakte og krever kun $\Theta(V + E)$ minne, noe som gjør de egnet til traversering. En ulempe med nabolister er at de ikke gir noen raskere måte å bestemme om en kant (u, v) er tilstede i grafen, annet enn å søke etter v i nabolisten til u. Nabolister er derfor ikke så egnet til raske oppslag.

Nabomatriser – raskere, men krever mer plass

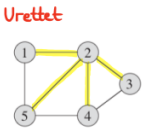


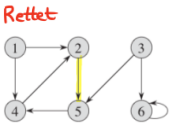


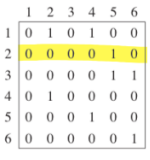
Nabomatrise representasjonen av grafen $G = (V, E)$ antar at verteksene er nummerert som 1, 2, ... $|V|$, og den består av en $|V| \times |V|$ matrise $A = (a_{ij})$, slik at:

$$a_{ij} = \begin{cases} 1 & \text{hvis } (i, j) \in E \\ 0 & \text{ellers} \end{cases}$$

Altså vil element a_{ij} i matrisen være 1 dersom det er en kant mellom verteks i og verteks j, hvis ikke vil den være 0.





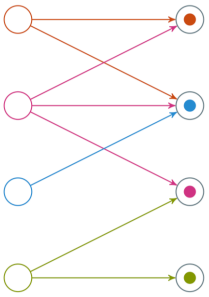


Dersom G er en urettet graf vil raden til u ha 1 for alle vertekser som u er koblet til via en kant. For eksempel vil verteks 2 ha 1 ved verteks 1, 3, 4 og 5. **For en urettet graf vil det være symmetri om hoveddiagonalen** siden $A[u, v] = A[v, u]$, og dermed vil $A = A^T$. **Dersom G er en rettet graf vil raden til u kun ha 1 for verteksene som kanten til u peker mot.** I dette tilfellet vil verteks 2 ha 1 ved kun verteks 5, siden dette er eneste verteks som 2 peker mot. Nabomatriser kan også brukes for å representere **vektede grafer**. Dette gjør vi ved å lagre vekten $w(u, v)$ til kant (u, v) som oppføringen ved rad u og kolonne v (dvs. vi bruker vekten istedenfor 1).

Ulempen med nabomatrisen er at den krever $\Theta(V^2)$ minne uansett antall kanter i grafen. Dette gjør at nabomatriser ikke er så egnet til traversering. Fordelen med nabomatriser er at de er enklere og egnet til raske oppslag.

Nabolister vs Nabomatriser

Nabomatriser egner seg til direkte oppslag, mens nabolister egner seg til traversering. Nabolister tar også mindre plass dersom grafen har få kanter, men ikke ellers!



Vi skal nå se på traversering som blant annet trengs for å matche flest mulig donorer med resipienter (se figur). **Traversering går generelt ut på å besøke noder, oppdage noder langs kanter og vedlikeholde en huskeliste.** Vi skal nå se på to ulike typer, nemlig BFS og DFS.

22.2 Bredde-først-søk (BFS)

BFS er en av de enkleste algoritmene for å søke en graf. **Gitt en graf $G = (V, E)$ og en kildeverteks s, vil bredde-først-søk systematisk undersøke alle kantene til G for å oppdage alle vertekser som kan nås fra s.** Den vil regne ut avstanden fra s, altså minimum antall kanter, og vil produsere et "**bredde-først tre**" med rot s, som inneholder alle verteksene som kan nås. I dette treet vil den enkle banen fra s til verteks v være den korteste banen, altså banen som inneholder minst antall kanter. **BFS virker på både rettede og urettede grafer.** Algoritmen kalles bredde-først fordi den vil oppdage alle vertekser ved avstanden k fra s før den oppdager noen av verteksene ved avstanden $k + 1$.

For å holde styr på fremgangen vil BFS fargelegge hver verteks hvit, grå eller svart. Hvit er starttilstanden, grå er oppdaget og svart er at alle nabonodene er oppdaget (grå). Alle vertekser vil være hvite i begynnelsen, og kildeverteksen er den første til å bli farget grå. Deretter vil alle nabonodene oppdages og farges grå. Siden alle nabonodene er grå vil kildeverteksen farges svart. Dersom en verteks er grå vil algoritmen forstå at den har noen nabonoder som enda ikke er oppdaget (dvs. som er hvite). Hvis verteksen er svart, vil den derimot vite at alle nabonodene er oppdaget.

BFS vil produsere et bredde-først tre som i begynnelsen kun inneholder roten. Dersom algoritmen følger kanten (u, v) ut fra u og oppdager en hvit verteks v, vil den legge verteksen og kanten til i treet. Vi kaller u for forgjengeren eller forelderen til v i bredde-først treet. En verteks vil kun oppdages en gang og har derfor kun en forelder. Forgjenger og etterfølger defineres relativt til roten, for eksempel hvis u er mellom roten og v, vil u være forgjengeren og v er etterfølgeren.

Vi har følgende egenskaper:

- $u.color$ – lagrer fargen til verteks u
- $u.\pi$ – lagrer forgjengeren til verteks u
- $u.d$ – lagrer avstanden fra kildeverteksen s til verteksen u

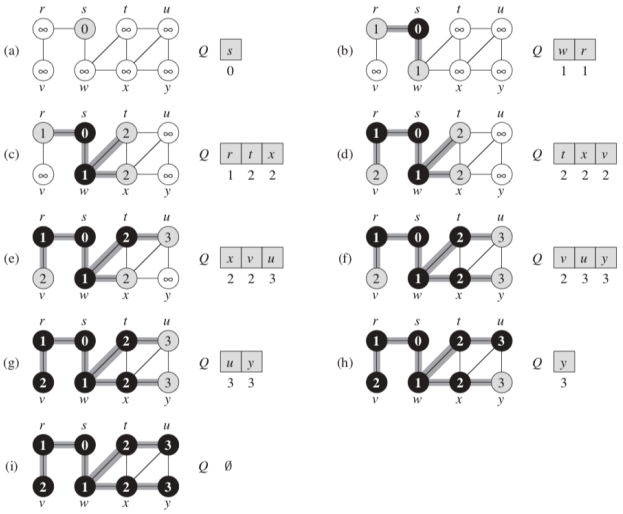
For å holde styr over alle verteksene som blir oppdaget, bruker BFS en first-in, first-out (FIFO) kø (s. 18).

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

<u>BFS – lager et bredde-først tre fra grafen G med rot s</u>

Denne metoden tar inn en graf G og en kildeverteks s, og vil lage et bredde-først tre med roten s. Metoden antar at G er gitt som naboliste, slik at hver verteks vil ha en liste av verteksene den er koblet til via en kant. For-løkken ved linje 1-4 vil farge alle verteksene hvite, bortsett fra kildeverteksen. Den vil også sette avstanden fra kildeverteksen lik uendelig og forgjengeren settes lik NIL. For kildeverteksen vil fargen settes lik grå, avstanden settes lik 0 og forgjengeren settes lik NIL. Ved linje 8 vil vi initialisere FIFO køen Q og ved linje 9 legger vi til kildeverteksen s. Denne køen vil inneholde grå vertekser, og while-løkken vil kjøre så lenge det er flere grå vertekser igjen. Ved linje 11 vil vi hente u fra køen, og dermed fjerne denne verteksen fra Q. Deretter vil vi se på verteksen v som er først i nabolisten til u. Dersom denne verteksen er hvit vil vi endre fargen til grå, vi vil sette avstanden lik $u.d + 1$ og vi vil sette forgjengeren lik u. Siden v har blitt en grå verteks, vil vi legge den til ved halen til Q. Deretter gjentar vi prosessen for alle verteksene i

nabolisten til u, og til slutt farger vi u svart, siden alle naboverteksene har blitt oppdaget (dvs. blitt farget grå). Deretter vil while-løkken gå til neste grå verteks, helt til alle grå verteksene er farget svarte. **Kjøretiden til BFS er $\boldsymbol{\Theta(V+E)}$,** siden alle noder farges grå og svarte, og alle kanter undersøkes. (Best-case: $\Theta(1)$).



Figuren viser et eksempel på BFS. Legg merke til at kildeverteksen er 0 og grå, mens resten av verteksene er ∞ og hvite ved begynnelsen. Ved første iterasjon vil naboverteksene oppdages (farges grå) og kildeverteksen farges svart. Legg merke til at de to verteksene blir lagt til i Q i vilkårlig rekkefølge, men det er verteksen som ble lagt til først som undersøkes i neste iterasjon. Derfor vil rekkefølgen for oppdagingen variere, men avstandene d vil bli de samme uansett.

Korteste bane

Korteste-bane avstanden $\delta(s,v)$ er minimum antall kanter i enhver bane fra s til v. Dersom det er ingen bane vil $\delta(s,v) = \infty$. **BFS vil regne ut denne korteste-bane avstanden, så lenge vi bruker en FIFO kø (ellers finner vi noder via omveier!).** Vi ønsker nå å vise at BFS vil regne ut $v.d = \delta(s,v)$ for hver verteks $v \in V$.

LEMMA 1:
Vi lar $G = (V,E)$ være en rettet eller urettet graf og s er en vilkårlig verteks. Da vil følgende gjelde for enhver kant $(u,v) \in E$:

$$\delta(s,v) \leq \delta(s,u) + 1$$

Dersom u kan nås fra s, kan også v nås fra s. Den korteste banen fra s til v kan derfor ikke være lengre enn den korteste banen fra s til u etterfulgt av kanten (u,v). Dersom u ikke kan nås vil $\delta(s,u) = \infty$ og ulikheten gjelder fortsatt.

Lemma 2:
Dersom BFS kjører fra kildeverteks s vil verdien $v.d$ som regnes ut tilfredsstille $v.d \geq \delta(s,v)$, dvs. $v.d$ er en øvre grense. Dette kan bevises vha induksjon:

- Grunntilfellet (før første iterasjon): $s.d = 0 = \delta(s,s)$ og $v.d = \infty \geq \delta(s,v)$, slik at $v.d \geq \delta(s,v)$ er oppfylt for alle $v \in V$
- Induktivt steg: den induktive hypotesen er at $u.d \geq \delta(s,u)$. Linje 15 i BFS algoritmen og formelen over gir at $v.d = u.d + 1 \geq \delta(s,u) + 1 \geq \delta(s,v)$. Denne avstanden vil ikke endres, så derfor vil den induktive hypotesen stemme.

Lemma 3:
Køen Q vil ved alle tidspunkt holde maksimum to ulike d verdier, noe som skyldes at køen er FIFO. Anta at Q inneholder verteksene $\langle v_1, v_2, \ldots, v_r \rangle$ der v_1 er ved hodet og v_r er ved halen. Da vil $v_r.d \leq v_1.d + 1$ (v_r er kun én kant lengre bort fra s) og $v_i.d \leq v_{i+1}.d$ (vertekser ved halen er lengre eller like langt bort fra kildeverteksen som vertekser ved hodet). Dette kan bevises ved induksjon. **Dersom v_i blir lagt til i køen før v_j vil $v_i.d \leq v_j.d$, og v_i vil fjernes først.**

Teorem – korrekthet ved BFS

Vi lar $G = (V, E)$ være en rettet eller urettet graf, og antar at BFS kjører fra en kildeverteks $s \in V$. I **løpet av utførelsen vil BFS oppdage alle vertekser $v \in V$ som kan nås fra s, og ved terminering vil $v.d = \delta(s, v)$ for alle $v \in V$**. For $v \neq s$ vil korteste bane fra s til v være korteste bane fra s til $v.\pi$ etterfulgt av kanten $(v.\pi, v)$. Dette kan **bevises med motsigelse**, ved at vi antar at verteks v med korteste-bane $\delta(s, v)$ får en feil avstand $v.d \neq \delta(s, v)$. Lemma 2 gir at $v.d \geq \delta(s, v)$ som i dette tilfellet blir $v.d > \delta(s, v)$. Lemma 1 gir at $\delta(s, v) = \delta(s, u) + 1$ og vi har at $u.d = \delta(s, u)$. Dermed får vi at:

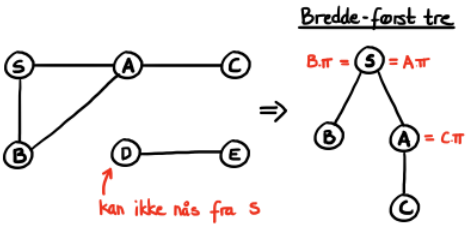
$$v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$$

Vi lar v være en verteks som blir undersøkt som følger av at det er en naboverteks til u. Vi har tre tilfeller for v, som alle motsier ulikheten:

- **v er hvit**, slik at linje 15 vil sette $v.d = u.d + 1$
- **v er svart**, slik at den har blitt fjernet før u og dermed vil Lemma 3 gi at $v.d \leq u.d$.
- **v er grå**, slik at den har blitt oppdaget fordi den er nabomatrisen til en verteks w som ble fjernet før u. Derfor vil linje 15 gi at $v.d = w.d + 1$ og Lemma 3 gir at $w.d \leq u.d$, slik at $v.d = w.d + 1 \leq u.d + 1$

Dermed kan vi konkludere med at $v.d = \delta(s, v)$ for alle $v \in V$.

Bredde-først tre



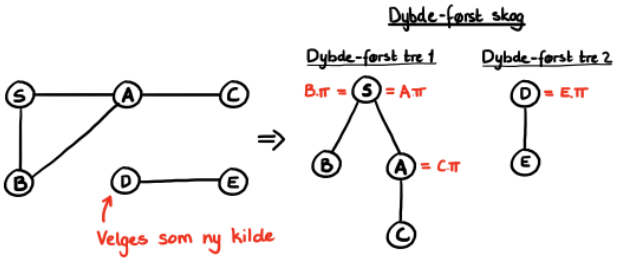
Bredde-først treet vil korrespondere til π egenskapene, altså forgjengerne. Forgjenger subgrafen til G defineres som $G_\pi = (V_\pi, E_\pi)$, der V_π er alle verteksene som har en forgjenger pluss kildeverteksen, mens E_π er kantene $(v.\pi, v)$. G_π vil være et bredde-først tre dersom V_π består av verteksene som kan nås fra s og G_π inneholder en unik, korteste bane fra s til v. Kantene E_π kalles trekantene.

22.3 Dybde-først-søk (DFS)

Strategien til DFS er å søke "dypere" i grafen når det er mulig. Dybde-først-søk vil undersøke kanter i den mest nylig oppdaget verteksen v som fortsatt har uoppdaget kanter. Når alle kantene til v har blitt oppdaget vil metoden gå tilbake til verteksen som v ble oppdaget fra. Denne prosessen fortsetter helt til vi har oppdaget alle verteksene som kan nås fra kildeverteksen. **Dersom det er igjen noen uoppdaget vertekser vil DFS velge en av de som ny kilde og deretter gjenta søket fra denne kilden**. Algoritmen vil gjenta hele prosessen til alle vertekser er oppdaget.

Dybde-først tre

Dersom DFS søker i nabolisten til verteks u og oppdager en verteks v, vil den sette forgjenger egenskapen $v.\pi = u$. Forgjenger subgrafen til BFS kan danne et tre, mens **forgjenger subgrafen til DFS kan bestå av flere trær, siden dette søket kan gjentas fra flere kilder.** Forgjenger subgrafen til DFS defineres som $G_\pi = (V, E_\pi)$, der V er alle verteksene i G (DFS undersøker alle verteksene) og E_π er alle kantene $(v.\pi, v)$, dvs. kanter mellom en verteks og forgjengeren. **Forløper subgrafen til DFS danner en dybde-først skog som består av flere dybde-først trær**. Kantene E_π kalles trekantene.



Merk: en verteks vil kun være i ett dybde-først tre!

DFS algoritme

DFS fungerer på lignende måte som BFS, men **bruker en LIFO-kø** istedenfor en FIFO-kø. Altså, vil køen til DFS fungere som en **stack** der **elementet som sist ble lagt til, er det første som blir tatt ut**. Dette gjør at det er enklere å implementere DFS rekursivt.

DFS vil også fargelegge vertekser underveis i søket for å holde styr på fremgangen. Alle vertekser er hvite i begynnelsen, blir grå når de blir oppdaget og svarte når de er ferdige, altså når nabolisten deres har blitt fullstendig oppdaget. Dette gjør at det er garantert at hver verteks ender opp i nøyaktig ett dybde-først tre, slik at trærne er atskilte. **DFS bruker også tidsstemplene $v.d$ som er tiden når v oppdages og $v.f$ som er tiden når søket er ferdig med å undersøke nabolisten til v** (og v blir svart). Disse gir informasjon om strukturen til grafen og om oppførselen til DFS, og de vil være heltall mellom 1 og $2|V|$. Vi har at $u.d < u.f$, og verteks u vil være hvit før $u.d$, grå mellom $u.d$ og $u.f$ og svart etter $u.f$. **DFS virker på både rettete og urettede grafer.**

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

DFS – lager en dybde-først skog for grafen G

Denne metoden tar inn en graf G, og vil lage en dybde-først skog (legg merke til at kildeverteksen ikke blir gitt). For-løkken ved linje 1-3 vil farge alle verteksene hvite og sette forgjengeren lik NIL. Linje 4 vil resette den globale *time* telleren. For-løkken ved linje 5-7 vil se på alle verteksene $u \in V$, og dersom denne er hvit vil den kalle på DFS-VISIT metoden. **Dette kallet vil gjøre at u blir roten til et nytt dybde-først tre i dybde-først skogen**. Når DFS returnerer vil hver verteks u ha fått en oppdagelsestid $u.d$ og en ferdigtid $u.f$.

Kjøretiden til DFS er $\Theta(V + E)$, siden alle noder farges grå og svarte, og alle kanter undersøkes. (Best-case: $\Theta(V + E)$ fordi DFS starter fra alle noder).

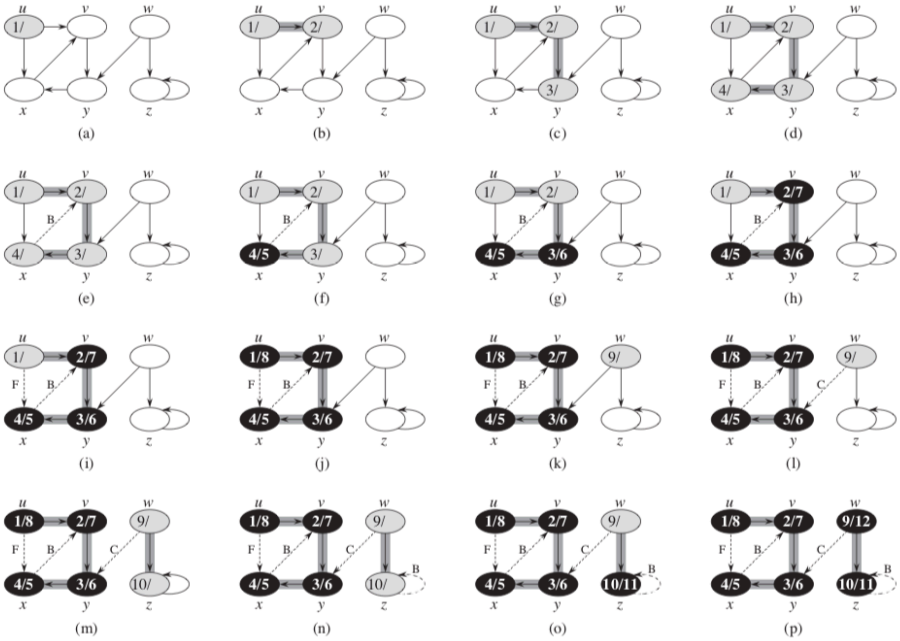
```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

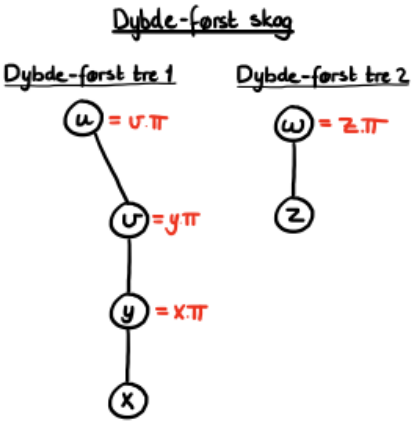
DFS-VISIT – lager et dybde-først tre for grafen med rot s

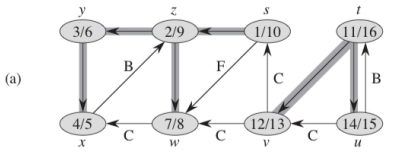
Denne metoden tar inn en graf G og en verteks u som er kildeverteksen, og vil lage et dybde først tre. Linje 1 vil øke *time* variabelen og linje 2 vil sette $u.d$ lik denne. Deretter vil u farges grå. For-løkken ved linje 4 vil undersøke alle verteksene i nabolisten til u. Dersom verteks v er hvit vil forgjengeren til v settes lik u og vi utfører et rekursivt kall med v som kildeverteks. Dette rekursive kallet vil sørge for at alle verteksene i nabomatrisen til v blir oppdaget og farget grå. Deretter vil v farges svart og for-løkken i det opprinnelige kallet går videre til neste verteks i nabolisten til u. Denne prosessen fungerer derfor på en LIFO måte, fordi ettersom prosedyren beveger seg nedover en bane vil den legge til vertekser som farges grå, helt til den når verteksen som ikke har noen hvite nabovertekser. Dette er den siste som ble lagt til stacken og vil være den første som farges svart. Deretter vil prosedyren bevege seg et hakk opp til forgjengeren og se på de andre verteksene til denne. Når alle verteksene er farget grå vil denne farges svart, og prosedyren beveger seg ett hakk opp. Slik vil den fortsette helt til den har undersøkt alle verteksene som kan nås fra u og u farges svart. Linje 9 vil øke *time* variabelen og linje 10 vil sette $u.f$ lik denne. DFS-VISIT vil så returnere og DFS metoden kan fortsette med å lage neste dybde-først tre.

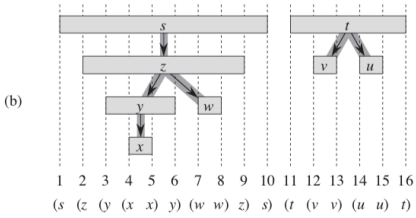
Figuren under viser et eksempel på hvordan DFS fungerer. Her kan vi se at verteks u blir valgt som første kildeverteks, og den har en nabomatrise som består av v og x. Hvilken av disse som velges er vilkårlig, og i dette tilfellet blir neste verteks v (merk: derfor kan

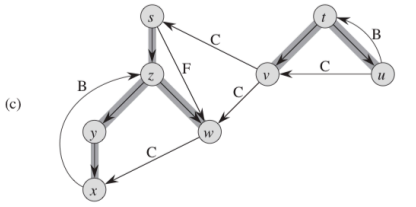
resultatet variere, men gir sjeldent problemer). Dermed vil de rekursive kallene bevege seg nedover til verteks x som ikke har noen hvite naboverteker og som derfor farges svart. Dermed vil prosedyren bevege seg oppover og farge verteksene svarte siden de har ingen andre hvite naboverteker. Merk: w er hvit, men kanten peker fra w til y, så derfor vil ikke nabolisten til y inneholde w. For å oppdage w og z, må derfor DFS lage et nytt dybde-først tre med w som kildeverteks. Figuren under viser dybde-først skogen som blir laget via π egenskapene.











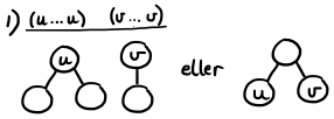
Parentesteoremet

Forgjenger subgrafen G_π vil danne en skog av dybde-først trær. $v.\pi = u$ kun hvis DFS-VISIT(G, v) har blitt kalt i løpet av søket gjennom nabolisten til u. **I tillegg vil v være en etterfølger til u, kun hvis v ble oppdaget når u var grå**. En viktig egenskap ved DFS er at oppdagelse og ferdig tidene har **parentesstruktur**. Dersom vi representerer **oppdagelsen av verteks u med en venstre parentes: "(u" og representerer at oppdagelsen er ferdig ved en høyre parentes "u)",** så vil historien med oppdagelser og avslutninger danne et velformet uttrykk siden parentesene er riktig nestet. For eksempel på figuren kan vi se at DFS av grafen på figur a, vil gi parentesene på figur b. Her ser vi at fra "(s" til "s)" vil det være flere andre vertekser som blir oppdaget og ferdig søkt gjennom, for eksempel "(x x)". Legg merke til at to vertekser som er atskilte, også vil ha atskilte parenteser, for eksempel "(x x)" og "(w w)". Dette er en god måte å beskrive hvordan dybde-først søket foregår!

<u>Parentesteoremet (viktig)</u>

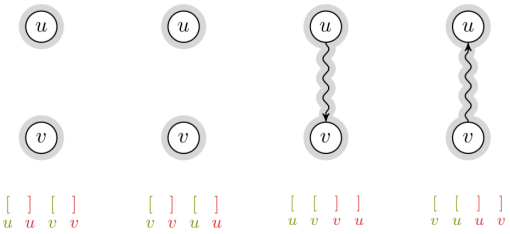
I DFS av en graf $G = (V, E)$ for to vertekser u og v, vil <u>én</u> av følgende være et krav:

1. **Intervallene $[u.d, u.f]$ og $[v.d, v.f]$ er helt atskilte, og verken u eller v er en etterfølger til den andre i dybde-først skogen**



2. **Intervallet $[u.d, u.f]$ er helt innenfor intervallet $[v.d, v.f]$, og u er en etterfølger av v i dybde-først treet**
3. **Intervallet $[v.d, v.f]$ er helt innenfor intervallet $[u.d, u.f]$, og v er en etterfølger av u i dybde-først treet**

For tilfelle 1 vil $u.f < v.d$, altså verteks v blir oppdaget når u er svart. Siden $u.d < u.f$, får vi at $u.d < u.f < v.d < v.f$, og derfor er intervallene til u og v helt atskilte. For tilfellet 2 vil $v.d < u.f$, altså v blir oppdaget når u fortsatt er grå og derfor vil u være forgjenger til v. Det samme gjelder for tilfelle 3, men bytter på u og v. **Verteks v er en ordentlig etterfølger av u i dybde-først skogen dersom $u.d < v.d < v.f < u.f$, fordi da vil hele intervallet til verteks v være innenfor intervallet til verteks u.**



<u>Hvit-bane teoremet</u>

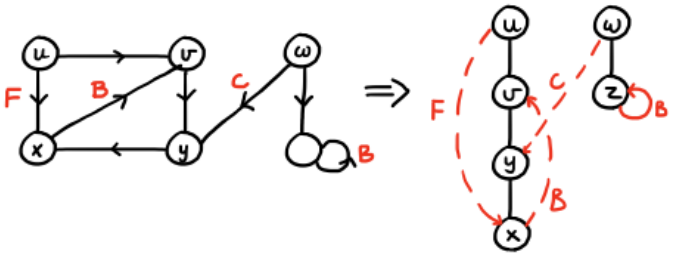
I et dybde-først tre vil **verteks v være en etterfølger til verteks u dersom banen mellom de kun består av hvite vertekser ved tidspunktet $u.d$, når u blir oppdaget**. Hvis ikke vil v ha blitt oppdaget tidligere fordi den er i nabolisten til en annen verteks w som oppdages før u, og dermed vil v være en etterfølger til w i stedet for u.

Klassifisering av kanter

DFS kan brukes for å klassifisere kantene til grafen $G = (V, E)$. Type kant kan gi viktig informasjon om en graf, for eksempel vil en rettet graf være asyklisk kun hvis DFS gir ingen bakoverkanter. DFS gir fire typer kanter:

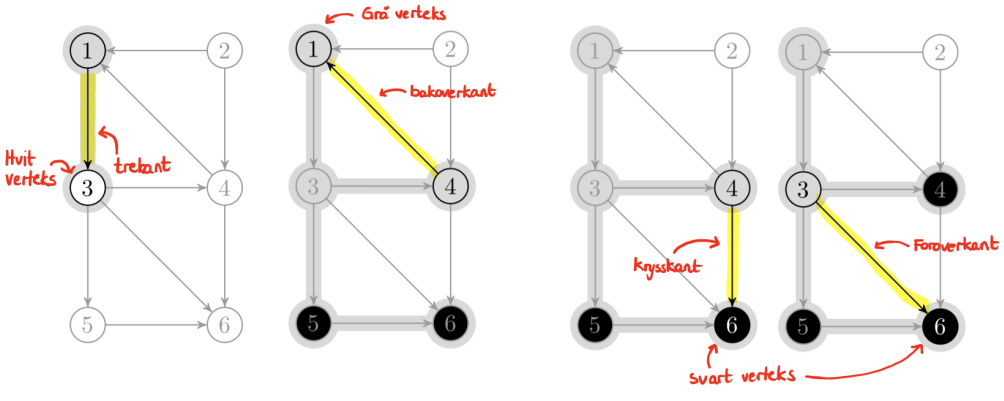
1. **Trekanter** – kanter i dybde-først skogen G_π. Kanten (u, v) vil være en trekant dersom v ble oppdaget første gang ved å undersøke kanten (u, v).
2. **Bakoverkanter** – kanter som kobler verteksen til en forgjenger. Løkker blir sett på som bakoverkanter.
3. **Foroverkanter** – kanter utenfor dybde-først skogen som kobler en verteks til en etterkommer i dybde-først treet.
4. **Krysskanter** – alle andre kanter. De kan gå mellom vertekser i samme dybde-først tre, så lenge en verteks ikke er en forgjenger til en annen. De kan også gå mellom vertekser i ulike dybde-først trær.

På figuren til høyre kan vi se et eksempel på bakoverkant (B), foroverkant (F) og krysskant (C).

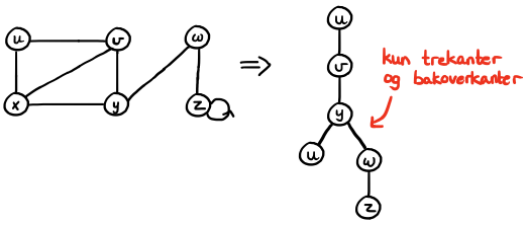


Når DFS undersøker en kant (u, v), vil **fargen til verteks v si noe om kanten:**

1. **Hvit v indikerer en trekant**
2. **Grå v indikerer en bakoverkant**
3. **Svart v indikerer en foroverkant ($v.d > u.d$) eller en krysskant ($v.d < u.d$)**



Denne klassifiseringen kan være tvetydig for urettede grafer, siden (u, v) og (v, u) er samme kant. I et slikt tilfelle vil vi klassifisere kanten som den første typen som kan brukes, altså basert på om det er (u, v) eller (v, u) som undersøkes først. **I DFS av urettede grafer vil kantene enten være trekanter eller bakoverkanter.** Siden kantene ikke har retning, vil foroverkanter omdannes til bakoverkanter og krysskanter omdannes til vanlige trekanter. En urettet graf vil derfor ikke ha foroverkanter eller krysskanter.



22.4 Topologisk sortering

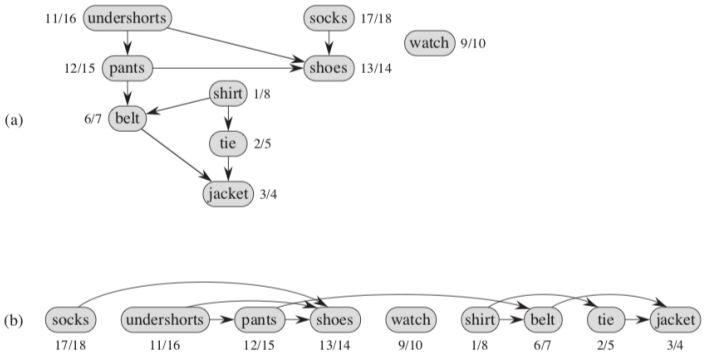
Vi kan bruke DFS for å utføre en topologisk sortering på en rettet asyklisk graf (DAG). En topologisk sortering er en lineær ordning av alle verteksene, slik at dersom G inneholder en kant (u, v), så vil u komme før v i ordningen. **En slik kant vil bety at u blir forgjengeren til v i dybde-først treet, og dermed at $v.f < u.f$ ($(u\ (v\ v)\ u)$).** En slik lineær ordning er ikke mulig dersom grafen inneholder en syklus, så derfor må G være en DAG. Ved topologisk sortering kan vi ordne verteksene langs en horisontal linje, slik at alle rettede kanter går fra venstre til høyre. Denne metoden blir ofte brukt når det er flere hendelser som må skje i bestemt rekkefølge, for eksempel påkledning.

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

TOPOLOGICAL-SORT – topologisk sortering vha DFS

Denne metoden vil gi verteksene en rekkefølge, der foreldre blir plassert før barna. Den bruker DFS for å regne ut *finishing* tiden til alle verteksene, og deretter vil den plassere verteksene i en lenket liste avhengig av disse tidsstemplene. Det er denne sorterte listen som blir returnert. **Kjøretiden er $\Theta(V + E)$** siden DFS tar tiden $\Theta(V + E)$ og det tar $O(1)$ tid å sette verteksene inn i lenket listen.

Figuren til høyre viser et eksempel, der topologisk sortering blir brukt for å bestemme rekkefølgen man skal kle på seg. **Verteksene er plassert i reversert orden basert på *finishing* tidene, altså vil denne tiden synke mot høyre.** Brøkene representerer *discover-/finishing* tider.



Korrekthet til topologisk sortering

En rettet graf er asyklisk hvis og bare hvis DFS ikke gir noen bakoverkanter. Dette fordi en bakoverkant vil bety at grafen inneholder en bane fra v til u, slik at det dannes en syklus. Vi ønsker å vise at TOPOLOGICAL-SORT produserer en topologisk sortering av en DAG, noe vi kan gjøre ved å vise at **$v.f < u.f$ når G inneholder en kant (u, v)**. Når kanten (u, v) blir undersøkt av DSF kan ikke v være grå, fordi da må v være forgjenger til u og (u, v) vil være en bakoverkant, slik at grafen blir syklisk. **Derfor må v være hvit eller svart**. Hvis v er hvit blir den en etterfølger av u, og dermed vil $v.f < u.f$. Hvis v er svart har den allerede blitt ferdig og siden vi fortsatt undersøker u vil $v.f < u.f$. Dermed har vi vist at $v.f < u.f$ for enhver kant (u, v) når G er en DAG. Altså vil en sortering basert på DSF og *finishing*-tid være topologisk.

Appendiks E – Generell graftraversering

Vi har sett på to traverseringsalgoritmer: bredde-først-søk (BFS) og dybde-først-søk (DFS). Disse kan fremstå ganske ulike, men de er nært beslektede. Dersom vi bytter ut FIFO-køen Q med en LIFO-kø eller stack, kan vi få BFS prosedyren til å oppføre seg omtrent likt med DFS. Da vil vi miste tidsstemplene ($v.d$ og $v.f$), men rekkefølgen noder farges grå og svarte på vil være den samme. En annen forskjell mellom DFS og BFS, slik vi har sett på de, er at DFS vil starte ved en vilkårlig node helt til den har nådd hele grafen. Sånn sett er BFS mer beslektet med DFS-VISIT.

Køen Q i BFS er en liste med noder vi har oppdaget via kanter fra tidligere besøkte noder, men som vi enda ikke har besøkt. Noder er hvite før de legges inn, grå når de er i Q og svarte etterpå. **Det at vi bruker en FIFO-kø er det som lar BFS finne de korteste stiene til alle noder, siden vi utforsker grafen "lagvis" utover**. Dersom vi kun ønsker å traversere alle verteksene som kan nås fra kildeverteksen, kan vi velge vilkårlige noder fra Q i hver iterasjon.

Vi skal se at Prims og Dijkstras algoritmer bruker svært lignende ideer, men MST-PRIM og DIJKSTRA forenkler ting ved å sette $Q = G.V$ før løkken som bruker Q, så vi mister den delen av traverseringen som handler om å oppdage noder. Disse prosedyrene kan fremstilles så de ligner mer på BFS, ved at noder legges inn i Q når de oppdages. **Den sentrale forskjellen mellom disse er hvilke noder som tas ut av Q, altså hvilke noder som prioriteres:**

- **BFS prioriterer de eldste (FIFO)**
- **DFS prioriterer de nyeste (LIFO)**
- **DIJKSTRA prioriterer noder med lavt avstandsestimat ($v.d$)**
- **PRIM prioriterer noder som har en lett kant til én av de besøkte (svarte nodene)**

Disse prioritetene blir oppdatert underveis. Andre prioriteringer vil gi andre traverseringsalgoritmer.

Forelesning 9 – Minimale spenntrær

Her har vi en graf med vekter på kantene, og ønsker å bare beholde akkurat de kantene vi må for å koble sammen alle nodene, men en så lav vektsum som mulig. Erke-eksempel på grådighet: Velg én og én kant, alltid den billigste lovlige. Læringsmålene er:

- Forstå *skog*-implementasjon av *disjunkte mengder* (CONNECTED-COMPONENTS, SAME-COMPONENT, MAKE-SET, UNION, LINK, FIND-SET)
- Vite hva spenntrær og minimale spenntrær er
- **Forstå GENERIC-MST**
- Forstå hvorfor *lette kanter* er *trygge kanter*
- Forstå MST-KRUSKAL
- Forstå MST-PRIM

Kapittel 21 – Datastrukturer for disjunkte sett

Noen applikasjoner vil fordele n ulike elementer over en samling av disjunkte sett (dvs. sett som er atskilte/ikke-overlappende). Vanlige operasjoner er å finne det unike settet som inneholder ett bestemt element eller foreningen av to sett. Vi skal se på en datastruktur som støtter disse operasjonene.

21.1 Disjunkt-sett operasjoner

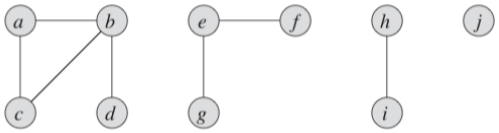
En disjunkt-sett datastruktur inneholder en samling $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ av disjunkte dynamiske sett (dvs. sett som støtter innsetting og sletting)**.** En representant brukes for å identifisere hvert sett, og det er et medlem av settet. Dette kan for eksempel være roten til treet eller det minste elementet. Hvert element i settet blir representert av et objekt x, og følgende operasjoner skal implementeres:

- **MAKE-SET(x)** – lager en nytt sett der eneste medlem er x. Siden settene er disjunkte kan ikke x være i noen av de andre settene
- **UNION(x, y)** – vil forene settene som inneholder x og y, til et nytt sett som er unionen av de to settene: $S_x \cup S_y$. Representanten vil bli et element fra denne unionen (vanligvis representanten til S_x eller S_y). Siden settene i $\mathcal{S}$ skal være disjunkte, vil vi fjerne S_x og S_y fra $\mathcal{S}$, og legge til det nye settet.
- **FIND-SET(x)** – returnerer en peker mot representanten til settet som har x.

Kjøretiden til disjunkt-sett datastrukturer kan analyseres mht. n som er antall MAKE-SET operasjoner eller m som er totalt antall MAKE-SET, UNION og FIND-SET operasjoner (merk: $m \geq n$). Etter n MAKE-SET operasjoner vil vi ha n disjunkte sett. Hver UNION operasjon vil redusere antall sett med 1, så derfor vil antall UNION operasjoner være maksimalt $n - 1$ (= ett sett igjen).

Bruksområde for disjunkt-sett datastrukturer VIKTIG

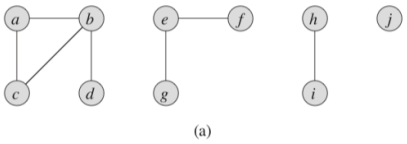
Disjunkt-sett datastrukturer brukes til å bestemme komponentene som er koblet sammen i en urettet graf. En urettet graf kan bestå av en eller flere **koblede komponenter**, som er vertekser koblet sammen av kanter. På figuren kan vi se en urettet graf som består av fire koblede komponenter.



```
CONNECTED-COMPONENTS(G)
1  for each vertex v ∈ G.V
2      MAKE-SET(v)
3  for each edge (u, v) ∈ G.E
4      if FIND-SET(u) ≠ FIND-SET(v)
5          UNION(u, v)
```

CONNECTED-COMPONENTS – finner koblede komponenter

Denne metoden bruker operasjonene for disjunkte sett til å finne de koblede komponentene i en urettet graf. Metoden begynner med å plassere alle verteksene i hvert sitt disjunkte sett. Deretter vil den gå igjennom alle kantene, og for hver kant (u, v) vil den forene settene som inneholder u og v dersom dette er to disjunkte sett. Etter å ha gått igjennom alle kantene vil metoden ha produsert en samling av disjunkte sett, der hvert sett inneholder alle verteksene i en koblet komponent.



(a)

Figuren viser et eksempel der vi har en urettet graf med fire koblete komponenter. Figur b viser de disjunkte settene ved hver iterasjon av for-løkken ved linje 3. I begynnelsen vil hver verteks ha sitt eget sett. Ved første iterasjon ser vi på kanten (b, d) og vi får $\{b\} \cup \{d\} = \{b, d\}$. Ved femte iterasjon ser vi på kanten $\{a, b\}$ og vi ser på $\{a, c\} \cup \{b, d\} = \{a, b, c, d\}$. Til slutt vil vi få en samling av fire disjunkte sett som representerer de fire koblete komponentene.

Edge processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

(b)

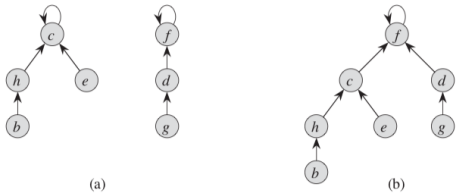
```
SAME-COMPONENT(u, v)
1  if FIND-SET(u) == FIND-SET(v)
2      return TRUE
3  else return FALSE
```

SAME-COMPONENT – sjekker om vertekser er i samme koblede komponent

Denne metoden vil returnere TRUE dersom to vertekser er i samme disjunkte sett, og FALSE ellers. CONNECTED-COMPONENTS vil sørge for at de disjunkte settene inneholder verteksene som er i samme koblede komponent, så derfor kan vi bruke FIND-SET på de to verteksene for å sjekke dette. I eksempelet over vil SAME-COMPONENT(a, d) returnere TRUE, mens SAME-COMPONENT(f, j) returnerer FALSE.

21.3 Disjunkte-sett skoger

Sett/mengder kan representeres av rotfaste trær. **I en disjunkt-sett skog vil hvert medlem kun peke mot sin foreldrenode og roten til treet er representanten for settet og er sin egen foreldrenode.** MAKE-SET operasjonen vil lage et tre med én node, FIND-SET operasjonen vil følge foreldrepekerne til den finner roten (nodene som besøkes kalles ***find path***), mens UNION operasjonen vil slå sammen to trær ved å få den ene roten til å peke på den andre roten (se figur). For å forbedre kjøretiden skal vi introdusere to heuristikker (tommelregler).

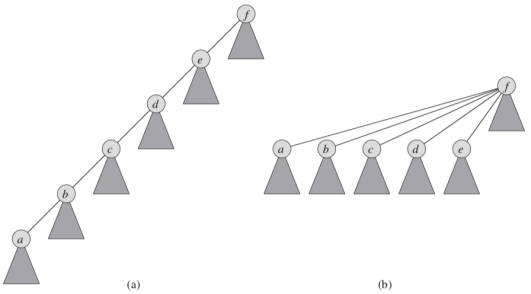


(a) (b)

Heuristikker for å forbedre kjøretiden

Vi kan bruke to heuristikker for å oppnå en kjøretid som er nesten lineær i totalt antall operasjoner m:

1. **Union etter rang** – for hver node vil vi holde styr på en rang, som er en øvre grense på høyden til noden (dvs. maksimum antall kanter fra x til et blad som er etterfølger av x). I UNION etter rang vil vi få roten med lavere rang til å peke mot roten med høyere rang. Dette er enkelt og svært effektivt!
2. **Banesammenligning** – hver node som blir besøkt i løpet av FIND-SET(a) (dvs. langs *find path*) blir satt til å peke direkte mot roten (se figur). Dette vil ikke endre rangen til noen av nodene.



(a) (b)

Operasjoner for disjunkt-sett skoger VIKTIG

Dersom vi inkluderer union etter rang og banesammenligning vil de tre operasjonene for disjunkt-sett skoger bli:

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0
```

MAKE-SET – lager en ny disjunkt-sett skog der x er eneste medlem

Denne metoden vil ta inn en node x og vil lage et nytt disjunkt-sett tre som kun består av denne noden, altså vil x være roten. Derfor setter vi $x.p = x$ (roten har seg selv som forelder) og $x.rank = 0$ (øvre grense på høyden til roten er 0).

```
UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))
```

UNION – forener to disjunkt-sett skoger

Denne metoden vil forene to disjunkt-sett skoger som inneholder node x og y vha LINK prosedyren.

```
LINK(x, y)
1  if x.rank > y.rank
2       y.p = x
3  else x.p = y
4       if x.rank == y.rank
5            y.rank = y.rank + 1
```

LINK – forener to disjunkte-sett skoger basert på rangen

Denne metoden tar inn to røtter og vil forene de to tilhørende disjunkte-sett skogene. Dersom rangen til x er høyest, vil dette gjøres ved å få y til å peke mot x og rangene er uendret. Dersom de to røttene har lik rang, vil vi velge en av de vilkårlig og øke rangen med 1. Deretter vil vi få den andre roten til å peke mot denne.

```
FIND-SET(x)
1  if x ≠ x.p
2       x.p = FIND-SET(x.p)
3  return x.p
```

FIND-SET – returnerer en peker mot roten til disjunkt sett skogen

Denne metoden vil ta inn en node x og vil returnere en peker mot roten, og den får også alle nodene mellom x og roten (kalles *find-path*) til å peke mot roten. Dersom x ikke er roten, vil metoden kalles rekursivt på foreldrenoden til x. Til slutt vil $x = x.p$ og vi returnerer $x.p = x$ (roten har seg selv som forelder). Siden alle nodene langs find-path har fått rota som forelder vil vi ha en **snarvei til neste gang**.

Heuristikken sin effekt på kjøretiden

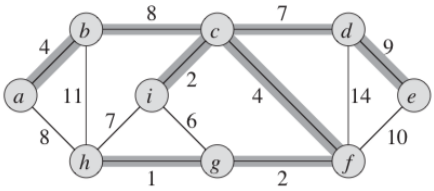
Union ved rang vil alene gi en kjøretid på $O(m \lg n)$, som er en tett grense. For n MAKE-SET operasjoner og f FIND-SET operasjoner, vil banesammenligning alene gi en worst-case kjøretid på $\Theta\left(n + f \cdot \left(1 + \lg_{2+f/n} n\right)\right)$. **Dersom vi bruker begge vil worst-case kjøretid være $O(m\,\alpha(n))$, der $\alpha(n) = O(\lg n)$ er en vekstfunksjon som vokser svært sakte** (se figur). I mange applikasjoner vil $\alpha(n) \leq 4$, slik at kjøretiden blir lineær mht. m.

$$\alpha(n) = \begin{cases} 0 & \text{hvis } 0 \leq n \leq 2, \\ 1 & \text{hvis } n = 3, \\ 2 & \text{hvis } 4 \leq n \leq 7, \\ 3 & \text{hvis } 8 \leq n \leq 2047, \end{cases}$$

Kapittel 23 – Minimale spenntrær

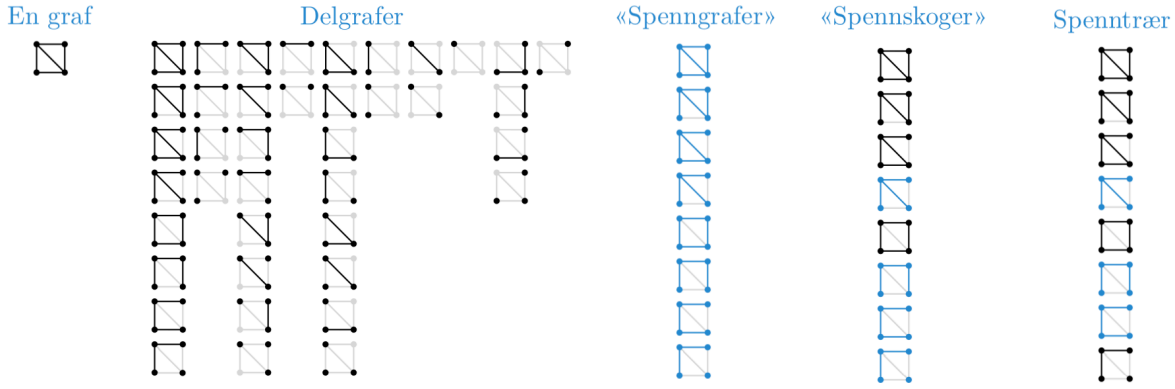
Vi har en urettet graf $G = (V, E)$, der hver kant $(u, v) \in E$ vil ha en vekt $w(u, v)$ som gir kostnaden forbundet med å koble sammen u og v (eks: mengde ledning som trengs for å koble sammen to nåler). Vi ønsker å finne **et asyklisk subsett $T \subseteq E$ som kobler sammen alle verteksene og minimerer den totale vekten** gitt av:

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$



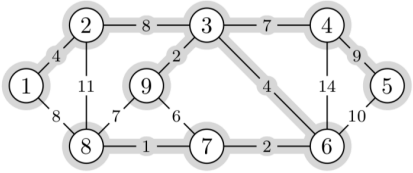
Siden T er asyklisk og kobler sammen alle vertekser på en slik måte at vekten minimeres, må den danne et tre som vi kaller **det minimale spennetreet** til grafen G. Problemet av å bestemme treet T kalles **minimum-spenntre problemet**. Det innebærer altså å finne ut hvilke kanter som skal velges ut for å danne et minimum spenntre, slik at total vekt blir minimert og det blir ikke dannet en syklus. På figuren kan vi se et eksempel på en graf der kantene til et minimum spenntre er markert. Dette treet har total vekt 37 og **det er ikke unikt**, siden (b, c) kan erstattes med (a, h) og fortsatt gi et asyklisk spennetre med total vekt 37.

Minimum-spenntre problemet kan løses med **Kruskals algoritme** og **Prims algoritme**, som begge kan ha kjøretid $O(E \lg V)$, dersom ordinære binære heaps blir brukt. **Disse to er grådige algoritmer**, siden de ved hvert steg vil ta et valg som ser best ut ved det øyeblikket.



Figuren over viser sammenhengen mellom en graf, delgrafer og spenntrær:

- **Graf** – består av verteksene og kantene mellom disse.
- **Delgrafer –** består av delmengder av nodene og kantene til den opprinnelige grafen. De kan altså variere i hvilke noder og kanter de inkluderer.
- **Spenngraf** – en dekkende delgraf, dvs. den har samme settet av noder som originalgrafen.
- **Spennskog** – en asyklisk spenngraf, dvs. spenngraf uten syklus
- **Spenntre** – en sammenhengende spennskog, altså må spenntreet ha samme nodesett som originalgrafen, ingen sykluser og det må være minst én kant til alle nodene.



Videre kan vi legge til **vekter på kantene**, noe som også omtales som **lengder** eller **kostnader**. Vi ønsker å knytte sammen nodene billigst mulig, altså finne en delmengde $T \subseteq E$ som inkluderer alle verteksene i V og minimerer $\sum_{e \in T} w(e)$. **Dersom w er positiv vil T alltid være asyklisk. Generelt vil vi tillate at w er negativ, men vi krever at T er asyklisk.**

23.1 Vekst av et minimalt spenntre

Vi ønsker å finne et minimalt spenntre for grafen $G = (V, E)$ som er koblet, urettet og har vektfunksjon $w(u, v)$. Dette kan gjøres med en grådig algoritme, som vi nå skal illustrere vha en generisk metode. **Denne metoden lar spenntreet vokse ved å legge til en kant om gangen til en kantmengde A. Loop invarianten er at A vil utgjøre en del av et minimalt spenntre før hver iterasjon.** En trygg kant vil være en kant som

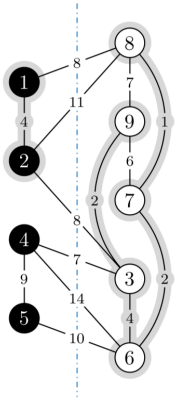
bevarer denne invarianten. Altså, vi kaller (u, v) for en **trygg kant** dersom den kan legges til A, slik at $A \cup (u, v)$ fortsatt er en del av et minimalt spenntre.

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

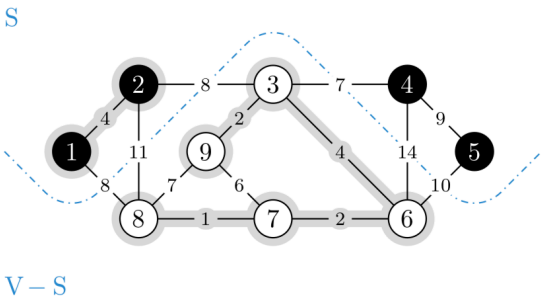
GENERIC-MST – grådig metode for å finne minimalt spenntre
Denne metoden tar inn en graf $G = (V, E)$ og en vektfunksjon w, og vil returnere et sett av kanter A som representerer et minimalt spenntre. Metoden begynner med å initialisere A til et tomt sett. While-løkken vil legge til en ny trygg kant så lenge A ikke er et minimalt spenntre enda, dvs. så lenge A ikke har dekt alle nodene i originalgrafen G.

Vi må vise tre ting ved loop invarianten:

- **Initialisering** – etter linje 1 vil $A = \emptyset \subseteq \mathcal{T}$
- **Vedlikehold** – kun trygge kanter blir lagt til, så derfor vil A fortsette å være en del av et minimalt spenntre etter hver iterasjon av while-løkken
- **Terminering** – alle kantene som legges til A er i et minimalt spenntre, så derfor må settet A som returneres være et minimalt spenntre.



Den utfordrende delen er å finne en trygg kant i linje 3. Vi skal se på et teorem som lar oss finne trygge kanter effektivt, men først noe terminologi: Et **snitt** $(S, V - S)$ er en oppdeling av nodesettet V (blå linje på figur). En kant $(u, v) \in E$ **krysser** kuttet dersom u er i S og v er i $V - S$. Et snitt **respekterer** et kantmengden A dersom ingen kanter i A krysser snittet (se figur). En **lett kant** over et snitt er kanten med minimal vekt blant kantene som krysser snittet. På figuren vil $(4, 3)$ være en (unik) lett kant over snittet $(S, V - S)$. Figuren til venstre viser en alternativ måte å tegne snittet.



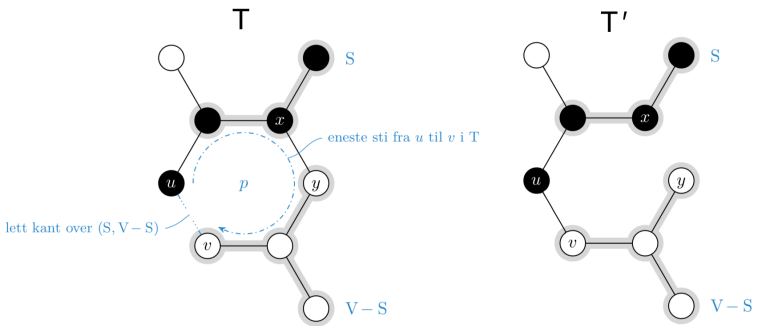
Trygg kant teoremet

La $G = (V, E)$ være en koblet, urettet graf med vektfunksjon w og A være et subsett av E som er inkludert i et minimalt spenntre for G. **Dersom $(S, V - S)$ er et snitt av G som respekterer A og (u, v) er en lett kant som krysser $(S, V - S)$, så vil (u, v) være en trygg kant for A.**

For å bevise dette lar vi T være et minimalt spenntre som inneholder A, men ikke (u, v). Vi skal bruke en klipp-ut-lim-inn teknikk for å vise at vi kan lage et annet minimalt spenntre T' som inkluderer $A \cup \{(u, v)\}$. De er tre ting vi må bevise:

1. **T' er et spenntre** – et minimalt spenntre T må være koblet til alle verteksene i grafen. Siden det er oppgitt at $(u, v) \notin T$, så må vi ha en enkel bane p som kobler u til v, slik at dersom vi legger til (u, v) blir det dannet en syklus (se figur). Vi har et snitt $(S, V - S)$ som deler T i to, der u og v er på hver sin side. Dette betyr at banen p må inneholde en kant (x, y) som krysser dette snittet. Siden snittet respekterer A, kan ikke (x, y) være en kant i A.
   Dersom vi fjerner kanten (x, y) vil T deles inn i to separate komponenter. Vi kan så legge til (u, v) for å koble disse komponentene sammen igjen, og vi vil da få et nytt spenntre **:**

$$T' = T - \{(x, y)\} \cup \{(u, v)\}$$



2. **T' er et minimalt spenntre** - Både (x, y) og (u, v) krysser $(S, V - S)$, og siden (u, v) er en lett kant vil $w(u, v) \leq w(x, y)$. Derfor vil:

$$w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$$

Siden T er et minimalt spenntre, vil $w(T') \leq w(T)$ bety at også T' er et minimalt spenntre.

3. **(u, v) er en trygg kant for A –** siden $A \subseteq T$ og $(x, y) \notin A$, så vil også $A \subseteq T'$. Siden $(u, v) \in T'$ vil derfor $A \cup \{(u, v)\} \subseteq T'$. Derfor vil (u, v) være en trygg kant for A

Dette teoremet gir altså at en lett kant over et snitt som respekterer løsningen vår vil være en trygg kant. Dette gjør at vi kan løse problemet vha grådige algoritmer, fordi ved hvert steg kan vi velge kanten som oppfyller dette kravet. Forskjellen er hvilket snitt vi bruker.

Egenskaper ved A som følger av trygg-kant-teoremet

I løpet av prosedyren GENERIC-MST vil A alltid være asyklisk, fordi ellers vil T som inkluderer A inneholde en syklus, noe som er en motsigelse. **Grafen $G_A = (V, A)$ vil være en skog og hver av de koblede komponentene i G_A er et tre.** En trygg kant (u, v) vil koble sammen to trær i G_A, siden $A \cup \{(u, v)\}$ må være asyklisk. Når $A = \emptyset$ vil det være $|V|$ trær (en for hver verteks) og hver iterasjon av while-løkken vil redusere antallet med 1, siden to trær blir kombinert. **Når skogen kun består av ett tre, vil metoden terminere. Dersom $C = (V_C, E_C)$ er en koblet komponent i skogen $G_A = (V, A)$, så vil (u, v) være en trygg kant for A dersom det er en lett kant som kobler C til en annen komponent i G_A.**

23.2 Kruskals og Prims algoritme

Kruskals og Prims algoritme er utvidelser av den generiske metoden og de bruker ulike regler for å bestemme hva som er en trygg kant:

- **Kruskals algoritme** – settet A danner en skog av flere trær. Den trygge kanten er den som har minst vekt i grafen og som kobler sammen to ulike vertekser uten å danne en syklus.
- **Prims algoritme** – settet A danner ett enkelt tre. Den trygge kanten er kanten med minst vekt som kobler treet til en verteks som ikke er i treet.

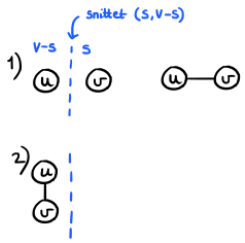
Kruskals algoritme

Kruskals algoritme vil finne en trygg kant som den legger til den voksende skogen ved å gå igjennom alle kantene som kobler samme to trær i skogen og deretter velge kanten (u, v) som har minst vekt. **Ved hvert steg vil algoritmen legge til kanten som har minst mulig vekt, så derfor er Kruskals algoritme en grådig algoritme.**

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

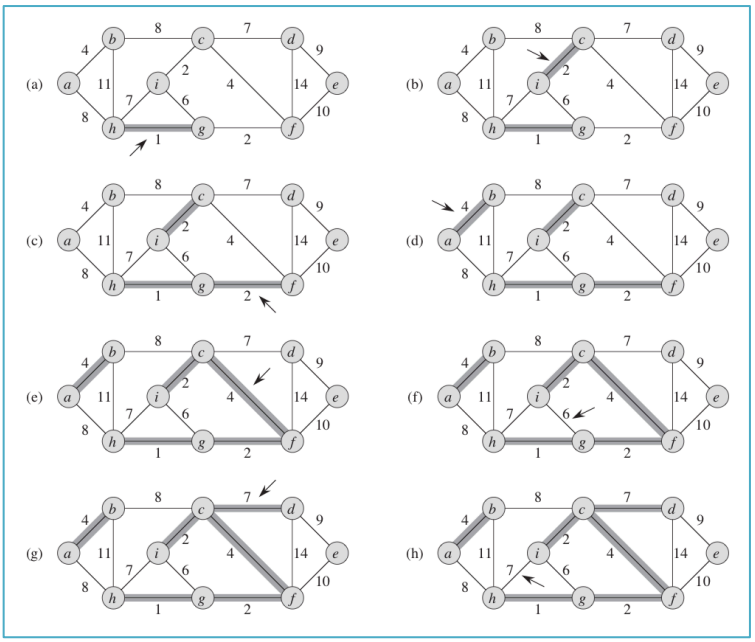
MST-KRUSKAL – finner minimale spenntrær

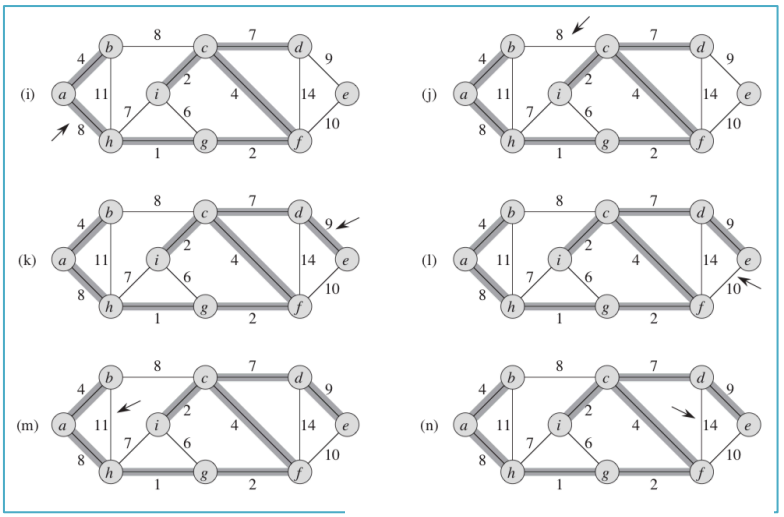
Denne metoden tar inn en graf $G = (V, E)$ og en vektfunksjon w, og vil returnere en kantmengde A som representerer et minimalt spenntre til G. For-løkken ved linje 2 bruker metoden MAKE-SET for å lage $|V|$ trær som hver inneholder én verteks. Ved linje 4 vil vi sortere kantene i G i økende rekkefølge basert på vekten.



Deretter vil for-løkken ved linje 5 gå igjennom alle kantene. For en kant (u, v) vil to trær slås sammen dersom u er i det ene treet og v er i det andre (dvs. de er på hver sin side av snittet som deler trærne fra hverandre). Sorteringen av kantene sørger for at (u, v) er kanten med minst vekt og siden u og v er i hvert sitt tre kan ikke unionen danne en syklus (figur 1). Derfor vil (u, v) være en trygg kant for A og legges derfor til denne kantmengden. Metoden UNION brukes for å kombinere de to trærne. Dersom u og v er i samme tre vil ingenting skje (figur 2) og for-løkken går til neste kant. Dette blir gjentatt helt til alle trærne i skogen er koblet sammen til ett, vha kanter som har blitt lagt til i A. Til slutt vil metoden returnere A. **Kjøretiden til Kruskals algoritme er $O(E \lg V)$** (bevis nedenfor).

Figuren under et eksempel. Metoden begynner med (h, g) siden den har lavest vekt og vil derfor plasseres først etter sorteringen. Dette vil føre til at treet med rot h og treet med rot g blir kombinert. Deretter vil metoden se p (c, i) siden den har vekt 2 og blir derfor plassert på andre plass etter sorteringen. De tro trærne blir kombinert, siden c og i er i hvert sitt tre. Merk: kanten (g, f) har også vekt 2 og det er vilkårlig hvilken en av disse som velges først, så lenge ikke en av de lager en syklus. **Kruskals algoritme vil gå igjennom alle kantene og vil legge kanten til A dersom den ikke danner en syklus.** For eksempel kan vi se at kant (i, g) ikke blir lagt til fordi ved dette tidspunktet vil i og g være i samme tre.





Operasjon	Antall	Kjøretid
MAKE-SET	V	O(1)
Sortering	1	O(E lg E)
FIND-SET	O(E)	O(α(V))
UNION	O(E)	O(α(V))

Kruskals algoritme analyse

Kjøretiden til Kruskals algoritme er $O(E \lg V)$. Vi antar at disjunkt-sett datastrukturen er implementert med union ved rang og banesammenligning, siden dette vil gi best asymptotisk kjøretid. Tabellen viser kjøretiden til de ulike stegene i algoritmen. Sorteringen av kantene tar tiden $O(E \lg E)$. MAKE-SET blir utført $|V|$ ganger (lager $|V|$ trær), mens FIND-SET og UNION blir utført $O(E)$ ganger (går igjennom alle kanter). Sammen vil disse bruke tiden $O\big((V + E)\alpha(V)\big)$. Siden grafen G er koblet, vil antall kanter være større enn antall vertekser: $|E| \geq |V| - 1$ og dermed blir kjøretiden $O\big(E\alpha(V)\big)$. Videre kan vi bruke at $\alpha(|V|) = O(\lg V) = O(\lg E)$ (siden $\alpha(n) = O(\lg n)$) og $|V|^2 > |E|$ for å finne at $O\big(E\alpha(V)\big) = O(E \lg E) = O(E \lg V)$.

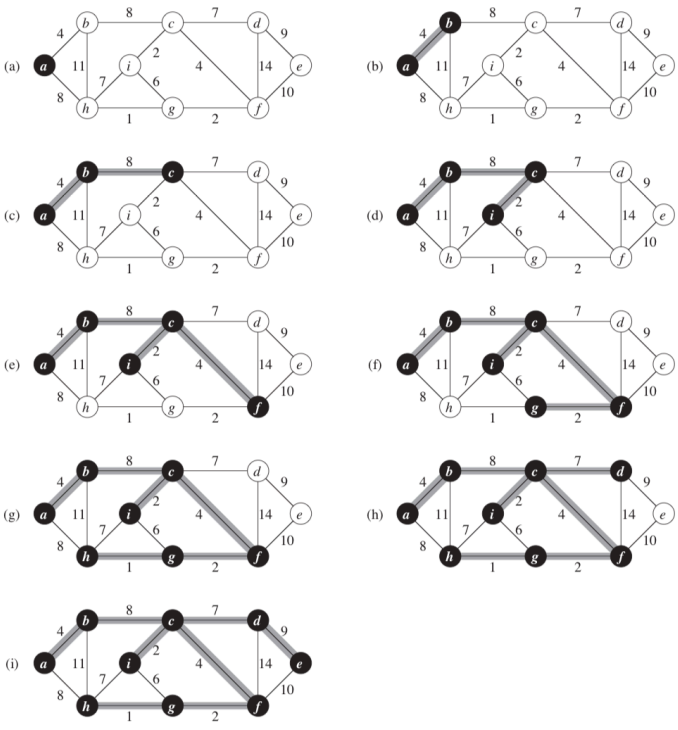
Prims algoritme

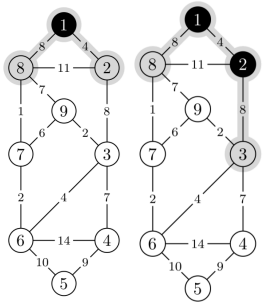
Prims algoritme finner minimalt spenntre på lignende måte som Dijkstras algoritme finner korteste bane i graf (mer senere). **Ved Prims algoritme vil kantene i kantmengden A alltid danne ett enkelt tre.** Dette treet vil starte i en vilkårlig rot og vil så vokse til det dekker alle verteksene i V. Hvert steg vil legge til en lett kant som kobler A til en verteks som ikke allerede er en del av A. **Prims algoritme er en grådig algoritme**, fordi ved hvert steg vil den legge til kanten som har lavest vekt, så lenge dette ikke lager en syklus. For at Prims algoritme skal være effektiv, må den ha en rask måte å velge neste kant som skal legges til A. Dette blir gjort vha en **min-prioritetskø Q som vil inneholde alle verteksene som ikke er i treet.** En verteks i Q vil ha en ***key*-attributt som er den minste vekten til en kant som kobler verteksen til treet**. Hvis verteksen ikke har noen kant til treet vil $v.key = \infty$. Foreldreverteksen til v er gitt av attributten $v.\pi$.

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

MST-PRIM – finner minimale spenntrær

Denne metoden tar inn en graf $G = (V, E)$, en vektfunksjon w og roten r, og den vil lage et minimalt spenntre som lagres i $v.\pi$ attributter. Den første for-løkken vil gå igjennom alle verteksene og initialisere key og π attributtene. key attributten til roten settes lik 0 og min-prioritetskøen Q fylles med alle verteksene, siden denne køen skal inneholde alle verteksene som ikke er i treet (ved dette tidspunktet består treet kun av roten r). While-løkken ved linje 6 vil utføres så lenge denne køen ikke er tom, fordi da er det fortsatt vertekser som ikke er en del av treet. Verteksene i Q vil ha *key*-attributter som er den letteste kanten mellom noden og treet. Metoden EXTRACT-MIN vil derfor hente ut verteksen u som er koblet til treet via den letteste kanten. For-løkken ved linje 8 vil oppdatere naboverteksene til u. Dersom naboverteksen ikke er en del av treet og vekten til (u, v) er lavere enn nåværende key-attributt til v, vil vi sette $v.\pi = u$ og oppdatere key-attributten til å være vekten til kanten (u, v). Denne if-setningen er nødvendig for å sikre at key-attributten er lik den letteste kanten mellom noden og treet (merk: u er nå en del av treet siden den fjernes fra Q, så derfor vil (u, v) være en kant mellom noden og treet). Kjøretiden til Prims algoritme er $O(E \lg V)$ (bevis på neste side).





Denne metoden kan illustreres vha farging av nodene. Svarte noder er en del av treet og kanter mellom slike er endelige. If-setningen ved linje 9 vil farge verteksene i nabolisten til u grå for å markere at π og key attributtene er endret. Ved neste steg vil EXTRACT-MIN hente den grå verteksen som har lavest key-attributt (dvs. lavest vekt). På figuren kan vi se at kanten $(1, 2)$ blir valgt først fordi den har lavest vekt.

Figuren til venstre viser et eksempel der $r = a$. Siden $(a, b) = 4$ er den letteste kanten vil $u = b$ ved første iterasjon. Dermed vil $c.key$ oppdateres til 8. Siden $w(a, h) = w(b, c)$ er det vilkårlig hvilken som velges først, men i dette tilfellet blir c valgt. Dermed blir $d.key = 7$, $i.key = 2$ og $f.key = 4$. Siden $w(c, i) = 2$ er den laveste vekten vil i velges som neste node. Slik fortsetter prosessen helt til $Q = \emptyset$.

Her vil snittet være $(Q, V - Q)$, altså mellom treet og verteksene som ikke er en del av treet, og (u, v) vil derfor krysse snittet.

Prims algoritme analyse

Kjøretiden til Prims algoritme er $O(E \lg V)$ som vi nå ønsker å bevise. Denne kjøretiden vil avhenge av hvordan vi implementerer min-prioritetskøen Q. Dersom Q er en binær min-heap kan vi bruke BUILD-MIN-HEAP for å lage Q, slik at linje 1-5 tar tiden $O(V)$. While-løkken utføres $|V|$ ganger og hvert kall på EXTRACT-MIN bruker tiden $O(\lg V)$, slik at total tid blir $O(V \lg V)$. For-løkken ved linje 8-11 blir utført $O(E)$ ganger (summen av lengden til nabolistene er $2|E|$) og hvert kall på DECREASE-KEY operasjonen tar tiden $O(\lg V)$ på en min-heap, slik at total tid blir $O(E \lg V)$. Den totale kjøretiden til Prims algoritme blir derfor $O(V) + O(V \lg V) + O(E \lg V) = O(V \lg V + E \lg V) = O(E \lg V)$, siden $E \geq |V| - 1$.

Denne kjøretiden kan forbedres til $O(E + V \lg V)$ dersom vi bruker Fibonacci heaps for å implementere Q, fordi DECREASE-KEY operasjonen for en slik heap bruker tiden $O(1)$.

Forelesning 10 – Korteste vei fra én til alle

Bredde-først søk kan finne stier med færrest mulig kanter, men hva om kantene har ulik lengde/vekt? Det generelle problemet er uløst, men vi kan løse problemet med gradvis bedre kjøretid for grafer uten negative sykler, uten negative kanter og uten sykler. Og vi bruker samme prinsipp for alle tre! Læringsmålene er:

- Forstå ulike varianter av *korteste-vei* eller *korteste-sti*-problemet (*Single-source, single-destination, single-pair, all-pairs*)
- Forstå strukturen til *korteste-vei*-problemet
- Forstå at negative sykler gir mening for korteste *enkle vei* (*simple path*)
- Forstå at *korteste enkle vei* kan løses vha. *lengste enkle vei* og omvendt
- Forstå hvordan man kan representere et *korteste-vei-tre*
- **Forstå *kant-slakking* (*edge relaxation*) og RELAX**
- Forstå ulike egenskaper ved korteste veier og slakking (*Triangle inequality, upper-bound property, no-path property, convergence property, path-relaxation property, predeccessor-subgraph property)*
- Forstå BELLMAN-FORD
- Forstå DAG-SHORTEST-PATH
- **Forstå kobling mellom DAG-SHORTEST-PATH og dynamisk programmering**
- Forstå DIJKSTRA

Kapittel 24 – Single-Source Shortest Paths

Vi har sett at **BFS algoritmen kan brukes for å finne korteste bane mellom to vertekser i en graf der alle kantene har samme vekt**. Denne algoritmen vil likevel ikke fungere på grafer der vekten er ulik. Vi skal nå se på hvordan vi kan finne korteste bane i slike tilfeller. I **shortest-path problemer** får vi gitt en rettet graf $G = (V, E)$ som har vektfunksjon w. Vekten $w(p)$ til banen $p = \langle v_0, v_1, \dots, v_k \rangle$ er summen av vekten til alle kantene i banen:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Shortest-path vekten $\delta(u, v)$ fra u til v er definert som:

Merk: i forelesning brukes **inn-nabo** for å betegne u og **ut-nabo** for å betegne v.

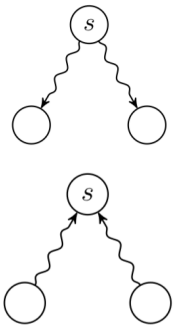
$$\delta(u, v) = \begin{cases} \min\{w(p) : u \rightsquigarrow^p v\} & \text{hvis det er en bane fra } u \text{ til } v \\ \infty & \text{ellers} \end{cases}$$

En shortest-path fra verteks u til verteks v er derfor definert som en bane p med vekt $w(p) = \delta(u, v)$.

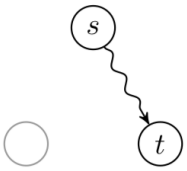
Ulike varianter av korteste-bane VIKTIG!

Det finnes flere varianter av shortest-path problemet gitt en graf $G = (V, E)$:

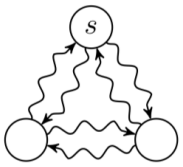
- **Single-source shortest path (SSSP) problem** – ønsker å finne den korteste banen fra en gitt kildeverteks $s \in V$ til enhver verteks $v \in V$.
- **Single-destination shortest path (SDSP) problem** – ønsker å finne den korteste banen til en gitt destinasjonsverteks $t \in V$ fra enhver verteks $v \in V$.



Dette problemet blir ekvivalent med SSSP dersom vi snur på retningen til alle kantene i grafen $G = (V, E)$.



- **Single-pair shortest path (SPSP) problem** – ønsker å finne korteste bane fra u til v for gitte vertekser u og v. Hvis vi løser SSSP med u som kildeverteks, vil vi også løse dette problemet. Vil ha samme worst-case kjøretid som SSSP algoritmer
- **All-pairs shortest path (APSP) problem** – ønsker å finne korteste bane fra u til v for alle vertekspar u og v. Dette kan løses ved å kjøre SSSP algoritmen for alle vertekser, men det er som regel andre metoder for å løse problemet raskere (mer kapittel 25).



Vi skal se på algoritmer som tar en rettet graf $G = (V, E)$ med vektfunksjon w og kildeverteks $s \in V$ som input. Output vil for hver node $v \in V$ være en bane $p = \langle v_0, v_1, \dots, v_k \rangle$, der $v_0 = s$ og $v_k = v$, som har minimal vektsum $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) = \delta(s, v)$. Men først skal vi se på noen egenskaper ved SSSP problemet.

Optimal delstruktur til shortest-paths

$G = (V, E)$ er en rettet graf med vektfunksjon w og $p = \langle v_0, v_1, \dots, v_k \rangle$ er korteste bane fra verteks v_0 til v_k. **Dersom p_{ij} er en delbane av p fra verteks v_i til v_j, så vil p_{ij} være korteste bane mellom disse verteksene.**

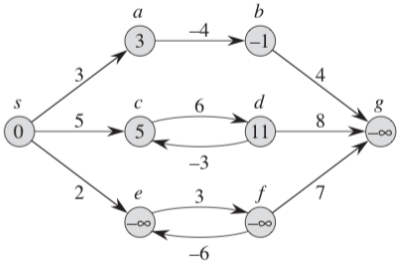
Shortest-paths problemer har altså optimal delstruktur, slik at korteste bane mellom to vertekser inneholder andre korteste baner. Dette kan bevises vha en klipp-ut-lim-inn teknikk, altså anta at p_{ij} ikke er optimal, sette inn en optimal bane og deretter vise at dette fører til en motsigelse. Vi begynner med å dekomponere banen p inn i $v_0 \rightsquigarrow^{p_{0i}} v_i \rightsquigarrow^{p_{ij}} v_j \rightsquigarrow^{p_{jk}} v_k$, slik at $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Deretter antar vi at det finnes en bane p'_{ij} fra v_i til v_j slik at $w(p'_{ij}) < w(p_{ij})$. Dermed vil det eksistere en bane $v_0 \rightsquigarrow^{p_{0i}} v_i \rightsquigarrow^{p'_{ij}} v_j \rightsquigarrow^{p_{jk}} v_k$ som har total vekt mindre enn $w(p)$, noe som er en motsigelse på at p er korteste bane fra v_0 til v_k

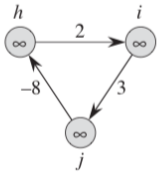
Optimal delstruktur er en av kravene for at dynamisk programmering og grådige algoritmer skal kunne brukes. Vi skal se på Dijkstras algoritme som er en grådig algoritme og Floyd-Warshalls algoritme som bruker dynamisk programmering. Disse algoritmene kan brukes siden shortest-path problemet har optimal delstruktur.

Kanter med negativ vekt VIKTIG

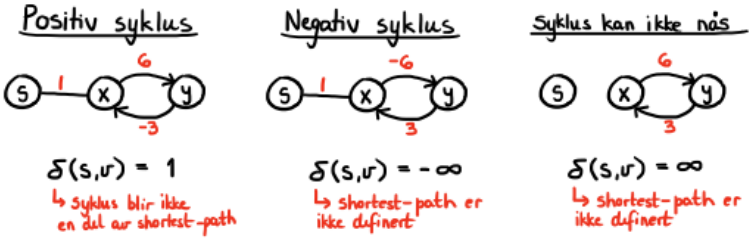
Single-source shortest-path problemer kan inkludere kanter som har negativ vekt ($\delta(s, v)$ kan være negativ), men det er viktig at disse ikke danner en syklus! **Shortest-path vekten $\delta(s, v)$ for banen mellom s og $v \in V$ er veldefinert dersom grafen $G = (V, E)$ ikke inneholder noen sykluser med negativ vekt som kan nås fra s.** Dersom en slik syklus kan nås fra s vil ingen bane kunne være shortest-path, fordi vi kan alltid finne en bane med lavere vekt ved å traversere den negative-vektede syklusen. Dersom det er en negativt-vektet syklus på banen fra s til v vil $\delta(s, v) = -\infty$.

Figuren illustrerer hvordan negative vekter og negativt-vektede sykluser kan påvirke shortest-path problemet. Det er kun én bane fra s til a, slik at $\delta(s, a) = w(s, a) = 3$. Det er også kun én bane fra s til b, slik at $\delta(s, b) = w(s, a) + w(s, b) = -1$. Det er uendelig mange baner fra s





til c: $\langle s, c\rangle$, $\langle s, c, d, c\rangle$, $\langle s, c, d, c, d, c\rangle$, osv. **Syklusen $\langle c, d, c\rangle$ er positiv siden den har vekt $w(c, d) + w(d, c) = 3 > 0$, og derfor vil korteste bane fra s til c være $\langle s, c\rangle$ med vekt $\delta(s, c) = w(s, c) = 5$.** På samme måte vil det være uendelig mange baner fra s til e: $\langle s, e\rangle$, $\langle s, e, f, e\rangle$, $\langle s, e, f, e, f, e\rangle$, osv. **Det vil likevel ikke være noen shortest-path mellom s og e, fordi syklusen $\langle e, f, e\rangle$ er negativ siden den har vekt $w(e, f) + w(f, e) = -3 < 0$.** Vi kan alltid finne en kortere bane fra s til e ved å traversere syklusen én gang til. Derfor vil $\delta(s, e) = -\infty$. Siden g kan nås fra f vil også $\delta(s, g) = -\infty$. Verteksene h, i og j danner også en negativ-vektet syklus, men disse kan ikke nås fra s. Derfor vil $\delta(s, g) = \delta(s, i) = \delta(s, j) = \infty$.



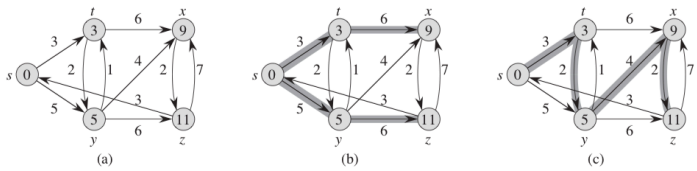
Shortest-path problemet vil altså kunne løses dersom den totale vekten til syklusen er positiv, fordi da vil en runde i syklusen kun øke vekten til banen. Dersom den totale vekten er negativ vil $\delta(s, v) = -\infty$, mens hvis syklusen ikke kan nås fra s vil $\delta(s, v) = \infty$. **Dijkstras algoritme antar at alle kantvektene er positive, mens Bellman-Ford algoritmen tillater kanter med negativ vekt så lenge en negativt-vektet syklus ikke kan nås fra kildeverteksen. Algoritmer kan som regel detektere og gi beskjed om at en slik syklus eksisterer.**

Sykluser

En negativt-vektet syklus vil ikke kunne gi en veldefinert shortest-path. Shortest-path vil være definert selv om en positivt-vektet syklus kan nås fra s, men denne syklusen kan ikke være en del av den korteste banen, fordi den vil kun øke vekten til banen. En 0-vekt syklus vil kunne være en del av shortest-path, fordi traversering av denne vil ikke øke den totale vekten og vi kan fjerne disse uten å endre vekten til banen. **Derfor kan vi anta at shortest-paths er enkle baner, altså at de har ingen sykluser.** Slike asykliske baner vil inneholde $|V|$ vertekser og maksimalt $|V| - 1$ kanter.

Representasjon av shortest-paths

Ofte vil vi finne verteksene på shortest-path i tillegg til vekten, noe som gjøres vha π attributten, altså forgjengere (likt bredde-først trær). Når shortest-path har blitt funnet vil hver verteks $v \in V$ ha en forgjengeren $v.\pi$ som enten er NIL (ikke del av kortest bane) eller verteksen før v i den korteste banen fra s til v. Vi ser igjen på forgjenger subgrafen $G_\pi = (V_\pi, E_\pi)$ fremkalt av π verdiene (s. 107). Her vil $V_\pi = \{v \in V : v.\pi \neq NIL\} \cup \{s\}$ og E_π er kantene mellom verteks og forløper i den korteste banen. G_π vil være et **shortest-path tre** som er et rotfast tre som inneholder en korteste bane fra roten s til alle vertekser som kan nås fra s. **Et bredde-først tre gir shortest-path til vertekser basert på antall kanter, mens et shortest-path tre gir shortest-path basert på vekt.** Shortest-paths er ikke alltid unike, og derfor vil vi kunne få flere ulike shortest-paths trær. Figuren viser en vektet, rettet graf som kan representeres av to shortest-paths trær med samme rot.



Relaksjon (*Relaxation*) VIKTIG

For hver verteks $v \in V$ vil det være en attributt $v.d$, kalt **shortest-path estimat**, som er en øvre grense på vekten til en shortest-path fra s til v. Denne blir initialisert av metoden på neste side.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

INITIALIZE-SINGLE-SOURCE – initialiserer d og π attributtene

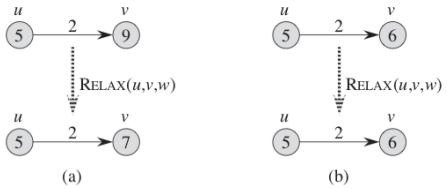
Denne metoden tar inn en graf $G = (V, E)$ og en kildeverteks s og vil initialisere d og π attributtene til alle verteksene i G. Dette gjøres ved at metoden går igjennom alle verteksene og setter $v.d = \infty$ (ingen korteste bane mellom s og v ved start) og $v.\pi = NIL$ (ingen forgjenger ved start). Til slutt vil metoden sette $s.d = 0$ (vekten mellom s og s er 0). **Kjøretiden er $\boldsymbol{\Theta(V)}$**, siden den går igjennom alle verteksene.

Etter initialisering vil vi derfor ha $v.\pi = NIL$ for alle $v \in V$, $s.d = 0$ og $v.d = \infty$ for alle $v \in V - \{s\}$. **Relaksering går ut på å teste om vi kan forbedre shortest-path til v ved å gå igjennom u og i så fall oppdatere $v.d$ og $v.\pi$.**

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

RELAX – relaksering av kanten (u, v)

Denne metoden tar inn to vertekser u og v, og en vektfunksjon w, og den vil sjekke om banen via u er en ny shortest-path fra s til v. Hvis den nåværende shortest-path fra s til v, betegnet som $v.d$, er større enn vekten fra s til u pluss $w(u, v)$, så har vi funnet en ny shortest-path fra s til v. Derfor vil vi oppdatere $v.d$ til $u.d + w(u, v)$ og sette $v.\pi = u$, siden u er den nye forgjengeren til v langs shortest-path for v. Dersom det finnes en kortere bane fra s til v, vil ingenting skje. **Kjøretiden er $\boldsymbol{O(1)}$.**



Figuren viser to eksempler for relaksering. Legg merke til at tallet i verteks v er d attributten, altså den totale vekten til nåværende shortest-path fra s til v. På figur a er $v.d = 9 > 7 = u.d + w(u, v)$, så derfor vil (u, v) relakseres, slik at $v.d = 7$ og $v.\pi = u$. På figur b er $v.d = 6 < 7 = u.d + w(u, v)$, så derfor vil ikke (u, v) relakseres, slik at $v.d = 9$ og $v.\pi \neq u$ (dvs. banen via u er ikke kortere).

Alle algoritmene for å finne shortest-paths bruker disse to metodene, men det varierer hvor mange ganger de relakserer hver kant og i hvilken rekkefølge relakseringen skjer.

Egenskaper ved shortest-path og relaksering

Noen viktige egenskaper ved shortest-path og relaksering er:

- **Triangel ulikhet** – for enhver kant $(u, v) \in E$, vil $\delta(s, v) \leq \delta(s, u) + w(u, v)$
- **Øvre-grense egenskap** – for hver verteks $v \in V$, vil $v.d \geq \delta(s, v)$ og når $v.d = \delta(s, v)$ vil den aldri endre verdi
- **Ingen-bane egenskap** – hvis det er ingen bane fra s til v, vil $v.d = \delta(s, v) = \infty$
- **Konvergens egenskap** – hvis $s \rightsquigarrow u \rightarrow v$ er korteste bane og $u.d = \delta(s, u)$, vil relaksering av (u, v) føre til at $v.d = \delta(s, v)$ hele tiden.
- **Bane-relaksering egenskap** – hvis p er en korteste bane fra s til v og vi relakserer kantene til p i rekkefølge, så vil v få $v.d = \delta(s, v)$. Det gjelder uavhengig av om andre slakkinger forekommer, selv om de kommer innimellom
- **Forgjenger-subgraf egenskap** – når $v.d = \delta(s, v)$ for alle $v \in V$, vil forgjenger subgrafen være en shortest-path tre med rot s.

Shortest-simple-path og longest-simple path

En enkel bane vil være en som ikke har en syklus, og i dette tilfellet vil longest-path mellom to vertekser i en vektet graf $G = (V, E)$ være lik shortest-path i grafen – G, som vi finner ved å endre alle vektene til sin negasjon (dvs. bytte fortegn).

Merk: i forelesning brukes **inn-nabo** for å betegne u og **ut-nabo** for å betegne v.

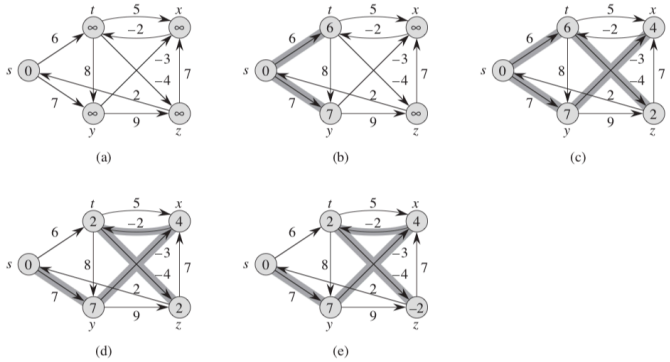
24.1 Bellman-Ford algoritme

Bellman-Ford algoritme vil løse single-source shortest-path (SSSP) problemet ved å relaksere alle kantene i grafen slik at det <u>må</u> bli rett. Disse kantene kan være negative, så lenge det ikke er noen negativt-vektet syklus.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

<u>BELLMAN-FORD – løser SSSP</u> <u>(tillater negative kanter)</u>
Denne metoden vil ta inn en graf $G = (V, E)$, en vektfunksjon w og en kildeverteks s, og vil returnere en shortest-path gitt via π attributtene dersom ingen negativ syklus kan nås fra s. Metoden begynner med å initialisere d og π attributtene. For-løkken ved linje 2 vil gå igjennom alle kantene i G og relaksere disse ved å kalle på RELAX metoden (merk: det oppgis ikke hvordan dette blir gjort, men en mulig implementasjon er å ta utgangspunkt i én verteks og se på alle utgående kanter). Ved å relaksere alle kantene vil metoden sørge for at $v.d = \delta(s, v)$ og $v.\pi$ vil være forgjengeren til v i shortest-path fra s til v (merk: dersom det er ingen slik bane vil $v.d = \infty$ og $v.\pi = NIL$). **Denne algoritmen vil returnere TRUE hvis og bare hvis grafen ikke inneholder en negativt-vektet syklus som kan nås fra s**. Dette gjøres ved å gå igjennom alle kantene som har blitt relaksert og sjekke om $v.d > u.d + w(u, v)$. Dersom dette er tilfellet etter relakseringen må det være en negativt-vektet syklus som inkluderer u og v, og metoden returnerer FALSE for å indikere at SSSP problemet har ingen løsning. **Kjøretiden er $O(VE)$**, siden initialiseringen tar tiden $\Theta(V)$, for-løkken ved linje 2 utføres $|V| - 1$ ganger, der hver iterasjon tar tiden $\Theta(E)$ og for-løkken ved linje 5 tar tiden $O(E)$.

Figur a viser at $v.d = \infty$ og $s.d = 0$ etter initialiseringen. Etter første iterasjon vil $t.d = 6$ og $y.d = 7$, og vi får $t.\pi = y.\pi = s$. Deretter blir prosessen gjentatt for alle kantene i grafen. På figur d kan vi se at kanten (s, t) blir fjernet fra shortest-path fordi det finnes en kortere bane fra s til t som går via x. Denne endringen fører til at shortest-path fra s til z også må oppdateres (figur d). Tilslutt vil vi ha en shortest-path fra s til alle vertekser v, der selve banen er gitt av π attributtene og vekten er gitt av d attributten. Denne grafen har ingen negativt-vektet syklus ($\langle t, x, t \rangle$ har positiv vekt), så derfor vil algoritmen returnere TRUE.



Bellman-Ford korrekthet

Følgende er bevis på at Bellman Ford gir en korrekt løsning på SSSP problemet:

- **Etter $|V| - 1$ iterasjoner av for-løkken på linje 2-4 vil $v.d = \delta(s, v)$ for alle vertekser som kan nås fra s**. *Bevis:* shortest-path mellom s og v vil ha maks $|V| - 1$ kanter siden den er enkel (ingen syklus, fordi da kan banen bli kortere ved å fjerne denne). For-løkken blir utført $|V| - 1$ ganger, slik at alle kantene blir relaksert i rekkefølge. Derfor vil bane-relaksering egenskapen gi at $v.d = \delta(s, v)$
- **For hver verteks $v \in V$ vil det være en bane fra s til v hvis og bare hvis BELLMAN-FORD terminerer med $v.d < \infty$**. $s.d$ initialiseres til ∞ og dersom v kan nås via verteks u som kan nås fra s vil RELAX føre til at $v.d = u.d + w(u, v)$, der $u.d \neq \infty$. Dersom u og v ikke kan nås fra s vil $u.d = v.d = \infty$, slik at RELAX metoden vil gi $v.d = u.d + w(u, v) = \infty$.

- **Dersom $G = (V, E)$ ikke inneholder en negativt-vektet syklus som kan nås fra s, vil BELLMAN-FORD returnere TRUE, $v.d = \delta(s, v)$ for alle $v \in V$ og G_π er et shortest-paths tre med rot s. Hvis G inneholder en negativt-vektet syklus som kan nås fra s, vil algoritmen returnere FALSE.** *Bevis:* Vi har allerede vist at $v.d = \delta(s, v)$ etter terminering. Dette kombinert med forgjenger-subgraf egenskap gir at G_π vil være et shortest-path tre. Det gjenstår å vise at algoritmen returnerer TRUE. Triangel ulikheten gir at $v.d = \delta(s, v) \leq \delta(s, u) + w(u, v) = u.d + w(u, v)$. Derfor vil ikke testen ved linje 6 utføres og algoritmen returnerer TRUE. Bevis ved motsigelse kan brukes for å vise motsatt tilfelle

Bellman-Fort med smartere relaksering

Dersom et estimat $v.d$ blir endret, så vil relaksering av kanter ut fra v ha vært bortkastet, siden dette nå må gjøres på nytt. For å gjøre relakseringen smartere, kan vi sørge for at kanter som går ut fra v kun relakseres når $v.d$ ikke kan forbedres mer, altså når $v.d = \delta(s, v)$. Dette kan oppnås ved å relaksere kanter i topologisk sortert rekkefølge (krever en rettet, asyklisk graf). Verteks u vil altså relakseres før verteks v. Dette gjør at antall relakseringer blir mindre, fordi vi reduserer antall situasjoner der v må relakseres på nytt etter at u har blitt relaksert. **Dersom grafen ikke inneholder negative kanter kan vi anta at tidligere noder har fått riktig estimat: $u.d = \delta(s, u)$.**

24.2 DAG-Shortest-Path algoritme

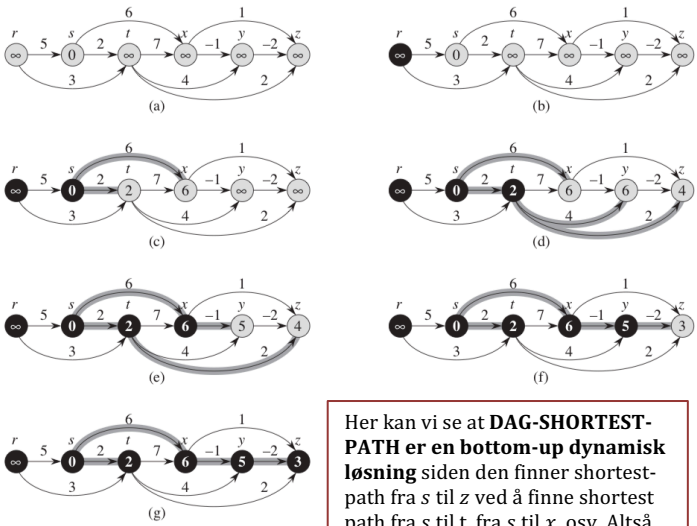
Dersom $G = (V, E)$ er en vektet DAG (rettet asyklisk graf) og vi relakserer kantene når verteksene er topologisk sortert, kan vi finne shortest-path ila tiden $\Theta(V + E)$. Shortest-paths er alltid definert i en DAG, fordi det er ingen sykluser.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4      for each vertex v ∈ G.Adj[u]
5          RELAX(u, v, w)
```

DAG-SHORTEST-PATHS – finner shortest-paths

Denne metoden vil ta inn en rettet DAG $G = (V, E)$, en vektfunksjon w og en kildeverteks s, og vil returnere en shortest-path gitt via π attributtene. Metoden begynner med å topologisk sortere verteksene i G (dvs. $(u, v) \in E$ betyr at u kommer før v i ordningen). Deretter vil den initialisere d og π attributtene. Den første for-løkken vil gå igjennom alle verteksene i grafen og den andre for-løkken vil relaksere alle kantene som går ut fra verteks u. Dette gjøres ved at for-løkken går igjennom nabolisten til u og vil for hver av naboverteksene kalle på RELAX metoden. Den topologiske ordningen gjør at vi kun passerer hver verteks én gang. **Kjøretiden er $\Theta(V + E)$**, siden topologiske sorteringen tar tiden $\Theta(V + E)$, initialiseringen tar tiden $\Theta(V)$ og for-løkkene blir utført E ganger til sammen, der kjøretiden til RELAX er $\Theta(1)$.

Figur a viser at $v.d = \infty$ og $s.d = 0$ etter initialiseringen og den topologiske sorteringen. Merk: kanten (r, s) gjør at verteks r kommer før s, og dermed vil det ikke finnes en shortest-path fra s til r, slik at $r.d = \infty$. På figur c vil $u = s$, slik at vi ser på nabolisten til s som inneholder t og x. Deretter vil prosedyren gjentas med $u = t$, osv. Legg merke til at verteksen blir merket svart når den er u, for å indikere at **$u.d$ blir ikke endret** etter dette tidspunktet, siden det er ingen sykler i grafen. De grå pilene markerer G_π treet.



Merk: i forelesning brukes **inn-nabo** for å betegne u og **ut-nabo** for å betegne v. Foreleser skiller mellom "reaching" og "pulling". "**Pulling**" er når vi ser på $v \in V$ og $(u, v) \in E$ og finner $v.d = u.d + w(u, v)$, altså vi ser på en ut-nabo og drar med oss info fra en inn-nabo u. "**Reaching**" er når vi ser på $u \in V$ og $(u, v) \in E$ og finner $v.d = u.d + w(u, v)$, altså vi sprer informasjon fra en inn-nabo u til ut-nabo v.

Her kan vi se at **DAG-SHORTEST-PATH er en bottom-up dynamisk løsning** siden den finner shortest-path fra s til z ved å finne shortest path fra s til t, fra s til x, osv. Altså den løser delproblemer for å finne løsningen på det originale problemet

DAG-Shortest-Path korrekthet

Hvis en vektet, rette graf $G = (V, E)$ har kildeverteks s og ingen sykluser, så vil termineringen av DAG-SHORTEST-PATH gi $v.d = \delta(s, v)$ for alle vertekser $v \in V$, og forgjenger subgrafen G_π er et shortest-path tre. *Bevis:* Dersom v ikke kan nås fra s, så vil ingen-bane egenskapen gi at $v.d = \delta(s, v) = \infty$. Dersom v kan nås fra s, så vil det eksistere en shortest-path p fra s til v. Siden verteksene i p er ordnet i topologisk rekkefølge vil kantene i p relakseres i rekkefølge, slik at bane-relaksering egenskapen gir at $v.d = \delta(s, v)$. Siden $v.d = \delta(s, v)$ for alle vertekser $v \in V$, vil forgjenger-subgraf egenskapen gi at G_π er et shortest-path tre.

Bellman-Ford og DAG-Shortest-Path – dynamisk programmering VIKTIG

Bellman-Ford og DAG-Shortest-Path bruker dynamisk programmering for å løse Shortest-path problemet fordi de **finner løsningen på delproblemet $(u.d)$ og bruker dette sammen med $w(u, v)$ for å finne løsningen på det "originale" problemet $v.d$. Hvilket valg som tas blir lagret i π attributtene, altså hvilken forgjenger som ga $v.d = \delta(s, v)$.** DAG-SHORTEST-PATH er en bottom-up løsning, siden relakserer kantene i topologisk orden helt til den når v.

24.3 Dijkstras algoritme

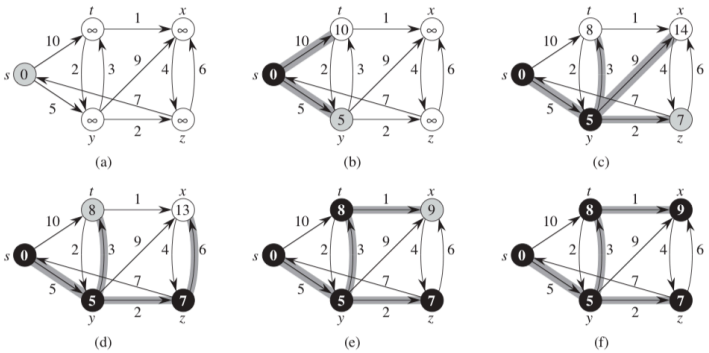
Dijkstras algoritme vil løse single-source shortest-path (SSSP) problemet dersom $G = (V, E)$ er en vektet, rettet graf og **alle kantene har positiv vekt: $w(u, v) \geq 0$ for alle kanter $(u, v) \in E$.** Kan ha lavere kjøretid enn Bellman-Ford. **Settet S inneholder verteksene der shortest-path er bestemt, mens min-prioritetskø Q inneholder verteksene der shortest-path ikke er bestemt**.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

DIJKSTRA – finner shortest-paths (tillater IKKE negative kanter)

Denne metoden vil ta inn en rettet graf $G = (V, E)$, en vektfunksjon w og en kildeverteks s, og vil returnere en shortest-path gitt via π attributtene. Metoden begynner med å initialisere d og π attributtene. Deretter vil den sette $S = \emptyset$ og fyller Q med alle verteksene i grafen. Q er en min-prioritetskø basert på d-verdien til verteksene, og så lenge denne ikke er tom vil while-løkken utføres. Ved linje 5 blir u satt lik verteksen med lavest d-verdi. Siden $s.d = 0$ og $v.d = \infty$ vil $u = s$ i første iterasjon. Dermed vil s legges til settet S og for-løkken går igjennom nabolisten til s. Hver utgående kant i nabolisten til s blir relaksert, slik at d attributtene blir oppdatert. Dermed vil neste iterasjon av while-løkke sette u til verteksen som fikk lavest d-verdi og prosedyren gjentas. Til slutt vil alle verteksene som kan nås fra s ha blitt lagt til S og vi har funnet shortest-path fra s til alle disse. Kjøretiden er $O(V \lg V + E \lg V)$ dersom alle verteksene kan nås fra s (bevis neste side). **Dijkstras algoritme grådig algoritme fordi den velger alltid den "letteste" verteksen i Q.**

Figur a viser at $v.d = \infty$ og $s.d = 0$ etter initialiseringen. Figur b viser grafen etter første iterasjon der $u = s$. Nabolisten til s består av t og y, så derfor vil (s, t) og (s, y) relakseres og vi får $s.d = 5$ og $t.d = 10$. Siden $s.d$ er minst vil $u = s$ ved neste iterasjon. Deretter blir prosedyren gjentatt helt til alle verteksene har blitt lagt til S.

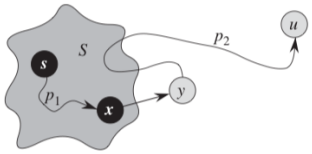


Observer at algoritmen vil aldri sette vertekser inn i Q etter linje 3 og at hver verteks blir hentet ut fra Q og plassert i S nøyaktig en gang. Derfor vil while-løkken utføres nøyaktig $|V|$ ganger. **Nodene blir farget svarte i det de legges til S eller i det de er ferdigbehandlet.**

Dijkstras algoritme korrekthet

Dijkstras algoritme som kjører på en vektet, rettet graf $G = (V, E)$ der ingen kanter har negativ vekt, vil terminere med $u.d = \delta(s, u)$ for alle vertekser $u \in V$. *Bevis:* vi kan bevise at dette stemmer vha loop invarianten: "Ved starten av hver iterasjon av while-loopen vil $u.d = \delta(s, u)$ for hver verteks $u \in S$". De tre tilfellene er:

- **Initialisering** – i begynnelsen vil $S = \emptyset$, og derfor er invarianten oppfylt.
- **Vedlikehold** – vi må vise at $u.d = \delta(s, u)$ for verteksen som legges til S. Dette gjør vi ved å anta at u er første verteks som legges til S der $u.d \neq \delta(s, u)$ og vise at dette fører til en motsigelse. Denne antagelsen gjør at $u \neq s$ siden $s.d = \delta(s, s) = 0$ og det må være minst én bane fra s til u for ellers vil ingen-bane egenskapen gi at $u.d = \delta(s, u) = \infty$. Siden det er minst én bane vil det være en shortest-path p fra s til u. Dersom vi lar y være første verteks som ikke er i S og x er forgjengeren til y, kan denne banen deles inn i $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$ (se figur, merk: kan hende at $x = s$ og $y = u$). Ved tidspunktet der u blir lagt til S vil $x.d = \delta(s, x)$ (pga. antagelsen) og (x, y) er relaksert. Konvergens egenskapen gir derfor at $y.d = \delta(s, y)$ når u blir lagt til S. Siden y er foran u langs den korteste banen og alle vektene er positive vil $\delta(s, y) \leq \delta(s, u)$. Øvre grense egenskapen gir også at $u.d \geq \delta(s, u)$. Dermed får vi at:



$$y.d = \delta(s, y) \leq \delta(s, u) \leq u.d$$

  Siden både u og v var i Q når u ble valgt, må det bety at $u.d \leq y.d$. Derfor vil eneste mulighet være at $u.d = \delta(s, u)$, noe som er en motsigelse på antagelsen. Derfor kan vi konkludere med at $u.d = \delta(s, u)$ når u blir lagt til S og at dette gjelder alle gangene etterpå.
- **Terminering** – ved terminering vil $Q = V - S = \emptyset$, noe som gir at $S = V$. Dermed vil $u.d = \delta(s, u)$ for alle vertekser $u \in V$.

Dijkstras algoritme analyse

Operasjon	Antall	Kjøretid
Initialisering	1	Θ(V)
BUILD-HEAP	1	Θ(V)
EXTRACT-MIN	V	O(lg V)
DECREASE-KEY*	E	O(lg V)

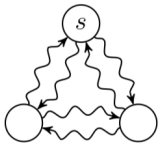
Kjøretiden til Dijkstras algoritme er $O(V \lg V + E \lg V)$. Initialiseringen tar tiden $\Theta(V)$. Dersom vi implementerer Q med en binær min-heap vil BUILD-HEAP bruke tiden $\Theta(V)$, EXTRACT-MIN bruker tiden $O(\lg V)$ og $DECREASE\text{-}KEY$ som er nødvendig i RELAX bruker tiden $O(\lg V)$. Siden hver verteks i Q kun blir hentet ut én gang vil EXTRACT-MIN utføres $|V|$ ganger, slik at total kjøretid blir $O(V \lg V)$. Siden for-løkken ved linje 7 undersøker alle verteksene i nabolisten og disse har samlet lengde $|E|$ (s. 104), vil $DECREASE\text{-}KEY$ utføres maksimalt $|E|$ ganger, slik at total kjøretid blir $O(E \lg V)$. Total kjøretid blir $\Theta(V) + \Theta(V) + O(V \lg V) + O(E \lg V) = O(V \lg V + E \lg V)$.

Forelesning 11 – Korteste vei fra alle til alle

Vi kan finne de korteste veiene fra hver node etter tur, men mange av delproblemene vil overlappe. Om vi har mange nok kanter lønner det seg derfor å bruke dynamisk programmering med dekomponeringen "skal vi innom k eller ikke?" Læringsmålene er:

- Forstå *forgjengerstrukturen* for *alle-til-alle* varianten av korteste-vei-problemet (PRINT-ALL-PAIRS-SHORTEST-PATH)
- Forstå FLOYD-WARSHALL
- Forstå TRANSITIVE-CLOSURE
- Forstå JOHNSON

Kapittel 25 – All-Pairs Shortest Paths



Vi skal se på APSP problemet som går ut på å finne **shortest-paths mellom alle vertekspar** i en rettet graf $G = (V, E)$ med vektfunksjon w. Vi ønsker altså å finne korteste bane fra u til v for alle vertekspar u og v. APSP algoritmer vil som regel ha output i tabellform, der **oppføringen i rad u og kolonne v vil være vekten til shortest-path mellom u og v.** APSP problemet kan løses ved å kjøre Single-Source Shortest-Path (SSSP) algoritmen $|V|$ ganger, altså en gang for hver verteks som kilde. Hvis grafen ikke har noen negative kanter kan vi bruke Dijkstras algoritme en gang for hver verteks. Kjøretiden til denne vil avhenge av hvordan vi implementerer min-prioritetskøen Q, for eksempel vil den være $O(V^3 + VE)$ med tabell, $O(VE \lg V)$ med binær min-heap og $O(V^2 \lg V + VE)$ med Fibonacci heap. Hvis grafen har negative kanter må vi bruke den tregere Bellman-Ford algoritmen en gang for hver verteks. Denne vil gi en kjøretid på $O(V^2 E)$. Det finnes likevel andre metoder for å løse APSP problemet raskere, og vi skal nå se på disse.

SSSP algoritmer antar at grafen bruker en naboliste representasjon, mens **de fleste algoritmene som løser APSP problemet bruker en nabomatrise representasjon** (Johnsons algoritme er unntak). Input til disse algoritmene er altså en rettet graf G med n vertekser, der **vekten til kant (i, j) er gitt av en $n \times n$ matrise $W = (w_{ij})$**, der:

$$w_{ij} = \begin{cases} 0 & \text{hvis } i = j \\ w(i,j) & \text{hvis } i \neq j \text{ og } (i,j) \in E \\ \infty & \text{hvis } i \neq j \text{ og } (i,j) \notin E \end{cases}$$

Kanter med negativ vekt er tillatt, men vi antar for nå at de har positiv vekt. **Output til APSP algoritmer vil være en $n \times n$ matrise $D = (d_{ij})$, der oppføring d_{ij} inneholder vekten til shortest-path fra verteks i til verteks j, altså $d_{ij} = \delta(i, j)$.**

Algoritmene vil også finne en **forgjenger matrise $\prod = (\pi_{ij})$, der π_{ij} er forgjengeren til j langs en shortest-path fra i**. Dersom $i = j$ eller det er ingen bane fra i til j, vil $\pi_{ij} = NIL$. Dersom vi tar utgangspunkt i en kildeverteks i, dvs. vi ser på rad i, vil kolonnene i $\prod$ matrisen representere shortest-paths tre med rot i. Vi kaller disse forgjenger subgrafene for $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, der $V_{\pi,i} = \{j \in V : \pi_{ij} \neq NIL\} \cup \{i\}$ (dvs. alle verteksene

j som har en forgjenger) og $E_{\pi,i}$ er alle kantene mellom en verteks j og dens forgjenger π_{ij}. Dersom $G_{\pi,i}$ er et shortest-paths tre, vil følgende prosedyre printe shortest-path fra verteks i til verteks j:

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
1  if i == j
2      print i
3  elseif π_ij == NIL
4      print "no path from" i "to" j "exists"
5  else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
6      print j
```

PRINT-ALL-PAIRS-SHORTEST-PATH

Denne metoden tar inn en $\prod$-matrise av forgjengere og verteksene i og j, og vil printe shortest-path fra i til j. Dersom disse er like ($i = j$) vil shortest-path være en loop fra i til i, så metoden printer i. Dersom j ikke har noen forgjenger langs shortest-path ($\pi_{ij} = NIL$) vil det ikke være noen shortest-path til j. Hvis ikke disse utføres vil metoden kalles rekursivt på i og forgjengeren til j, altså finner den shortest path fra i til j ved å se på tidligere shortest-paths (husk: shortest-paths har optimal delstruktur). Hvert kall vil printe j, altså verteksen i enden av de ulike korteste banene. I bunnen vil metoden nå kilden, slik at første if-setning printer i. Deretter vil metoden bevege seg oppover og printe j, slik at den til slutt har printet alle verteksene langs shortest-path fra i til j.

25.2 Floyd-Warshall algoritme

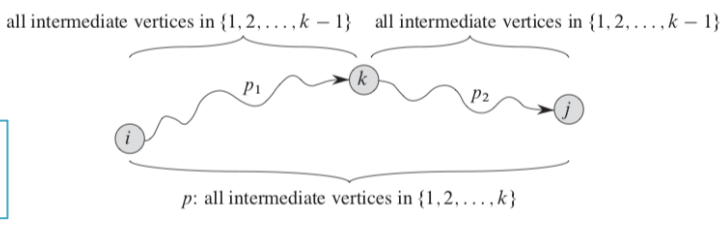
Floyd-Warshall algoritme er en form for **dynamisk programmering** som løser APSP problemet på en rettet graf $G = (V, E)$ med **kjøretid $\boldsymbol{\Theta(V^3)}$**. Denne tillater kanter med negativ vekt, men ingen negativt-vektede sykluser. Siden dette er en dynamisk programmeringsalgoritme, bruker vi stegene på side 82 for å finne algoritmen.

Fire steg for å finne dynamisk programmeringsalgoritme:
1. Karakteriser strukturen til en optimal løsning
2. Rekursivt definer verdien til en optimal løsning
3. Regn ut verdien til en optimal løsning, vanligvis bottom-up
4. Lag en optimal løsning fra den utregnede informasjonen.

Steg 1 – Strukturen til shortest path

Floyd-Warshall algoritme ser på de **mellomliggende verteksene** til en shortest-paths. Altså for en simpel bane $p^* = \langle v_1, v_2, \dots, v_l \rangle$ vil disse være alle verteksene bortsett fra v_1 eller v_l. Floyd Warshall algoritme bygger på følgende observasjon. Vi antar at verteksene til G er $V = \{1, 2, \dots, n\}$ og ser på et subsett av vertekser $\{1, 2, \dots, k\}$. For ethvert vertekspar $i, j \in V$, ser vi på alle banene fra i til j der de mellomliggende verteksene hentes fra $\{1, 2, \dots, k\}$ og vi lar p være banen som har minst vekt. Floyd-Warshall algoritme bruker forholdet mellom banen p og shortest-paths fra i til j når alle mellomliggende vertekser hentes fra $\{1, 2, \dots, k-1\}$. Dette forholdet avhenger av om k er en mellomliggende verteks i bane p eller ikke:

- **k er ikke en mellomliggende verteks til bane p** – alle mellomliggende vertekser til bane p er i settet $\{1, 2, \dots, k-1\}$. Derfor vil shortest-path fra verteks i til verteks j når alle mellomliggende vertekser hentes fra $\{1, 2, \dots, k-1\}$ også være en shortest-path fra verteks i til verteks j når alle mellomliggende vertekser hentes fra $\{1, 2, \dots, k\}$
- **k er en mellomliggende verteks til bane p** – vi dekomponerer p inn i $i \rightsquigarrow^{p_1} k \rightsquigarrow^{p_2} j$ (se figur). Den optimale delstrukturen gir at p_1 er shortest-path fra i til k, der alle mellomliggende vertekser hentes fra $\{1, 2, \dots, k-1\}$, siden k ikke er en mellomliggende verteks for p_1. På samme måte vil p_2 være shortest-path fra k til j, der alle mellomliggende vertekser hentes fra $\{1, 2, \dots, k-1\}$, siden k heller ikke er en mellomliggende verteks for p_2.



Her ser vi at dekomponeringen gjør at vi får to baner der mellomliggende vertekser kan hentes fra $\{1, 2, \dots, k-1\}$, siden k ses på som en verteks i enden eller begynnelsen av banen

Steg 2 – Rekursiv løsning på All-Pairs Shortest-Path (APSP) problemet

Vi lar d_{ij}^{k} være vekten til shortest-path fra verteks i til verteks j når alle mellomliggende vertekser hentes fra $\{1, 2, \dots, k\}$. Merk: i og j kombinert med de mellomliggende verteksene gir hele shortest-path. Når $k = 0$, vil en bane fra i til j ikke ha noen mellomliggende vertekser, og derfor vil $d_{ij}^{0} = w_{ij}$ (vekten til shortest-path er lik vekten til kant (i, j)). Observasjonen på forrige side gir at vi enten må undersøke ett eller to delproblem:

- **Ett delproblem (d_{ij}^{k-1}):** k er ikke i shortest-path, så delproblemet er å finne shortest-path fra i til j når mellomliggende vertekser hentes fra $\{1, 2, \dots, k-1\}$
- **To delproblem ($d_{ik}^{k-1} + d_{kj}^{k-1}$):** k er i shortest-path, så delproblemene blir å finne shortest-path fra i til k og fra k til j når mellomliggende vertekser hentes fra $\{1, 2, \dots, k-1\}$

Hvilken av disse vi får vil bestemmes av den som gir minst vekt, siden k vil inkluderes i banen p kun hvis det gjør at banen får minst-mulig vekt. Derfor kan vi definere d_{ij}^{k} rekursivt som:

$$d_{ij}^{k} = \begin{cases} w_{ij} & \text{hvis } k = 0 \\ \min\left(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right) & \text{hvis } k \geq 1 \end{cases}$$

For enhver bane vil mellomliggende vertekser være i settet $\{1, 2, \dots, n\}$, så derfor vil matrisen $D^n = d_{ij}^{n}$ gi det endelige svaret $d_{ij}^{n} = \delta(i, j)$ for alle $i, j \in V$.

Steg 3 – Bottom-up utregning av vekten til shortest-paths

Basert på rekurrensen over kan vi lage en bottom-up prosedyre for å regne ut d_{ij}^{k} for økende verdier av k:

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

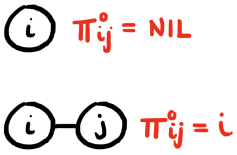
FLOYD-WARSHALL – finner shortest-paths vekter

Denne metoden tar inn en $n \times n$ matrise W som gir vekten til kant (i, j) og vil returnere D^n som er en $n \times n$ matrise for shortest-paths fra verteks i til verktes j. Metoden begynner med å sette n lik antall rader i W. Siden dette er en bottom-up metode må vi begynne med å finne D^0. Ved linje 2 setter vi $D^0 = W$, siden $d_{ij}^{0} = w_{ij}$, altså oppføringene i D^0 skal være vekten til kant (i, j) som er oppføringene i W. Deretter vil for-løkken ved linje 3-7 finne D^n ved å løse delproblemene fra bunnen av og opp. Vi begynner med å finne shortest-paths fra i til j når $k = 1$, dvs. mellomliggende vertekser kan velges fra $\{1\}$. Vi lager en ny $n \times n$ matrise D^1 som skal inneholde elementer d_{ij}^{1}, som vi finner vha en dobbelt for-løkke som går igjennom alle verteksene i og j. For hvert vertekspar vil vi finne shortest-path vekten d_{ij}^{k} ved å se på den minste av de to situasjonene (1 er med i shortest path eller ikke). Etter for-løkken vil det ha blitt laget en ferdig matrise D^1. Deretter vil for-løkken se på $k = 2$ og bruker løsningene fra D^1 (dynamisk programmering). Denne prosedyren gjentas helt til vi finner D^n som blir returnert. **Kjøretiden er $\Theta(n^3)$**, fordi de tre for-løkkene utføres n ganger og hver iterasjon tar tiden $O(1)$.

Merk: denne metoden vil kun finne vekten til shortest-paths for alle vertekspar $i, j \in V$. Vi skal nå se hvordan vi kan finne løsningen i form av verteksene langs denne banen.

Steg 4 – Konstruksjon av shortest-path

Det er flere ulike måter å lage shortest-paths i Floyd-Warshall algoritmen. En måte er å finne matrisen D for vekten til shortest-paths og deretter bruke denne for å lage forgjengermatrisen Π. En annen måte er å finne forgjengermatrisen Π samtidig som algoritmen regner ut D^k. Altså vil vi regne ut $\Pi^0, \Pi^1, \dots, \Pi^n$, der $\Pi = \Pi^n$ og vi definerer π_{ij}^k som forgjengeren til verteks j på shortest-path fra i når alle mellomliggende vertekser hentes fra $\{1, 2, \dots, k\}$. For å inkludere denne utregningen i algoritmen, er vi nødt til å finne en rekursiv løsning på π_{ij}^k. Ved $k = 0$ (se figur) vil vi enten være ved rota ($i = j$), slik at $\pi_{ij}^0 = NIL$ (rota har ingen forgjenger) eller vi kan ha én kant (i, j) slik at $\pi_{ij}^0 = i$ (forgjengeren til j er i).



På forrige side så vi at det er to tilfeller: k er med i shortest-path eller ikke. Dersom k er med i shortest-path vil $d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1}$, dvs. vekten til shortest-path uten k er større enn vekten til shortest-path med k. Når k er med i shortest-path vil $\pi_{ij}^k = \pi_{kj}^{k-1}$, altså vi finner forgjengeren til j langs shortest-path fra i ved å se på forgjengeren til j langs shortest-path fra k (følger av optimal delstruktur). Når k ikke er med i shortest-path vil $\pi_{ij}^k = \pi_{ij}^{k-1}$, altså vi fjerner k som mulig mellomliggende verteks. Altså:

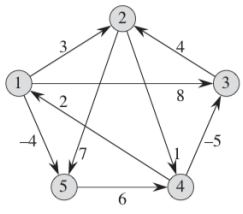
$$\pi_{ij}^k = \begin{cases} \pi_{kj}^{k-1} & \text{hvis } d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1} \\ \pi_{ij}^{k-1} & \text{hvis } d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1} \end{cases}$$

```
FLOYD-WARSHALL'(W)
1  n = W.rows
2  initialize D and Π
3  for k = 1 to n
4      for i = 1 to n
5          for j = 1 to n
6              if d_ij > d_ik + d_kj
7                  d_ij = d_ik + d_kj
8                  π_ij = π_kj
9  return D, Π
```

FLOYD-WARSHALL' – finner shortest-paths løsning

Denne metoden tar inn en $n \times n$ matrise W som gir vekten til kant (i, j) og vil returnere D^n som gir shortest-paths fra verteks i til verktes j og Π som gir forgjengeren til verteks j langs shortest-path fra i. Metoden begynner med å sette n lik antall rader i W. Siden dette er en bottom-up metode vil den initialisere D og Π, som vil si å finne D^0 og Π^0. Deretter vil for-løkken ved linje 3-7 finne D^n og $\Pi^n = \Pi$ ved å løse delproblemene fra bunnen av og opp. For hver startnode og for hver sluttnode vil metoden oppdatere vekten og forgjengeren hvis det er raskere å gå via k. Dette vil være tilfellet dersom vekten til shortest path fra i til j uten å gå via k er større enn hvis den går gjennom k. Altså, ved hver iterasjon vil en verteks legges til settet av mellomliggende vertekser og deretter vil if-setningen undersøke om banen mellom i og j blir kortere enn nåværende korteste bane dersom vi går via denne verteksen. Hvis det er tilfellet vil vi få en ny shortest-path og må dermed oppdatere vekten til banen og forgjengeren til j. Til slutt vil vi ha nådd $k = n$ slik at de mellomliggende verteksene kan være alle vertekser i grafen. Dermed vil D^n gi shortest-paths fra verteks i til verktes j og Π gir forgjengeren til verteks j langs shortest-path fra i.

Figuren viser et eksempel for grafen nedenfor. Siden $n = 5$ (antall rader) vil $D = D^5$ og $\Pi = \Pi^5$. Ved $k = 0$ kan vi se at $d_{11} = 0$ fordi $i = j$, $d_{21} = \infty$ fordi det er ingen kant $(2, 1)$ og $d_{12} = 3$ fordi det er en kant $(1, 2)$ med $w(1, 2) = 3$. Ved $k = 0$ kan vi også se at $\pi_{ij} = i$ dersom det er en kant (i, j) i grafen. Ellers vil $\pi_{ij} = NIL$. Etter første



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

iterasjon vil disse endres for (i, j) dersom den nye verteksen som blir lagt til settet av mellomliggende vertekser gir ny shortest-path fra i til j. Til slutt vil vi få D^5 og Π^5 som returneres.

Transitiv lukning av en rettet graf

Gitt en graf $G = (V, E)$, kan det hende at vi ønsker å vite om G inneholder en bane fra i til j for alle vertekspar $i, j \in V$. Altså, om det er en kobling mellom alle vertekseparene i G. Vi definerer den **transitive lukningen** av G som grafen $G^* = (V, E^*)$, der $E^* = \{(i, j)$: det er en bane fra i til j i $G\}$. Vi kan finne den transitive lukningen på tiden $\Theta(n^3)$ dersom vi setter $w(i, j) = 1$ for alle kanter $(i, j) \in E$ og bruker Floyd-Warshall algoritmen. Dersom det er en bane fra i til j vil $d_{ij} < n$, hvis ikke vil $d_{ij} = \infty$.

Det finnes likevel en mye enklere metode som kan redusere bruk av tid og rom. Denne metoden bruker $\vee$ og $\wedge$ istedenfor min og $+$. **Denne metoden bruker t_{ij}^k som vil være 1 hvis det eksisterer en bane fra i til j i grafen G når alle mellomliggende vertekser er hentet fra $\{1, 2, \dots, k\}$ og 0 hvis det ikke eksisterer en slik bane.** For å finne den transitive lukningen $G^* = (V, E^*)$, vil vi la E^* kun inneholde kanter (i, j) der $t_{ij}^n = 1$. En rekursiv definisjon av t_{ij}^k er:

$$t_{ij}^0 = \begin{cases} 0 & \text{hvis } i \neq j \text{ og } (i, j) \notin E \\ 1 & \text{hvis } i = j \text{ eller } (i, j) \in E \end{cases}$$

Altså, for $k = 0$ vil $t_{ij}^0 = 0$ dersom det er ingen kant mellom i og j, mens $t_{ij}^0 = 1$ hvis $i = j$ eller det er en kant mellom i og j. For $k \geq 1$ vil:

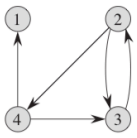
$$t_{ij}^k = t_{ij}^{k-1} \vee \left(t_{ik}^{k-1} \wedge t_{kj}^{k-1}\right)$$

Altså, t_{ij}^k vil være 1 dersom det er en bane fra i til j når mellomliggende vertekser velges fra $\{1, 2, \dots, k-1\}$ eller dersom det er en bane fra i til k og fra k til i når mellomliggende vertekser velges fra $\{1, 2, \dots, k-1\}$.

```
TRANSITIVE-CLOSURE(G)
1  n = |G.V|
2  let T^(0) = (t_ij^(0)) be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          if i == j or (i, j) ∈ G.E
6              t_ij^(0) = 1
7          else t_ij^(0) = 0
8  for k = 1 to n
9      let T^(k) = (t_ij^(k)) be a new n × n matrix
10     for i = 1 to n
11         for j = 1 to n
12             t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
13 return T^(n)
```

<u>TRANSITIVE-CLOSURE – finner transitiv lukning til grafen G</u>
Denne metoden tar inn en graf G og vil returnere den transitive lukningen T^n. Siden det er en bottom-up metode (dynamisk programmering) vil metoden begynne med å initialisere T^0 som er en ny $n \times n$ matrise. Den doble for-løkken ved linje 3-7 vil sørge for at $t_{ij}^0 = 1$ dersom $i = j$ eller $(i, j) \in E$, og at $t_{ij}^0 = 0$ ellers. Deretter vil for-løkken ved linje 8-12 finne T^n ved å løse delproblemene fra bunnen av og opp. For hver iterasjon lager vi en ny matrise T^k. Deretter går vi igjennom alle vertekseparene og lar $t_{ij}^k = 1$ dersom $t_{ij}^{k-1} = 1$ eller $t_{ik}^{k-1} = t_{kj}^{k-1} = 1$. Dvs. det er en bane fra i til j dersom det allerede er en bane fra i til j som ikke går via k eller dersom det er en bane fra i til k og en bane fra k til j som til sammen blir en bane fra i til j. Prosedyren blir gjentatt helt til $k = n$ og vi kan velge alle verteksene som mellomliggende verteks. Metoden returnerer T^n som vil være en matrise av t_{ij}^n verdiene, og hvis $t_{ij}^n = 1$ betyr det at det er en bane mellom i og j. **Kjøretiden er $\boldsymbol{\Theta(n^3)}$**, men logiske operatorer ($\vee$ og $\wedge$) kan utføres raskere og boolean verdier krever mindre plass.

Merk: Denne algoritmen ser om det eksisterer en bane mellom i og j, og ikke om dette er shortest-path



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Figuren viser et eksempel på transitiv lukning av en rettet graf. Ved T^0 kan vi se at $t_{11}^0 = 1$ siden $i = j$, $t_{34}^0 = 0$ siden det er ingen kant $(3,4)$ og $t_{23}^0 = 1$ siden det er en kant $(2,3)$ i grafen. Ved T^1 vil $t_{34}^1 = 0$, fordi det er ingen bane fra 3 til 4 via verteks 1. Ved T^2 derimot vil $t_{34}^2 = 1$, fordi det går en bane fra 3 til 4 via verteks 2. Altså, t_{ij} vil være 1 dersom det går en bane fra i til j via vertekser som er i settet av mellomliggende vertekser. Ved T^2 vil $t_{42}^2 = 0$ fordi den eneste banen fra 4 til 2 går via verteks 3 som enda ikke har blitt lagt til settet av mellomliggende vertekser. Til slutt vil vi få T^4 som er den transitive lukningen til grafen. Her kan vi se at $t_{12} = t_{13} = t_{14} = 0$ fordi det er ingen bane fra 1 til 2, 3 eller 4.

25.3 Johnsons algoritme for sparsomme grafer

Johnsons algoritme kan finne shortest-paths mellom alle vertekspar på tiden $O(V^2 \lg V + VE)$ for en rettet graf uten negative sykler og med vektmatrise $W = (w_{ij})$. For sparsomme grafer ($|E| \ll |V|^2$) vil den være asymptotisk raskere enn Floyd-Warshall algoritmen. Johnsons algoritme vil enten returnere en matrise av vekten til shortest-paths for alle vertekspar eller en beskjed om at grafen inneholder en negativt-vektet syklus. Dette gjør den vha både Dijkstras og Bellman-Ford algoritmen.

Johnsons algoritme bruker en teknikk kalt ***reweighting***, som brukes dersom grafen inneholder kanter med negativ vekt. Hvis alle kantvektene w i grafen $G = (V, E)$ er positive, kan vi finne shortest-paths mellom alle vertekspar vha **Dijkstras algoritme** med hver verteks som kilde. Dersom vi brukes Fibonacci min-heap for å implementere Q vil kjøretiden bli $O(V^2 \lg V + VE)$. **Hvis G inneholder kanter med negativ vekt, men ingen negativt-vektet syklus, kan vi regne ut et nytt sett med kanter som har positiv vekt som gjør at vi kan bruke samme metode.** Det nye settet med kantvekter $\hat{w}$ må tilfredsstille to viktige egenskaper:

1. **For alle vertekspar $u, v \in V$, vil banen p være shortest-path fra u til v med vektfunksjon w kun hvis p også er shortest-path fra med vekstfunksjon $\hat{w}$**
2. **For alle kanter (u, v) vil den nye vekten $\hat{w}(u, v) \geq 0$**

Vi skal se at det tar $O(VE)$ tid for å finne $\hat{w}$.

Bevis av 1. egenskap – Bevare shortest-path ved *reweighting*

Vi bruker δ for å betegne shortest-path vekter funnet vha vektfunksjon w og $\hat{\delta}$ for å betegne shortest-path vekter funnet vha vektfunksjon $\hat{w}$. Gitt en rettet graf $G = (V, E)$ med vektfunksjon w og en funksjon $h: V \to \mathbb{R}$ (verdien til verteksen), vil:

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v)$$

For at egenskap 1 skal gjelde må vi vise følgende:

- p er shortest-path fra v_0 til v_k med vektfunksjon w hvis og bare hvis p er shortest-path med vektfunksjon $\hat{w}$. Altså, $w(\boldsymbol{p}) = \boldsymbol{\delta(v_0, v_k)}$ **kun hvis** $\hat{\boldsymbol{w}}(\boldsymbol{p}) = \hat{\boldsymbol{\delta}}(\boldsymbol{v_0}, \boldsymbol{v_k})$**.** *Bevis:* vi begynner med å vise at $\hat{w}(p) = w(p) + h(v_0) - h(v_k)$:

$$\hat{w}(p) = \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) = \sum_{i=1}^{k} \big(w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)\big)$$

Her kan vi bruke noe som kalles **sum teleskop**, som går ut på at en sum vil kun ha et fast antall begrep etter kansellering. I dette tilfellet vil vi få $h(v_0) - h(v_1) + h(v_1) - h(v_2) + h(v_2) - \cdots - h(v_{k-1}) + h(v_{k-1}) - h(v_k) = h(v_0) - h(v_k)$.

$$\hat{w}(p) = \sum_{i=1}^{k} \big(w(v_{i-1}, v_i)\big) + h(v_0) - h(v_k) = w(p) + h(v_0) - h(v_k)$$

Enhver bane fra v_0 til v_k vil derfor ha vekt $\hat{w}(p) = w(p) + h(v_0) - h(v_k)$. $h(v_0)$ og $h(v_k)$ avhenger ikke av banen, så dersom en bane er kortere enn en annen ved vektfunksjon w, må den også være kortere ved vektfunksjon $\hat{w}$. Altså vil $w(p) = \delta(v_0, v_k)$ kun hvis $\hat{w}(p) = \hat{\delta}(v_0, v_k)$.

***G* vil ha en negativt-vektet syklus med vektfunksjon *w* kun hvis *G* har en negativt-vektet syklus med vektfunksjon $\hat{w}$.** *Bevis:* for en syklus $c = \langle v_0, v_1, \ldots v_k \rangle$ der $v_0 = v_k$ vil: $\hat{w}(c) = w(c) + h(v_0) - h(v_k) = w(c)$. Derfor vil c ha negativt vekt ved w kun hvis den har negativt vekt ved $\hat{w}$.

Bevis av 2. egenskap – Reweighting gir ikke-negative kanter

For alle kanter (u, v) vil den nye vekten $\hat{w}(u, v) \geq 0$. *Bevis:* Gitt en rettet graf $G = (V, E)$ med vektfunksjon w, kan vi lage grafen $G' = (V', E')$ ved å legge til en ny verteks $s \notin V$, slik at $V' = V \cup \{s\}$ og $E' = \{(s, v) : v \in V\}$. Vi legger til $w(s, v) = 0$ for alle vertekser $v \in V$, dvs. en kant fra s til v, med vekt 0. Siden ingen kanter går inn mot s, vil ingen shortest-paths i G' inneholde s, bortsett fra de med s som kildeverteks. Vi antar at G og G' har ingen negativt-vektede sykluser og lar $\boldsymbol{h(v) = \delta(s, v)}$ for $v \in V'$. **Triangel ulikheten ($\delta(s, v) \leq \delta(s, u) + w(u, v)$) gir at $h(v) \leq h(u) + w(u, v)$ for alle kanter $(u, v) \in E'$. Derfor vil $\hat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$** og den andre egenskapen er tilfredsstilt.

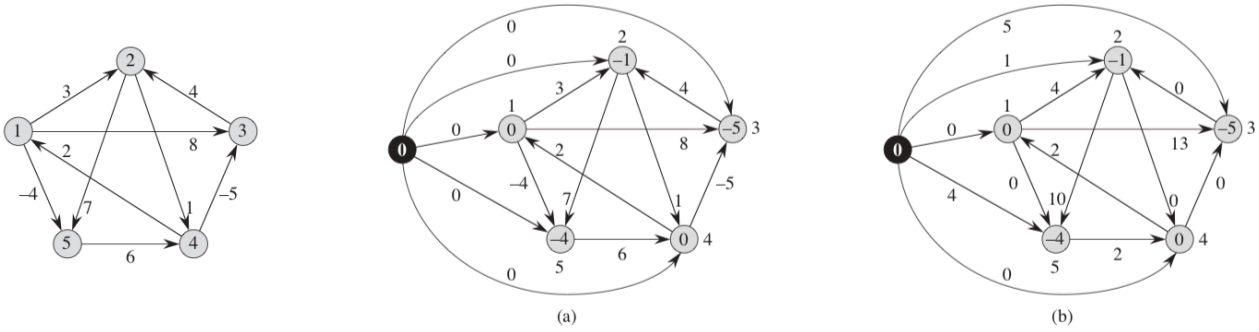
Utregning av nye vekter

Vi har vist at det nye settet med kantvekter $\hat{w}$ tilfredsstiller de to egenskapene, og kan dermed se hvordan vi kan finne $\hat{w}(u, v)$. Dersom vi har gitt en rettet graf $G = (V, E)$ med vektfunksjon w, kan vi finne $\hat{w}$ ved å se på:

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v)$$

der **$h(v) = \delta(s, v)$ og $h(u) = \delta(s, u)$.** På figur a kan vi se grafen G' med den originale vektfunksjonen w og den nye verteksen s som er markert i svart. Legg merke til at det går kanter fra s til alle verteksene i grafen og disse har vekt 0. For hver verteks v må vi finne $h(v) = \delta(s, v)$. For eksempel for $v = 5$ kan vi se at korteste bane er via 1, siden dette gir $h(5) = \delta(s, 5) = -4 < 0$, mens for $v = 4$ er korteste bane direkte fra s, slik at $h(4) = \delta(s, 4) = 0$. På figur b kan vi se grafen G' med de nye vektene. De nye vektene er:

- Kant $(1, 2)$: $\hat{w}(1, 2) = w(1, 2) + h(1) - h(2) = 3 + 0 - (-1) = 4$
- Kant $(1, 3)$: $\hat{w}(1, 3) = w(1, 3) + h(1) - h(3) = 8 + 0 - (-5) = 13$
- …



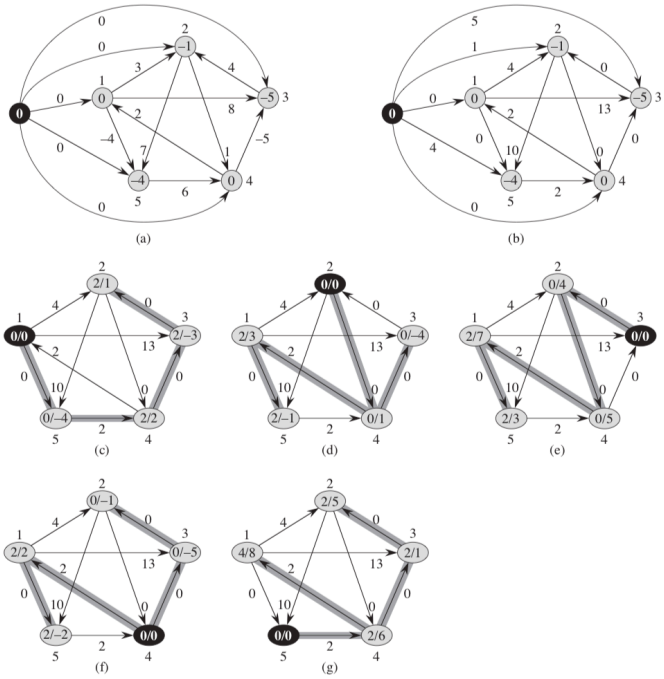
Johnsons algoritme – løsning på APSP problemet

Johnsons algoritme for å finne all-pair shortest-paths bruker både Bellman-Ford og Dijkstras algoritme som delrutiner, og den antar at kantene er lagret i nabolister.

```
JOHNSON(G, w)
1  compute G', where G'.V = G.V ∪ {s},
       G'.E = G.E ∪ {(s, v) : v ∈ G.V}, and
       w(s, v) = 0 for all v ∈ G.V
2  if BELLMAN-FORD(G', w, s) == FALSE
3       print "the input graph contains a negative-weight cycle"
4  else for each vertex v ∈ G'.V
5            set h(v) to the value of δ(s, v)
                 computed by the Bellman-Ford algorithm
6       for each edge (u, v) ∈ G'.E
7            ŵ(u, v) = w(u, v) + h(u) − h(v)
8       let D = (d_uv) be a new n × n matrix
9       for each vertex u ∈ G.V
10           run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ G.V
11           for each vertex v ∈ G.V
12                d_uv = δ̂(u, v) + h(v) − h(u)
13      return D
```

JOHNSON – finner shortest-paths for alle vertekspar

Denne metoden tar inn en rettet graf $G = (V, E)$ og en vektfunksjon w, og vil returnere en matrise D som gir vekten til shortest-paths for alle vertekspar i G. Ved linje 1 vil vi legge til en ny verteks s og finner $G' = (V', E')$ ved å legge s til V og kantene (s, v) til E med $w(s, v) = 0$. Ved linje 2 bruker vi Bellman-Ford algoritmen for å undersøke om G' inneholder en negativt-vektet syklus. Hvis ikke vil for-løkken ved linje 4 utføres og den vil gå igjennom alle verteksene. Den setter $h(v) = \delta(s, v)$ funnet vha Bellman-Ford algoritmen og vil deretter finne den nye vekten $\hat{w}(u, v)$ for alle vertekspar. Dermed har vi utført en *reweighting* og kan være sikker på at alle kantvektene er positive. Derfor kan vi kalle på Dijkstras algoritme på hver av verteksene i V, slik at vi finner $\hat{\delta}(u, v)$ (her vil u være verteksen vi ser på fra V og v er alle verteksene i nabolisten til u). Merk: vi må bruke $\hat{w}$ som vektfunksjon siden det er den som garanterer at kantvektene er positive. Deretter er vi nødt til å omdanne disse vektene til vektfunksjon w, noe vi gjør ved å gå igjennom alle verteksene og deretter bruke at $\delta(u, v) = \hat{\delta}(u, v) - h(u) + h(v)$. Dermed vil vi ha funnet $D = (d_{uv})$ for alle vertekspar $u, v \in V$, og dette blir returnert. **Kjøretiden er $O(V^2 \lg V + VE)$** dersom min-prioritetskøen i Dijkstras algoritme blir implementert med en Fibonacci heap. Dette er asymptotisk raskere enn Floyd-Warshall dersom grafen har få kanter ($|E| \ll |V|^2$).



Figuren viser et eksempel på Johnson algoritme. Figur a til b viser reweighting prosessen forklart på forrige side. Som vi kan se vil dette føre til at alle kantene får positiv vekt. Dermed kan vi kalle på Dijkstras algoritme med hver av verteksene som kilde. Det grå feltet markerer shortest-path treet og inni verteksen ser vi $\hat{\delta}(u, v)/\ \delta(u, v)$. For hver iterasjon ser vi på u lik verteksen markert i svart, og da vil Dijkstras algoritme returnere $\hat{\delta}(u, v)$ for alle de andre verteksene. Dermed kan vi finne $\delta(u, v)$ ved å se på $\delta(u, v) = \hat{\delta}(u, v) - h(u) + h(v)$, der $h(u)$ og $h(v)$ hentes fra figur b som viser shortest-path fra s til verteksene. På figur c har vi $u = 1$, slik at Dijkstras algoritme finner for eksempel $\hat{\delta}(1, 5) = 0$. Dermed vil $\delta(1, 5) = \hat{\delta}(1, 5) - h(1) + h(5) = 0 - 0 + (-4) = -4$. På figur d har vi $u = 2$, slik at Dijkstras algoritme finner for eksempel $\hat{\delta}(2, 5) = 2$. Dermed vil $\delta(2, 5) = \hat{\delta}(2, 5) - h(2) + h(5) = 2 - (-1) + (-4) = -1$. Til slutt vil vi finne $\delta(u, v)$ for alle verteksparene $u, v \in V$.

Forelesning 12 – Maksimal flyt

Maksimal flyt er et stort skritt i retning av generell lineær optimering (såkalt lineær programmering). Her ser vi på to tilsynelatende forskjellige problemer, som viser seg å være duale av hverandre, noe som hjelper oss med å finne en løsning. Læringsmålene er:

- Kunne definere *flytnettverk*, *flyt* og *maks-flyt*-problemet
- Kunne håndtere *antiparallelle kanter* og *flere kilder* og *sluk*
- **Kunne definere *residualnettverket* til et nettverk med en gitt flyt**
- Forstå hvordan man kan *oppheve* (*cancel*) flyt
- Forstå hva en *forøkende sti* (*augmenting path*) er
- Forså hva *snitt*, *snitt-kapasitet* og *minimalt snitt* er
- **Forstå maks-flyt/min-snitt teoremet**
- Forstå FORD-FULKERSON-METHOD og FORD-FULKERSON
- Vite at FORD-FULKERSON med BFS kalles *Edmonds-Karp*-algoritmen
- Forstå hvordan maks-flyt kan finne en *maksimum bipartitt matching*
- Forstå *heltallsteoremet* (*integrality theorem*)

Kapittel 26 – Maksimal flyt

En rettet graf kan også tolkes som et "flytnettverk" og brukes for å beskrive en flyt av materiale fra kilden til sluken (forbrukspunktet). Produksjons- og forbruksraten er den samme. Materialflyten vil være hvor raskt materialet beveger seg ved et punkt i systemet. Hver rettet kant i flytnettverket kan ses på som en ledning for materialet, der hver ledning har en gitt **kapasitet** (maksimum rate for materialflyt). Vertekser er ledende kryss som materialet vil flyte gjennom uten å akkumulere. Flytraten inn og ut av verteksen er derfor den samme, noe som kalles **flytbevaring**. I **maks-flyt-problemet** ønsker vi å finne den største raten for flyt av materiale fra kilden til sluken uten å overgå **kapasitetsgrensene**. Vi skal se på algoritmer for å løse dette problemet.

26.1 Flytnettverk

Et **flytnettverk** $G = (V, E)$ er en rettet graf der hver kant $(u, v) \in E$ har en positiv **kapasitet** $c(u, v) \geq 0$. **Dersom E inneholder kanten (u, v) vil den ikke inneholde kanten (v, u) i den motsatte retningen**. Dersom $(u, v) \notin E$ vil $c(u, v) = 0$. **Merk at løkker ikke er tillatt, men grafen kan ha sykler!** To viktige vertekser er **kilden** s og **sluken** t, og vi antar at alle vertekser ligger på en bane mellom disse ($s \rightsquigarrow v \rightsquigarrow t$). Derfor er grafen koblet og $|E| \geq |V| - 1$.

En **flyt** i G er en reell-verdi funksjon $f : V \times V \to \mathbb{R}$ som tilfredsstiller følgende egenskaper:

- **Kapasitetsbegrensning** – for alle $u, v \in V$, vil $\mathbf{0} \leq \boldsymbol{f}(\boldsymbol{u}, \boldsymbol{v}) \leq \boldsymbol{c}(\boldsymbol{u}, \boldsymbol{v})$. Flyten fra en verteks til en annen kan ikke være negativ eller overgå den gitte kapasiteten.
- **Flytbevaring** – for alle $u \in V - \{s, t\}$ (dvs. alle vertekser uten s og t) vil $\sum_{v \in V} \boldsymbol{f}(\boldsymbol{v}, \boldsymbol{u}) = \sum_{v \in V} \boldsymbol{f}(\boldsymbol{u}, \boldsymbol{v})$. Den totale flyten inn i en verteks, som ikke er kilden eller sluken, må være lik den totale flyten ut av denne verteksen. Flyt inn = flyt ut.

Flytnettverk: Rettet graf $G = (V, E)$

- Kapasiteter $c(u, v) \geq 0$
- Kilde og sluk $s, t \in V$
- $v \in V \implies s \rightsquigarrow v \rightsquigarrow t$
- Ingen løkker (*self-loops*)
- $(u, v) \in E \implies (v, u) \notin E$
- $(u, v) \notin E \implies c(u, v) = 0$

Flyt: En funksjon $f : V \times V \to \mathbb{R}$

- $0 \leq f(u, v) \leq c(u, v)$
- $u \neq s, t \implies \sum_v f(v, u) = \sum_v f(u, v)$

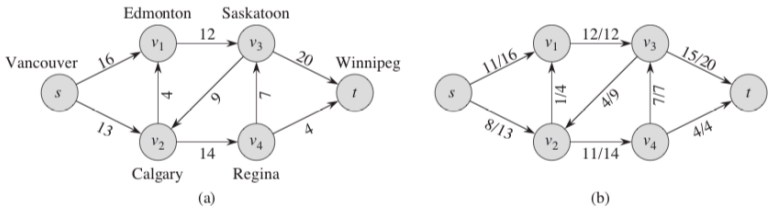
Flytverdi: $|f| = \sum_v f(s, v) - \sum_v f(v, s)$

Den ikke-negative mengden $f(u, v)$ kalles flyten fra verteks u til verteks v. Verdien til en flyt er definert som:

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

dvs. den totale flyten ut av en kilde, minus flyten inn i kilden. Som regel vil ikke flytnettverket ha noen kanter inn mot kilden, så derfor vil $|f| = \sum_{v \in V} f(s, v)$. I **maks-flyt-problemet får vi et flytnettverk G med kilde s og sluk t og vil finne den maksimale verdien til flyten**.

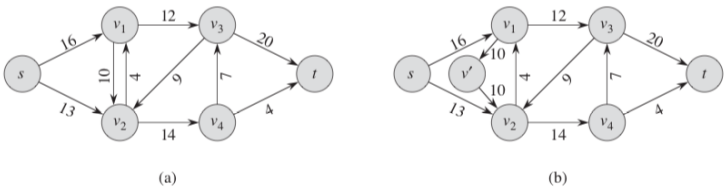
Et eksempel på flyt



Et flytnettverk kan modellere et fraktproblem vist på figur a, der s er fabrikken og t er et varehus for lagring. Siden lastebilene reiser bestemte ruter mellom byer (vertekser) og har begrenset kapasitet, kan maksimalt $c(u, v)$ kasser fraktes mellom byene u og v. Vi vil finne det største antall kasser p som kan sendes per dag, slik at dette bestemmer mengden som produseres. Dette kan beskrives som en flyt siden antall kasser som fraktes er underlagt kapasitetsbegrensningen og flyten er bevart siden antall kasser som fraktes inn og ut av en mellomliggende by må være like. På figur b kan vi se en flyt f i G med verdien $|f| = 19$. Hver kant er merket med $f(u, v)/c(u, v)$, der $f(u, v) \leq c(u, v)$. Legg merke til at $|f| = f(s, v_1) + f(s, v_2) = 11 + 8 = 19$ og for hver kant vil flyten inn være lik flyten ut. Vi skal senere se hvordan vi finner $f(u, v)$.

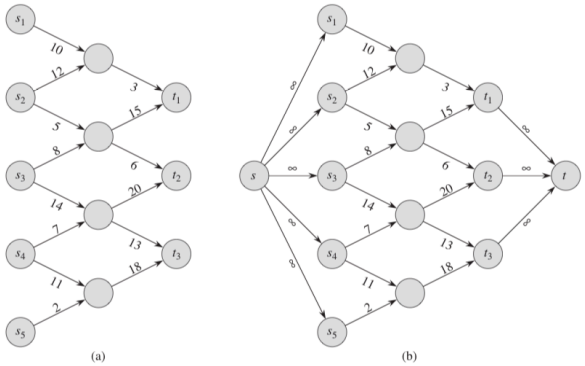
Nettverk med antiparallelle kanter

Anta at selskapet ordner en ny rute fra v_1 til v_2 for å danne nettverket på figur a. Dette nettverket vil bryte den originale antagelsen om at $(v_1, v_2) \in E$ gir at $(v_2, v_1) \notin E$. Disse to kantene kalles **antiparallelle**. I et slikt tilfelle må vi omforme nettverket ved å splitte den ene kanten vha en node v'(se figur b). Kapasiteten til de to nye kantene blir satt lik kapasiteten til den originale kanten.



Nettverk med flere kilder og sluker

Et maks-flyt-problem kan ha flere kilder og sluker, istedenfor kun en av hver (figur a). Et slikt flytnettverk kan reduseres til et flytnettverk med kun én kilde og én sluk ved å legge til en **superkilde** s og en **supersluk** t. Disse er koblet til de originale kildene og slukene via rettete kanter med uendelig kapasitet. Enhver flyt i nettverk a vil korrespondere til en flyt i nettverk b (og motsatt), siden superkilden vil gi så mye flyt som er ønsket av kildene og supersluken vil forbruke så mye flyt som er ønsket av slukene.



26.2 Ford-Fulkerson metode

Fold-Fulkerson metoden kan brukes for å løse maks-flyt-problemet ("metode" fordi det er flere implementasjoner med ulik kjøretid). Denne metoden avhenger av tre viktige egenskaper: **residualnettverket**, **forøkende sti** og **snitt**.

Fold-Fulkerson metoden vil iterativt øke verdien til flyten. Ved begynnelsen vil $f(u, v) = 0$ for alle $u, v \in V$, slik at den initiale flytverdien blir 0. Ved hver iterasjon vil vi øke flytverdien i G ved å finne en forøkende sti (*augmenting path*) i et assosiert residualnettverk G_f. Når vi kjenner kantene til den forøkende stien i G_f, kan vi lett finne de spesifikke kantene i G der vi kan endre flyten slik at flytverdien økes. Hver iterasjon vil altså øke flytverdien, mens flyten til en spesifikk kant i G kan både øke og minke. Vi vil gjentatt øke flytverdien helt til residualnettverket ikke har flere forøkende stier. **Max-flow min-cut teoremet** (mer senere) gir at denne prosessen ved terminering vil gi maksimal flyt mellom kilde og sluk.

```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p in the residual network G_f
3      augment flow f along p
4  return f
```

FORD-FULKERSON-METHOD – finner maks-flyt
Denne metoden tar inn en graf G, en kildeverteks s og en slukverteks t, og vil returnere flytfunksjonen f. Metoden vil først initialisere funksjonen ved å sette $f(u, v) = 0$ for alle $u, v \in V$. Deretter vil den øke flyten langs p så lenge det eksisterer en forøkende bane p i residualnettverket G_f.

Residualnettverk

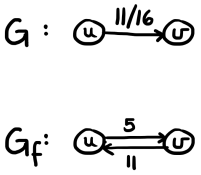
Gitt et flytnettverk G og en flyt f, vil residualnettverket G_f bestå av kanter med kapasitet som representerer hvordan vi kan endre flyten til kantene i G. En kant i flytnettverket kan tillate en ekstra mengde flyt som er lik kantens kapasitet minus flyten til kanten (dvs. det som er igjen av kapasiteten). **Dersom denne mengden er positiv vil vi plassere kanten i G_f med en residualkapasitet gitt av $c_f(u, v) = c(u, v) - f(u, v)$.** Dersom flyten til en kant i G er lik kapasiteten, vil $c_f = 0$ og kanten vil ikke være i G_f.

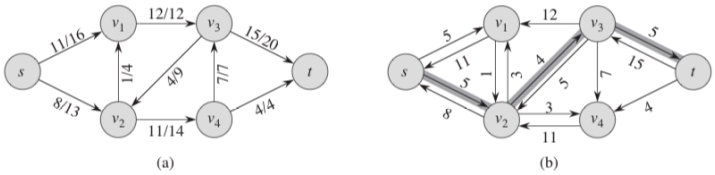
For at det skal være mulig å redusere flyten til kanter i G, må residualnettverket også inneholde kanter som ikke er i G. **For å redusere flyten til en kant, kan vi øke flyten i motsatt retning**. Dersom G inneholder en kant (u, v) med flyt $f(u, v)$, kan vi redusere flyten ved å legge til en kant (v, u) i G_f. **Flyten til (u, v) kan ikke reduseres med mer enn $f(u, v)$, så derfor vil residualkapasitet være $c_f(v, u) = f(u, v)$.** Merk: residual-kapasiteten gir hvor mye du kan endre flyten til en kant. Disse reverskantene lar algoritmen sende tilbake flyt som allerede har blitt sendt langs en kant. Residualkapasiteten er gitt av:

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{hvis } (u, v) \in E \\ f(v, u) & \text{hvis } (v, u) \in E \\ 0 & \text{ellers} \end{cases}$$

Vi ser på residualkapasiteten til kanten (u, v) i G_f. Dersom vi ikke har en slik kant i G vil residualkapasiteten være 0, siden det er ingen flyt. Dersom vi har en kant (u, v) i G vil vi kunne øke flyten i denne retningen med $c_f = c(u, v) - f(u, v)$. Dersom vi har en kant (v, u) i G med flyt $f(v, u)$ vil vi kunne redusere flyten i denne retningen med $f(v, u)$ (her blir det redusere siden vi har kant (u, v) i G_f og kant (v, u) i G, altså motsatt retning). Dette er ekvivalent med å si at du **kan ikke øke flyten mer enn kapasiteten, og du kan ikke redusere flyten slik at den blir mindre enn 0. Du øker flyten ved å ha en kant med samme retning i residualnettverket, og du reduserer flyten ved å ha en kant med motsatt retning i residualnettverket.**

For eksempel kan vi ha en kant (u, v) i G, slik at $c(u, v) = 16$ og $f(u, v) = 11$. Flyten til denne kanten kan økes ved å ha en kant (u, v) i G_f med residualkapasitet $c_f(u, v) = 16 - 11 = 5$, dvs. flyten kan økes med 5 uten å overgå kapasiteten. Flyten til denne kanten kan minkes ved å ha en kant (v, u) i G_f med residualkapasitet $c_f(v, u) = f(u, v) = 11$, dvs. flyten kan reduseres med 11 uten å bli negativ.





Gitt et flytnettverk $G = (V, E)$ og en flyt f, vil **residualnettverket** være $G_f = (V, E_f)$, der $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$, altså hver **residualkant** kan tillate en flyt som er større enn 0. Figur a viser flytnettverk G og figur b viser det korresponderende residualnettverket G_f. Her kan vi se at (v_1, v_2) ikke er en kant i G_f siden flyten til denne kanten er lik kapasiteten. Legg merke til at alle kantene har en revers kant i G_f fordi det skal være mulig å redusere flyten til alle disse. Derfor vil $|E_f| \leq 2|E|$. Bortsett fra at residualnettverket kan inneholde antiparallelle kanter, vil det ha samme egenskaper som et flytnettverk.

Flyten i residualnettverket (f') gir informasjon om hvordan flyten til det originale flytnettverket (f) kan endres. **Forøkningen** av flyten f som følger av f' er funksjonen:

$$(f \uparrow f')(u, v) = \begin{cases} f(u, v) + f'(u, v) - f'(v, u) & \text{hvis } (u, v) \in E \\ 0 & \text{ellers} \end{cases}$$

Her vil $(f \uparrow f')(u, v)$ være den "nye" flyten til kant (u, v) som følger av at f' øker f. Vi øker flyten $f(u, v)$ med mengden gitt av $f'(u, v)$ og redusere flyten med mengden gitt av $f'(v, u)$, siden **flyt på en revers kant i residualnettverket innebærer redusert flyt i det originale flytnettverket**. Flyt på den reverse kanten i et residualnettverk kalles **oppheving (*cancellation)*** av flyten i det originale flytnettverket.

Lemma 1 – $f \uparrow f'$ er en flyt

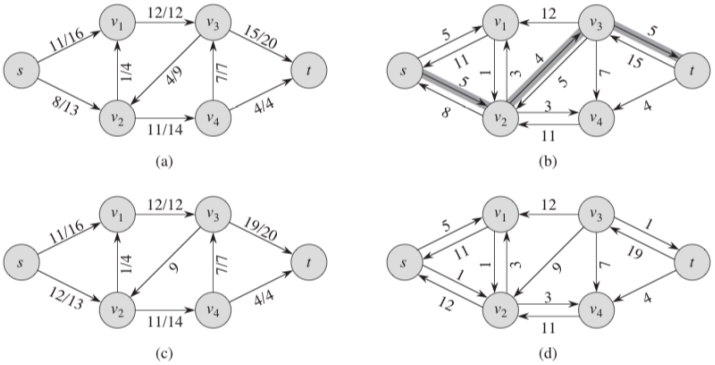
Funksjonen $f \uparrow f'$ vil være den "nye" flyten i G med verdi $|f \uparrow f'| = |f| + |f'|$. Altså, ved å øke eller redusere flyten bestemt av residualnettverket vil kanten få en ny flyt i G. For å vise at $(f \uparrow f')(u, v)$ er en flyt, må vi bevise at den oppfyller de to kravene (s. 138):

- **Kapasitetsbegrensning** – for alle $u, v \in V$, vil $0 \leq (f \uparrow f')(u, v) \leq c(u, v)$.
- **Flytbevaring** – for alle $u \in V - \{s, t\}$ vil $\sum_{v \in V}(f \uparrow f')(v, u) = \sum_{v \in V}(f \uparrow f')(u, v)$.

Dette innebærer en del regning (s 718-719 i boka).

Forøkende baner

En forøkende bane p er en simpel bane fra s til t i residualnettverket G_f. Vi kan øke flyten til kanten (u, v) i den forøkende banen med opptil $c_f(u, v)$ uten å bryte kapasitetsbegrensningen til kanten (u, v) eller (v, u) som er i det originale flytnettverket G. På figur b vil den gråe banen være en forøkende bane, og som vi kan se vil den minste residualkapasiteten være $c_f(v_2, v_3) = 4$. Derfor kan vi øke flyten gjennom hver kant langs denne banen med 4 enheter uten å bryte kapasitetsbegrensningen (figur c). Denne økningen vil gjøre at $f(v_2, v_3) = c(v_2, v_3)$ i G, og dermed vil $c_f(v_2, v_3) = 0$, slik at det er ingen kant (v_2, v_3) i G_f. De to andre kantene har større residualkapasitet og kan derfor videre økes.



Residualkapasiteten til en forøkende bane p er den maksimale mengden økning som er tillatt for alle kantene langs den forøkende banen. Dette er altså den største økningen som er mulig for alle kantene langs p, og vil være gitt av den minste residualkapasiteten langs banen, altså:

$$c_f(p) = \min\{c_f(u,v) : (u,v) \text{ er en kant langs } p\}$$

Lemma 2 – definisjon av f_p og $(f \uparrow f_p)$ er en ny flyt nærmere maksimum

Dersom $G = (V, E)$ er et flytnettverk, f er en flyt i G og p er en forøkende bane i G_f, så vil flyten f_p i G_f være gitt av:

$$f_p(u,v) = \begin{cases} c_f(p) & \text{hvis } (u,v) \text{ er på } p \\ 0 & \text{ellers} \end{cases}$$

Flyten til en kant (u, v) i G_f vil altså være $c_f(p)$ dersom denne kanten er langs den forøkende banen, **Dette betyr at kanten (u, v) i G endres dersom denne kanten er langs den forøkende banen i G_f, og mengden den endres bestemmes av den minste residualkapasiteten langs p.** I figuren på forrige side så vi at $c_f(p) = 4$ slik at flyten til alle kantene langs p ble økt med 4. **Dersom vi øker flyten f med f_p, vil vi få en ny flyt i G som er nærmere maksimum**. Altså, vil $f \uparrow f_p$ være en flyt i G med verdien $|f \uparrow f_p| = |f| + |f_p| > |f|$. For eksempel på figuren på forrige side kan vi se at $(f \uparrow f_p)(s, v_2) = 12 > f(s, v_2) = 8$ som er nærmere $c(s, v_2) = 13$.

Snitt hos flytnettverk

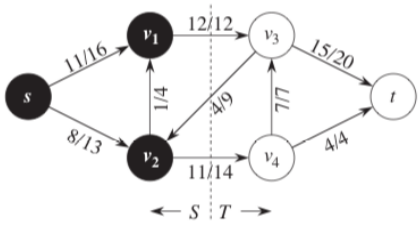
Ford-Fulkerson metoden vil gjentatt øke flyten langs forøkende baner helt til den har funnet en maksimal flyt. **Max-flow min-cut teoremet gir at metoden har funnet maksimum flyt når den terminerer**. For å forstå dette teoremet, må vi først se på snittet av et flytnettverk. **Et snitt (S, T) av et flytnettverk $G = (V, E)$ er en partisjon av V inn i S og $T = V - S$,** slik at $s \in S$ og $t \in T$. Hvis f er en flyt, så vil **nettoflyten $f(S, T)$ på tvers av snittet (S, T)** være gitt av:

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$

Dvs. flyten over snittet er gitt av den totale flyten fra kanter som peker mot T minus den totale flyten fra kanter som peker mot S. **Kapasiteten til snittet (S, T) er:**

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

Dvs. kapasiteten til snittet er gitt av summen av kapasiteten til kantene som krysser snittet og peker mot T. **Et minimum snitt av et nettverk er et snittet som gir minst mulig kapasitet, altså $c(S, T)$ er minst mulig.** Merk: det er en asymmetri mellom definisjonen av flyt og kapasitet. For kapasitet ser vi kun på kapasiteten til kanter som går fra S til T, og ignorer kanter i revers retning. For flyt ser vi på flyten som går fra S til T, minus flyten som går i den reverse retningen. Vi skal senere se hvorfor.



Figuren viser et eksempel med snittet $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$. I dette tilfellet vil nettoflyten og kapasiteten til snittet være:

$$f(S,T) = f(v_1,v_3) + f(v_2,v_4) - f(v_3,v_2) = 12 + 11 - 4 = 19$$
$$c(S,T) = c(v_1,v_3) + c(v_2,v_4) = 12 + 14 = 26$$

Lemma 3 – nettoflyten over et snitt er lik flytverdien

Flytverdien $|f|$ i et flytnettverk er gitt av flyten over snittet:

$$f(S,T) = |f|$$

Dette kan bevises vha definisjonen til $|f|$ og flytbevaring (se s. 722 i boka).

Korollar 4 – flytverdien er begrenset av kapasiteten til snittet

Flytverdien $|f|$ i et flytnettverk er begrenset ovenfra av kapasiteten til ethvert snitt av G:

$$|f| \leq c(S,T)$$

Dette kan vises ved å bruke definisjonen av $f(S,T)$ og $c(S,T)$, og at $f(u,v) \leq c(u,v)$:

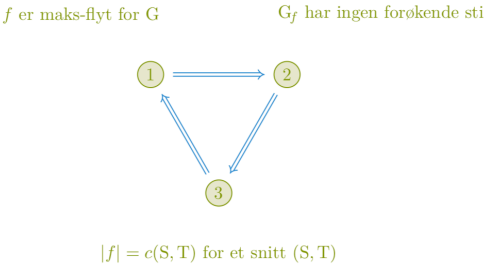
$$|f| = f(S,T) = \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) \leq \sum_{u\in S}\sum_{v\in T} f(u,v) \leq \sum_{u\in S}\sum_{v\in T} u(u,v) = c(S,T)$$

Max-flow min-cut teoremet

Max-flow min-cut teoremet

Hvis f er en flyt i flytnettverket $G = (V,E)$ med kilde s og sluk t, vil følgende forhold være gjensidig gyldige:

1. **f er maksimum flyten i G**
2. **Residualnettverket G_f inneholder ingen forøkende baner**
3. **$|f| = c(S,T)$ for et snitt (S,T) av G**



Dette teoremet sier altså at dersom f er maksimum flyten vil det bety at det er ingen forøkende baner i G_f og verdien til f er lik kapasiteten til et snitt (S,T). For å bevise dette teoremet må vi vise at de tre forholdene er gyldige:

- **1 ⇒ 2** kan bevises ved selvmotsigelse: vi antar at f er en maksimum flyt, men at G_f har en forøkende bane p. Da vil lemma 2 gi at $|f \uparrow f_p| > |f|$, altså flyten som følge av forøkningen er større enn flytverdien til f, noe som er en motsigelse på antagelsen om at f er en maksimum flyt.
- **2 ⇒ 3**: vi antar at G_f har ingen forøkende bane, altså ingen bane fra s til t. Et snitt vil dele V inn i S og $T = V - S$, slik at $s \in S$ og $t \in T$. Vi har et vertekspar $u \in S$ og $v \in T$, og siden det er ingen forøkende bane i G_f vil det ikke kunne være noen kant (u,v) over snittet, altså $c_f(u,v) = 0$. Hvis $(u,v) \in E$ må derfor $f(u,v) = c(u,v)$ og hvis $(v,u) \in E$ må $f(v,u) = 0$ (se definisjonen av c_f).

Dette er eksempel på dualitet

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{hvis } (u,v) \in E \\ f(v,u) & \text{hvis } (v,u) \in E \\ 0 & \text{ellers} \end{cases}$$

Dersom ingen av disse er i E vil $c_f(u,v) = 0$. Dersom vi bruker at $f(u,v) = c(u,v)$ og $f(v,u) = 0$, får vi:

$$|f| = f(S,T) = \sum_{u \in S}\sum_{v \in T} f(u,v) - \sum_{u \in S}\sum_{v \in T} f(v,u) = \sum_{u \in S}\sum_{v \in T} c(u,v) - \sum_{u \in S}\sum_{v \in T} 0 = c(S,T)$$

- **3 ⇒ 1:** korollar 4 gir at $|f| \leq c(S,T)$ for alle snitt (S,T). Derfor vil $|f| = c(S,T)$ bety at f er en maksimum flyt.

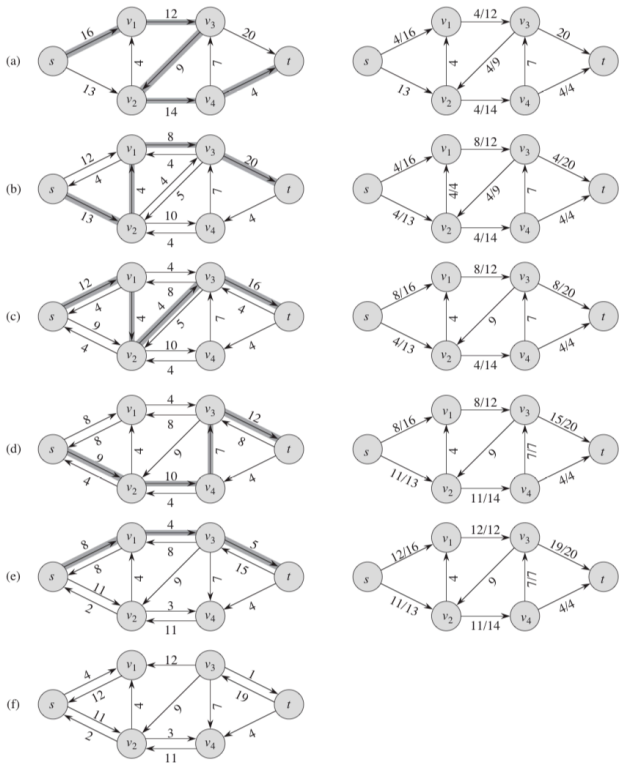
Den grunnleggende Ford-Fulkerson metoden

I hver iterasjon av Ford-Fulkerson metoden vil vi finne en forøkende sti p og bruke denne til å modifisere flyten f. Vi vil erstatte f med flyten $f \uparrow f_p$, slik at vi får en ny flyt med verdi $|f| + |f_p|$. Dette blir gjentatt så lenge det finnes forøkende sti fra kilden til sluken i G_f. En vanlig implementasjon er å finne den forøkende stien, videre finne flaskehalsen (minimal residualkapasitet) og deretter øke flyten langs stien med denne verdien. Denne økningen kan gjøres ved å oppdatere flytattributten $(u,v).f$ til alle kantene langs den forøkende stien.

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min {c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
```

FORD-FULKERSON – finner maksimal flyt

Denne metoden tar inn et flytnettverk G med kilde s og sluk t og vil returnere den maksimale flyten gitt av f attributtene til kantene. Metoden begynner med å sette flyten til alle kantene lik 0. Dette vil gjøre at $c_f(u,v) = c(u,v)$ i residualnettverket. Så lenge det eksisterer en forøkende sti i residualnettverket vil residualkapasiteten til denne ($c_f(p)$) settes lik den minste residualkapasiteten langs stien. Dette gir hvor mye flyten skal økes hos kantene langs den forøkende banen. Metoden vil gå igjennom alle kantene langs den forøkende stien, og disse vil **enten være en fremover kant (u,v) eller en revers kant (v,u) i G** (hvis ikke vil ikke (u,v) være i G_f). Hvis G inneholder en kant (u,v) skal flyten til denne økes med $c_f(p)$, mens hvis G inneholder en kant (v,u) skal flyten reduseres med $c_f(p)$. Dermed blir prosedyren gjentatt helt til det ikke lenger er noen forøkende bane i G (dvs. vi har nådd kapasitetsbegrensningen til bestemte kanter). Dermed har vi oppnådd maksimal flyt som er representert i f attributtene til kantene i G.

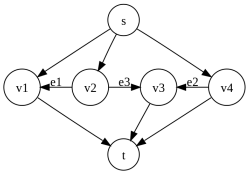


Figuren viser resultatet ved hver iterasjon, der venstre graf er G_f og høyre er resultatet i G etter forøkningen. På figur a ser vi G_f der $c_f(u,v) = c(u,v)$ som følger av at $f(u,v)$ i G initialiseres til 0. På figur a kan vi også se at den forøkende stien er $\langle s, v_1, v_3, v_2, v_4, t \rangle$, slik at $c_f(p) = 4$ (den minste kapasiteten er $c_f(v_4,t) = 4$). Siden alle kantene langs den forøkende banen er fremover kanter i G, vil flyten til alle disse kantene økes med 4 (figur b).

Denne økningen i flyten til kantene (u, v) langs den forøkende banen fører til at $c_f(u, v)$ reduseres med 4 og at $c_f(v, u) = 4$, siden flyten til disse kantene nå kan reduseres med 4 (de er ikke lenger 0). Legg merke til at $c_f(v_4, t) = 0$, siden denne var minimum residualkapasitet og den nye flyten vil derfor være lik kapasiteten. Denne prosedyren blir gjentatt helt til vi når situasjonen i f) der det er ingen forøkende, sammenhengende bane mellom s og t i G_f. Da vil metoden være ferdig, og flyten til alle kantene i f har blitt maksimert.

Analyse av Ford-Fulkerson

Kjøretiden til Ford-Fulkerson vil avhenge av hvordan vi finner den forøkende banen p. **Dersom vi implementerer flytnettverket som en rettet graf der kantene er (u, v) eller (v, u) (ikke begge deler), kan vi bruke BFS for å finne den forøkende banen på tiden $O(E)$** (går maksimalt igjennom alle kantene fra kilden til sluken)**.** Derfor vil hver iterasjon av while-løkken ta tiden $O(E)$, og neste steg er å finne antall iterasjoner. Maks-flyt problemet vil ofte ha kapasiteter som er heltall og i disse tilfellene vil Ford-Fulkerson metoden gi en maksimal flyt. Dersom kapasitetene er rasjonelle tall, kan vi bruke en skaleringstransformasjon for å omforme de til heltall. **Hvis f^* betegner maksimal flyt i dette transformerte nettverket**, så vil while-loopen utføres maks $|f^*|$ ganger, siden flytverdien øker med minst én enhet per iterasjon helt til den når maks verdi. **Siden hver iterasjon tar tiden $O(E)$ og det utføres $O(|f^*|)$ antall iterasjoner, vil total kjøretid til Ford-Fulkerson metoden være $O(E|f^*|)$.**

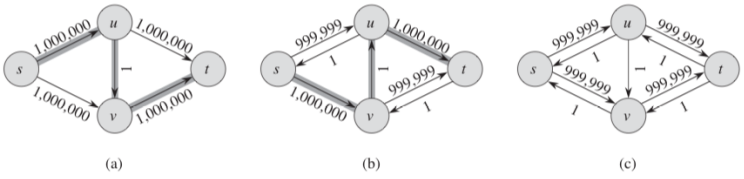


Step	Augmenting path	Sent flow	Residual capacities		
			e_1	e_2	e_3
0			$r^0 = 1$	r	1
1	$\{s, v_2, v_3, t\}$	1	r^0	r^1	0
2	p_1	r^1	r^2	0	r^1
3	p_2	r^1	r^2	r^1	0
4	p_1	r^2	0	r^3	r^2
5	p_3	r^2	r^2	r^3	0

Husk: irrasjonale tall er tall som ikke kan skrives som brøk.

Merk: for irrasjonelle tall vil det ikke eksistere noen skaleringstransformasjon for å omforme kapasitetene til heltall! **Dersom kapasitetene er irrasjonelle tall kan det derfor hende at metoden ikke terminerer**, fordi de forøkende banene kan brukes uendelig mange ganger uten at residualkapasitetene til kantene går mot 0 (se figur). I dette tilfellet vil flyten øke ved hver iterasjon, men den konvergerer ikke til en maksimum flytverdi.

Worst-case Ford-Fulkerson



Hvis kapasitetene er heltall og den optimale flytverdien $|f^*|$ er liten, vil Ford-Fulkerson metoden ha god kjøretid. Figuren viser et eksempel på hva som kan skje dersom $|f^*| = 2\,000\,000$ er stor og den forøkende banen velges vilkårlig. Den første forøkende banen er $s \to u \to v \to t$, slik at $c_f(p) = 1$. Dermed vil residualkapasitetene reduseres med 1, (u, v) fjernes fra residualnettverket og (v, u) blir lagt til. Den neste forøkende banen er $s \to v \to u \to t$, slik at $c_f(p) = 1$. Dermed vil residualkapasitetene reduseres med 1, (v, u) fjernes fra residualnettverket og (u, v) blir lagt til. Den neste forøkende banen blir igjen $s \to u \to v \to t$, og slik fortsetter prosessen. Dermed vil metoden bruke 2 000 000 forøkninger for å nå maksimum flyt (merk: 1 000 000 for de øvre og 1 000 000 for de nedre). Kjøretiden blir dermed svært stor!

Hvis vi bruker BFS vil den første forøkende banen bli $s \to u \to t$, slik at $c_f(p) = 1\,000\,000$. Dermed vil residualkapasitetene reduseres med 1 000 000 og kantene (s, u) og (u, t) fjernes fra residualnettverket. Den neste forøkende banen blir $s \to v \to t$, slik at $c_f(p) = 1\,000\,000$. Dermed vil residualkapasitetene reduseres med 1 000 000 og kantene (s, v) og (v, t) fjernes fra residualnettverket. Dermed vil ikke residualnettverket ha flere forøkende baner og vi har allerede funnet maksimal flyt!

Merk: pensum er å vite at FORD-FULKERSON med BFS kalles Edmonds-Karp algoritmen

Edmonds-Karp algoritmen

Når vi bruker BFS for å finne den forøkende banen p, sier vi at vi bruker **Edmonds-Karp algoritme**. Den forøkende banen vil være den korteste banen fra s til t når hver kant har samme vekt. **Kjøretiden til Edmonds-Karp algoritme er $O(VE^2)$**. Grunnen til dette er at BFS vil bruke tiden $O(E)$ for å finne en forøkende sti og antall forøkninger er $O(VE)$. Vi ønsker å vise at antall forøkninger er $O(VE)$, noe vi gjør ved å vise to ting:

1. For hver forøkning må **shortest-path mellom kilden og enhver annen verteks øke monotont** i residualnettverket (dvs. kan ikke reduseres).
2. **Total antall forøkninger er $O(VE)$.**

Del 1 – shortest-path øker monotont

Bevis ved selvmotsigelse: vi antar at for en verteks v finnes det en forøkning som fører til at shortest-path avstanden mellom s og v blir redusert. Vi lar f være flyten før forøkningen og f' være flyten etter forøkningen. Siden shortest-path blir redusert etter forøkningen vil $\delta_{f'}(s,v) < \delta_f(s,v)$. Etter forøkningen vil shortest-path i $G_{f'}$ være $s \rightsquigarrow u \rightarrow v$, slik at $(u,v) \in E_{f'}$ og $\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1$. Pga måten vi valgte v, vil ikke shortest-path fra s til u ha blitt redusert etter forøkningen, så derfor vil $\delta_f(s,u) \leq \delta_{f'}(s,u)$. Dette fører til at $(u,v) \notin E_f$, fordi $(u,v) \in E_f$ ville ha krevd at:

$$\delta_f(s,v) \leq \delta_f(s,u) + 1 \leq \delta_{f'}(s,u) + 1 = \delta_{f'}(s,v)$$

som er en motsigelse på vår antagelse om at $\delta_f(s,v) > \delta_{f'}(s,v)$. Derfor har vi at $(u,v) \notin E_f$ og $(u,v) \in E_{f'}$, noe som betyr at forøkningen må ha økt flyten til (v,u), slik at $f(u,v)$ blir redusert og (u,v) blir en kant i $G_{f'}$. Dette betyr at shortest-path fra s til u i G_f vil ha kanten (v,u) i enden, noe som gir at: $\delta_f(s,v) = \delta_f(s,u) - 1$. Dersom vi bruker at $\delta_f(s,u) \leq \delta_{f'}(s,u)$ og deretter at $\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1$, får vi:

$$\delta_f(s,v) \leq \delta_{f'}(s,v) - 2$$

noe som er en motsigelse på vår antagelse om at $\delta_{f'}(s,v) < \delta_f(s,v)$. Derfor har vi vist at $\delta_{f'}(s,v) \geq \delta_f(s,v)$, altså shortest-path vil øke monotont ved hver forøkning.

Del 2 – antall forøkninger er $O(VE)$

Totalt antall forøkninger er $O(VE)$. *Bevis:* vi sier at kanten (u,v) i residualnettverket er **kritisk** langs en forøkende bane dersom $c_f(p) = c_f(u,v)$, altså denne kanten har minst residualkapasitet. Hver forøkende bane må ha minst en kritisk kant, så derfor vil antall forøkninger være gitt av totalt antall kritiske kanter. Vi ser først på residualnettverket G_f som inneholder den kritiske kanten (u,v), slik at $\delta_f(s,v) = \delta_f(u,v) + 1$. Forøkningen vil føre til at (u,v) fjernes fra G_f, fordi det er en kritisk kant. For at (u,v) skal bli en del av en senere forøkende bane, må først (v,u) være en del av en forøkende bane, slik at $f(u,v)$ blir redusert. Vi ser derfor på $G_{f'}$, som inneholder kanten (v,u), slik at $\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1$. Fra del 1 har vi at $\delta_f(s,v) \leq \delta_{f'}(s,v)$, som vi kan bruke for å finne:

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1 \leq \delta_f(s,v) + 1 = \delta_f(s,u) + 2$$

Merk: del 1 gir at vi ikke kan gå tilbake til samme bane i G_f der siste kant var (u,v) fordi det vil bety at shortest-path avstanden ikke øker. Derfor vil det nye residualnettverket ha en annen forøkende bane med (u,v) som siste kant

Altså, neste gang (u,v) blir en kritisk kant vil avstanden fra kilden ha økt med minst 2. De mellomliggende verteksene i

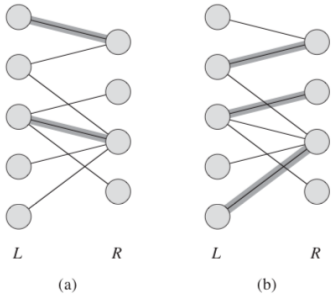
banen fra s til u kan ikke være s, u eller t, så derfor vil avstanden til verteks u være maks $|V| - 2$. Siden avstanden fra kilden økes med minst 2 hver gang kanten blir kritisk, kan (u, v) bli kritisk maks $(|V| - 2)/2$ ganger. Hver kant kan altså bli kritisk totalt $O(V)$ ganger og siden det er E kanter, vil totalt antall kritiske kanter være $O(VE)$.

Siden hver BFS krever tiden $O(E)$ for å finne en forøkende sti og det utføres $O(VE)$ antall forøkninger vil den totale kjøretiden bli $O(VE^2)$.

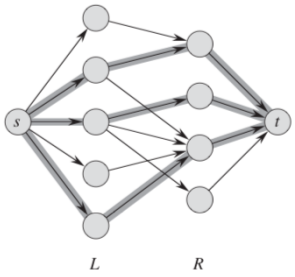
26.3 Maksimum bipartitt matching

Flere kombinatoriske problemer kan omformes til et maks-flyt problem, selv om de på overflaten ser ut til å ha lite med flytnettverk å gjøre. Et slikt eksempel er å finne maksimum matching i en bipartitt graf.

Maksimum bipartitt-matching problemet



Gitt en urette graf $G = (V, E)$ vil en **matching** være et subsett av kanter $M \subseteq E$, slik at for alle vertekser $v \in V$, vil maks én kant i M ha v som endepunkt. Dvs. ingen to kanter i M har samme endepunkt (figur a). Dersom v er endepunktet til en kant i M, sier vi at v er matched, ellers sier vi at v er unmatched. **En maksimum matching er matching med størst kardinalitet, dvs. størst antall kanter** (figur b viser maksimum matching der kardinaliteten er 3). Vi skal se hvordan vi kan finne maksimum matching til **bipartitte grafer**, som er grafer der vertekssettet kan deles inn i $\boldsymbol{V = L \cup R}$, der L og R er disjunkte og alle kantene i E går mellom L og R (ikke innad).



Finne en maksimum bipartitt matching

Vi kan bruke Ford-Fulkerson metoden for å finne maksimum matching i en urettet bipartitt graf, dersom vi lager et flytnettverk der flyten korresponderer til matchingen (se figur). Det korresponderende flytnettverket er $G' = (V', E')$. Vi lar kilden s og sluken t være nye vertekser som ikke er i V, slik at $V' = V \cup \{s, t\}$. Kantene i E' vil være lik kantene i E, i tillegg til kantene mellom s og L og kantene mellom t og R (se figur). Altså, vil: $E' = \{(s, u) : u \in L\} \cup \{(u, v) : (u, v) \in E\} \cup \{(v, t) : v \in R\}$. Dermed vil $|E'| = |E| + |V|$, siden E' er alle kantene i E pluss en kant til alle vertekser i $L \cup R = V$. Vi lar hver kant i E' ha en enhet kapasitet. Siden hver verteks i V er koblet til minst én kant vil $2|E| \geq |V|$ (to vertekser per kant). Derfor vil $|E| \leq |E'| = |E| + |V| \leq 3|E|$, slik at $|E'| = \Theta(E)$.

Lemma 5 – matching korresponderer til flyt i flytnettverk

Vi lar $G = (V, E)$ være en bipartitt graf med vertekspartisjon $V = L \cup R$ og $G' = (V', E')$ er det korresponderende flytnettverket. **Hvis $\boldsymbol{M}$ er en matching i $\boldsymbol{G}$, så vil det være en flyt i $\boldsymbol{G'}$ med verdi $|\boldsymbol{f}| = |\boldsymbol{M}|$, som er et heltall. Hvis $\boldsymbol{f}$ er en flyt i $\boldsymbol{G'}$ med heltallverdi i $\boldsymbol{G'}$, vil det være en matching gi $\boldsymbol{G}$ med kardinalitet $|\boldsymbol{M}| = |\boldsymbol{f}|$.**

Merk: Flyten må være et heltall, siden matchingen er et heltall!

Bevis: Vi må vise de to tilfellene:

- **En matching $\boldsymbol{M}$ i $\boldsymbol{G}$ korresponderer til en flyt i $\boldsymbol{G'}$:** hvis $(u, v) \in M$, så vil $f(s, u) = f(u, v) = f(v, t) = 1$, mens for alle andre kanter i E' vil flyten være 0. Det er lett å vise at f tilfredsstiller kapasitetsbegrensningen og flytbevaringen. Dermed vil f være en flyt, så neste steg er å vise at verdien til flyten er lik antall

kanter i matchingen. Hver kant $(u, v) \in M$ vil korrespondere til en enhet flyt i G' langs banen $s \to u \to v \to t$. Disse banene vil være disjunkte for ulike kanter i M (kun s og t som overlapper). Nettoflyten over snittet $(L \cup \{s\},\ R \cup \{t\})$ vil derfor være $|M|$, og dermed vil Lemma 3 gi at $|f| = |M|$.

- **En flyt i G' korresponderer til en matching M i G:** vi lar $M = \{(u, v) : u \in L, v \in R$ og $f(u, v) > 0\}$. Hver verteks $u \in L$ har kun en innkommende kant (s, u) som har kapasitet 1. Derfor vil maks én enhet flyte inn i u, og flytbevaring gir derfor at en enhet må flyte ut av u. Siden f har helltallverdi kan denne enheten med flyt entre og forlate maks én kant. En enhet med flyt vil altså entre u hvis og bare hvis det er nøyaktig en verteks $v \in R$, slik at $f(u, v) = 1$. Et symmetrisk argument gjelder for hver $v \in R$. Settet M er derfor en matching. Det står igjen å vise at $|M| = |f|$. For alle matchede vertekser vil $f(u, v) = 1$, mens for alle ikke-matchede vertekser vil $f(u, v) = 0$. Derfor vil nettoflyten over snittet være $|M|$, og Lemma 3 gir at $|M| = |f|$.

Dermed har vi vist at maksimum matching i en bipartitt graf G korresponderer til en maksimum flyt i det korresponderende flytnettverket G'. **Derfor kan vi kjøre en maks-fly algoritme på G' for å finne maksimum matching i G.** Et mulig problem er at maks-flyt algoritmen kan returnere en flyt der $f(u, v)$ ikke er et heltall, men Heltallsteoremet viser at dette problemet ikke vil oppstå.

Heltallsteoremet
Dersom kapasitetene i flytnettverket er heltall, så vil maksimum flyt f produsert av Ford-Fulkerson metoden ha en verdi $|f|$ som er et heltall. For alle vertekser u og v, vil verdien til $f(u, v)$ også være et heltall. **Heltallskapasitet = heltallsflyt!**

Altså, vil maksimum matching korrespondere til maksimum flyt, som er et heltall (merk: matchingen må være et heltall, så derfor har vi dette kravet til flyten).

Korollar 6 – kardinaliteten til maksimum matching er lik verdien til maksimum flyt
Kardinaliteten til en maksimum matching M i en bipartitt graf G er lik verdien til en maksimum flyt f i det korresponderende flytnettverket G'. *Bevis ved motsigelse:* vi antar at M er en maksimum matching i G og at den korresponderende flyten f i G' ikke er en maksimum flyt. Da vil det være en maksimum flyt f' i G', slik at $|f'| > |f|$. Siden kapasitetene i G' har heltallverdi vil Heltallsteoremet gi at $|f'|$ også er et heltall, og dermed vil f' korrespondere til en matching M' i G med kardinalitet $|M'| = |f'| > |f| = |M|$. Dette er en motsigelse på vår antagelse om at M er en maksimum matching med høyest kardinalitet. På lignende måte kan vi vise at f er en maksimum flyt i G'.

Dersom vi har gitt en bipartitt, urettet graf G, kan vi altså finne en maksimum matching ved å lage et korresponderende flytnettverk G' og deretter bruke Ford-Fulkerson metoden for å finne verdien til den maksimale flyten. **Kardinaliteten til den maksimale matchingen $|M|$ vil være lik heltallverdien til den maksimale flyten**. Kjøretiden for å finne en maksimum matching i en bipartitt graf er $O(VE') = O(VE)$, siden $|E'| = \Theta(E)$.

Forelesning 13 – NP-kompletthet

NP er den enorme klassen av ja-nei-problemer der ethvert ja-svar har et bevis som kan sjekkes i polynomisk tid. Alle problemer i NP kan i polynomisk tid reduseres til de såkalte komplette problemene i NP. Dermed kan ikke disse løses i polynomisk tid, med mindre alt i NP kan det. Ingen har klart det så langt. Læringsmålene er:

- Forstå sammenhengen mellom optimerings- og beslutningsproblemer
- Forstå koding (*encoding*) av en instans
- Forstå hvorfor løsningen vår på 0-1-ryggsekkproblemet ikke er polynomisk
- Forstå forskjellen på konkrete og abstrakte problemer
- Forstå representasjonen av beslutningsproblemer som formelle språk
- Forstå definisjonen av klassene P, NP og co-NP
- Forstå *redusibilitets-relasjonen* $\leq_P$
- **Forstå definisjonen av NP-*hardhet* og NP-*kompletthet***
- Forstå den konvensjonelle hypotesen om forholdet mellom P, NP og NPC
- **Forstå hvordan NP-kompletthet kan bevises ved én reduksjon**
- **Kjenne de NP-komplette problemene CIRCUIT-SAT, SAT, 3-CNF-SAT, CLIQUE, VERTEX-COVER, HAM-CYCLE, TSP og SUBSET-SUM**
- Forstå at 0-1-ryggsekkproblemet er NP-hardt
- Forstå at lengste enkle-vei-problemet er NP-hardt
- Være i stand til å konstruere enkle NP-kompletthetsbevis

Dette kapittelet er veldig langt og komplisert, så derfor vil jeg ha større fokus på å inkludere det som læringsmålene krever.

Kapittel 34 – NP-kompletthet

Nesten alle algoritmene vi har sett på frem til nå har vært **polynomial-tid algoritmer**, **som har worst-case kjøretid $O(n^k)$, der n er størrelsen til input.** Det finnes også problem som ikke kan løses i det hele tatt og problem som kan løses, men ikke på tiden $O(n^k)$. Problem som kan løses av polynomial-tid algoritmer kalles håndterbare eller lette, mens problem som krever superpolynomial tid (dvs. kjøretid er ikke begrenset av et polynom, eks: $\Theta(2^n)$) kalles uhåndterbare eller harde. **NP komplette problemer er problemer der statusen er ukjent.** For et NP-komplett problem har man ikke funnet en polynomial-tid algoritme, men man har heller ikke klart å bevise at ingen polynomial-tid algoritme kan eksistere for problemet. Flere av de NP-komplette problemene ligner på problemer som kan løses på polynomial tid, for eksempel:

- **Korteste vs. lengste simple bane** – vi kan finne shortest-path på tiden $O(VE)$, mens det er vanskeligere å finne longest-path. Det å bestemme om en graf inneholder en simpel bane med minst et gitt antall kanter, er NP-komplett.
- **Euler tur vs. Hamilton syklus** – en Euler tur er en syklus som traverserer hver kant kun én gang når grafen er sterkt koblet og rettet. Vi kan bestemme om grafen har en Euler tur og finne kantene som er i turen på tiden $O(E)$. En Hamilton syklus er en simpel syklus som inneholder alle verteksene i grafen. Å bestemme om grafen har en Hamilton syklus er NP-komplett.

- **2-CNF SAT vs 3-CNF SAT** – en boolean formel er tilfredsstillende (*SAT = satisfiable*) hvis den kan gis variabelverdier (0/1) slik at den blir 1. CNF er en konjunksjon (AND) av disjunkte klausuler (OR). En boolean formel er k-CNF dersom den er en CNF av nøyaktig k variabler, dvs. det er k variabler i hver OR-klausulene. For eksempel vil en 2-CNF være $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ fordi det er to variabler i hver parentes og den har en tilfredsstillende fordeling $x_1 = 1$, $x_2 = 0$ og $x_3 = 1$. Vi kan bruke polynomial tid for å bestemme om en 2-CNF er SAT, men å bestemme om en 3-CNF er SAT er NP-komplett.

Forstå definisjonen av klassene P, NP

NP-kompletthet og klassene P og NP

Vi skal se på tre klasser av problem:

- **P - består av problemer som kan løses på polynomial tid $O(n^k)$**, der n er inputstørrelsen og k er en konstant.
- **NP – består av problemer som kan verifiseres på polynomial tid.** Dersom vi får et sertifikat på problemet, kan vi verifisere at sertifikatet er korrekt på polynomial tid basert på inputstørrelsen. For eksempel for Hamilton-syklus problemet vil sertifikatet være en sekvens av $|V|$ antall vertekser og vi kan sjekke om dette er en Hamilton syklus på polynomial tid. For 3-CNF SAT problemet vil sertifikatet være en tildeling av variabelverdier, og vi kan sjekke om dette tilfredsstiller boolean formelen på polynomial tid. **Alle problemer i P vil også være i NP ($P \subseteq NP$)**, siden problemer i P kan løses på polynomial tid.
- **NPC (NP-komplett) – består av problemer som er i NP og er like hardt som ethvert problem i NP (mer senere). Hvis ethvert NP-komplett problem kan løses på polynomial tid, så vil alle problemer i NP ha en polynomial-tid algoritme.**

Dersom du kan vise at et problem er NP-komplett, vil du gi gode bevis for at det er uhåndterbart. I dette tilfellet er det bedre å finne en tilnærmende algoritme enn å forsøke å utvikle en rask algoritme som løser problemet eksakt.

Være i stand til å konstruere enkle NP-kompletthetsbevis

Overblikk for hvordan vise at problemer er NP-komplette

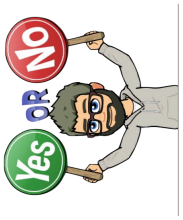
Når vi demonstrerer at et problem er NP-komplett ,vil vi påstå hvor hardt det er i stedet for hvor lett det er. Vi vil vise at det er lite sannsynlig at en effektiv algoritme eksisterer. Tre hovedkonsept for å vise at et problem er NP-komplett:

- **Forholdet mellom optimaliseringsproblem og beslutningsproblem**
- **Reduksjoner**
- **Et første NP-komplett problem**

Forstå sammenhengen mellom optimerings- og beslutningsproblemer

Beslutningsproblem vs. optimaliseringsproblem

Mange problemer er **optimaliseringsproblem** der hver mulig løsning har en assosiert verdi, og vi ønsker å finne løsningen som har best verdi. For eksempel vil løsningen til SHORTEST-PATH problemet være banen med minst vekt. **NP-kompletthet kan ikke brukes direkte på optimaliseringsproblem, men heller på beslutningsproblem der svaret er "ja" eller "nei" (se figur).** Selv om NP-komplette problem er begrenset til beslutningsproblem, kan vi bruke et nyttig forhold mellom optimaliseringsproblem og beslutningsproblem. **Vi kan omforme et optimaliseringsproblem til et beslutningsproblem ved å sette en grense på verdien som skal optimaliseres**. For eksempel i SHORTEST-PATH problemet kan vi lage et beslutningsproblem kalt PATH der vi ser om det eksisterer en bane med maks k

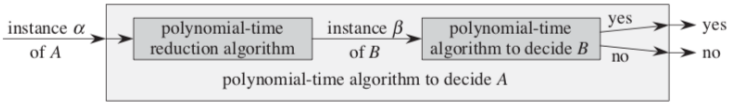


kanter. Dersom et optimaliseringsproblem er lett, vil det relaterte beslutningsproblemet også være lett. **Hvis vi finner bevis på at beslutningsproblemet er hardt, vil vi derfor også ha funnet bevis på at optimaliseringsproblemet er hardt.** Kompleksitetsklassen P er beslutningsproblemer som kan løses på polynomisk tid.

Reduksjoner

Dersom begge problemene er beslutningsproblemer, kan vi også bruke konseptet av å vise at et problem ikke er hardere eller lettere enn et annet. Vi ser på et beslutningsproblem A som vi ønsker å løse på polynomial tid. Input til et bestemt problem kalles en **instans** av problemet, for eksempel i PATH vil instansen være en graf, start- og slutt vertekser og heltallet k. Vi vet allerede hvordan vi kan løse et annet beslutningsproblem B på polynomial tid, og vi har en prosedyre for å omdanne enhver instans α av A til en instans β av B med følgende egenskaper:

- **Transformasjonen tar polynomial tid**
- **Svarene er de samme**, altså svaret for α er "Ja" kun hvis svaret for β også er "ja"



En slik prosedyre kalles en **polynomial-tid reduksjonsalgoritme**, og den gir oss en måte å løse problem A på polynomial tid:

1. **Gitt en instans α av problem A, bruk en polynomial-tid reduksjonsalgoritme for å transformere den til en instans β av problem B**
2. **Kjør polynomial-tid beslutningsalgoritmen for B på instansen β**
3. **Bruk svaret på β som svaret på α**

Så lenge hvert steg tar polynomial tid, så vil alle tre også ta polynomial tid og dermed vil vi kunne bestemme α på polynomial tid. Ved å redusere det å løse problem A til å løse problem B, kan vi bruke "lettheten" til B for å vise "lettheten" til A.

For å vise at et problem er NP-komplett gjør vi det motsatte, altså viser at problem B er NP-komplett ved antagelsen om at problem A også er NP-komplett. Anta at vi har et beslutningsproblem A som vi vet ikke har en polynomial-tid algoritme og en polynomial-tid reduksjon som omdanner instanser av A til instanser av B. Da kan vi bruke et enkelt selvmotsigelse bevis for å vise at ingen polynomial-tid algoritme kan eksistere for B. Dersom vi antar at B har en polynomial-tid algoritme, da vil metoden på figuren over gi at vi kan løse problem A på polynomial tid. Dette er en motsigelse på vår antagelse om at det finnes ingen polynomial-tid algoritme for A. Merk: vi kan ikke anta at det er absolutt ingen polynomial-tid algoritmer for problem A, men vi kan anta at A er NP-komplett og dermed vise at B også er NP-komplett.

Et første NP-komplett problem

Reduksjonsteknikken avhenger av at vi har et problem som allerede er kjent for å være NP-komplett. **For å kunne vise at et problem er NP-komplett, trenger vi derfor et "første" NP-komplett problem.** Vi skal se på dette problemet i 34.3.

34.1 Polynomial tid

Polynomial-tid problemer ses på som håndterbare, av tre grunner:

- Et problem som krever tiden $\Theta(n^{100})$ kan ses på som uhåndterbart, men veldig få problemer vil i praksis kreve tid med så høy grad. Når den første polynomial-tid

algoritmen for løsningen har blitt oppdaget, vil som regel mer effektive algoritmer finnes etter hvert
- Et problem som kan løses på polynomial tid i en beregningsmodell kan løses på polynomial tid i en annen beregningsmodell
- Klassen av polynomial-tid problemer har fine avslutningsegenskaper, siden polynomer lukkes ved addisjon, multiplikasjon og komposisjon. For eksempel hvis output til en polynomial-tid algoritme sendes som input til en annen, vil komposisjonsalgoritmen være polynomial

Forstå forskjellen på konkrete og abstrakte problemer

Abstrakte problem

Et abstrakt problem Q er en binær relasjon mellom et sett I av probleminstanser og et sett S av problemløsninger. For eksempel for SHORTEST-PATH problemet vil en instans være en trippel som består av en graf og to vertekser, mens en løsning er en sekvens av vertekser. Selve problemet SHORTEST-PATH er relasjonen som kobler hver instans av en graf og to vertekser til en shortest-path grafen mellom de to verteksene i grafen. Den korteste banen er ikke nødvendigvis unik, så derfor kan et gitt probleminstans ha flere problemløsninger.

I tilfelle med beslutningsproblemer, kan et abstrakt problem ses som en funksjon som kartlegger instanssettet I til løsningssettet $\{0, 1\}$. For eksempel for PATH problemet, dersom $i = \langle G, u, v, k \rangle$ er en instans, så vil $PATH(i) = 1$ hvis det er en shortest-path fra u til v som har maks k kanter og $PATH(i) = 0$ ellers

Forstå koding (*encoding*) av en instans

Koding (*encoding*)

For at et dataprogram skal kunne løse et abstrakt problem, må vi representere probleminstansene på en måte som programmet forstår. **En koding av et sett I av abstrakte objekter er en kartlegging e fra I til settet av binære strenger.** For eksempel kan de naturlige tallene $\mathbb{N} = \{0, 1, 2, 3 \dots\}$ kodes som strengene $\{0, 1, 10, 11, \dots\}$ og vi kan bruke denne kodingen for å finne $e(17) = 10001$. En dataalgoritme som "løser" et abstrakt beslutningsproblem vil derfor ta en kodet versjon av probleminstansen som input. **Et konkret problem er et problem der instanssettet I er et sett av binære strenger**. Husk: instanssettet er settet av inputverdier, og ved et konkret problem vil disse være kodet til binære strenger. Merk at settet for problemløsninger allerede er binær for beslutningsproblem: $S = \{0, 1\}$, så det er kun instanssettet I som må kodes.

Vi kan nå formelt definere kompleksitetsklassen P som settet av konkrete beslutningsproblemer som kan løses på polynomial tid. Med konkret menes det at instanssettet er et sett av binære strenger, mens med beslutningsproblem menes det at løsningen er "ja" eller "nei. Koding brukes for å kartlegge abstrakte problemer til konkrete. Dersom vi har et abstrakt beslutningsproblem Q som kartlegger et instanssett I til $\{0, 1\}$, kan koding brukes for å finne et relatert konkret beslutningsproblem $e(Q)$ der $e: I \to \{0, 1\}^*$. Hvis det abstrakte problemet har løsning $Q(i) \in \{0, 1\}$ for instansen $i \in I$, vil løsningen på det konkrete problemet $e(i) \in \{0, 1\}^*$ også være $Q(i)$. **Det konkrete problemet vil altså gi samme løsninger som det abstrakte problemet for kodet versjon av instansen**.

Kjøretiden til en algoritme vil avhenge av hvordan instanssettet (inputen) kodes, og forskjellen kan være fra polynomial tid (eks: $O(n)$) til superpolynomial tid (eks: $O(2^n)$)! Dersom vi ser bort fra "dyre" kodinger, vil likevel kodingen ha lite effekt på om

problemet kan løses på polynomial tid i praksis. Dersom to kodinger e_1 og e_2 er polynomial relatert, vil det ikke spille noen rolle hvilken koding vi bruker når vi skal bestemme om et problem kan løses på polynomial tid. Med polynomial relatert menes det at vi på polynomial tid kan finne kodingen $e_2(i)$ fra kodingen $e_1(i)$, og omvendt. **Dersom vi har et abstrakt beslutningsproblem vil $e_1(Q) \in P$ kun hvis $e_2(Q) \in P$, altså vil Q ha polynomial-tid løsning kun hvis det blir gitt av begge kodingene når de er polynomial relatert**. Dersom $e_2(Q)$ kan løses på polynomial tid, og vi kan finne bruke $e_2(Q)$ for å finne kodingen $e_1(Q)$ på polynomial tid, følger det at $e_1(Q)$ også kan løses på polynomial tid.

Vi antar at kodingen på et heltall er polynomial relatert til den binære representasjonen og at kodingen på endelige sett er polynomial relatert til kodingen som en liste av dens elementer. Dette er en standard koding som overføres til andre matematiske objekter og betegnes med $\langle \dots \rangle$, som for eksempel vil $\langle G \rangle$ være standard koding av en graf G. Så lenge vi bruker en koding som er polynomial relatert til denne standard kodingen, kan vi anta at kodingen ikke vil påvirke om algoritmen kan løses på polynomial tid.

Forstå representasjonen av beslutnings-problemer som formelle språk

Et formelt-språk rammeverk

Et språk L er et sett av strenger laget av symboler fra et alfabet $\sum$.
For eksempel dersom $\sum = \{0, 1\}$ vil $L = \{10, 11, 101, 111, \dots\}$ være språket av binær representasjon av primtall. Språket som inneholder mulige ord bygd opp av symbolene fra $\sum$ kalles $\sum^*$. For eksempel vil $\{0, 1\}^*$ være settet av alle binære sekvenser av enhver lengde $\sum^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$. Et språk over et alfabet $\sum$, vil være et subset av $\sum^*$, altså $L \subseteq \sum^*$. En rekke operasjoner kan utføres på språk, for eksempel **union**, **snitt**, **komplement** ($\overline{L} = \sum^* - L$) og **konkatenering** ($L_1 L_2$ slik at $L = \{x_1 x_2 : x_1 \in L_1 \text{ og } x_2 \in L_2\}$). Lukningen av et språk L er språket $L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \dots$, der L^k er språket vi finner ved å regne konkateneringen av L med seg selv k ganger.

For ethvert konkret beslutningsproblem Q vil instanssettet være settet av binære strenger, altså $\sum^*$, der $\sum = \{0, 1\}$. Mer spesifikt kan vi definere **språket L som et sett av strengene som gir 1 (= ja) som svar, altså: $L = \{x \in \sum^* : Q(x) = 1\}$**. Dvs. språket L inneholder kun de binære strengene x som kan brukes som input til problemet Q for å få 1 som output. Språket inneholder Ja-instanser, men ikke nei-instanser! Eks: PATH vil ha språket: $PATH = \{\langle G, u, v, k \rangle : G = (V, E)$ er en urettet graf, $u, v \in V, k \geq 0$ er et heltall, og det eksisterer en bane fra u til v i G som har maks k kanter$\}$.

En algoritme A aksepterer en streng x, dersom $A(x) = 1$ (dvs. input x gir 1 som output). Algoritmen avviser x dersom $A(x) = 0$. Vi skiller mellom akseptere og avgjøre:

- **Et språk L aksepteres av algoritmen A dersom det består av et sett av strenger $L = \{x : A(x) = 1\}$.** Dvs. språket er et sett av strenger som A aksepterer, $x \in L$ gir $A(x) = 1$.
- **Et språk L avgjøres av algoritmen A dersom $x \in L$ gir $A(x) = 1$ og $x \notin L$ gir $A(x) = 0$**. Dvs. alle binære strenger i L er akseptert OG alle binære strenger ikke i L er avvist. **Selv om L er akseptert av A, kan det derfor hende at L ikke avgjøres av A,** siden algoritmen kan la være å svare for nei-instanser (dvs. ikke gi $A(x) = 0$ når $x \notin L$, for eksempel som følge av at den entrer evig loop).

Dvs. avgjørelse krever også avvisning av $x \notin L$ i tillegg til aksептering av $x \in L$!

Vi sier at A aksepterer en binær streng x, dersom $A(x) = 1$. Siden et språk består av mange binære strenger, vil vi si at A aksepterer et språk dersom $A(x) = 1$ for alle disse strengene x.

Et språk L aksepteres på polynomial tid av algoritmen A dersom den blir gitt input $x \in L$ og aksepterer x på tiden $O(n^k)$. Et språk L avgjøres på polynomial tid av algoritmen A dersom den blir gitt input x og vil korrekt avgjøre om $x \in L$ på tiden $O(n^k)$. For å akseptere et språk må algoritmen kun gi et svar når den blir gitt en streng i L, mens for å avgjøre et språk må algoritmen riktig akseptere eller avvise hver streng i $\{0, 1\}^*$.

For eksempel har vi en algoritme som vi akseptere PATH på polynomial tid, men ikke avgjøre PATH. Denne algoritmen vil ta $\langle G, u, v, k\rangle$ som input, vil undersøke om G er en urettet graf, om u og v er vertekser i G og vil deretter bruke BFS for å finne shortest-path fra u til v og sammenligner dette med k. Dersom disse stemmer vil algoritmen returnere 1, siden det betyr at $\langle G, u, v, k\rangle \in PATH$. Denne algoritmen vil ikke avgjøre PATH, fordi den gir ikke output 0 for instanser der shortest-path har mer enn k kanter. En avgjørende algoritme for PATH må avvise binære strenger som ikke hører til PATH språket, noe som er lett å laget (få den til å returnere 0 når det er ingen bane med maks k kanter).

En **kompleksitetsklasse** er et sett av problemer som bruker lignende spekter av rom og tid for å løses (eks: P, NP, NPC). **Vi kan definere kompleksitetsklassen som et sett av språk, der hvilke språk som inkluderes bestemmes av et kompleksitetsmål til en algoritme som bestemmer om en gitt streng x hører til språket L.** For å bestemme om L kan tilhøre en bestemt kompleksitetsklasse, vil vi altså se på for eksempel kjøretiden til algoritmen som bestemmer om $x \in L$. Dermed får vi en alternativ definisjon av kompleksitetsklassen P:

$$P = \{L \subseteq \{0, 1\}^* : \text{de eksisterer en algoritme } A \text{ som avgjør } L \text{ på polynomial tid}\}$$

dvs. et språk L vil være i kompleksitetsklassen P dersom det finnes en algoritme som bruker polynomial tid for å avgjøre om en binær streng x er i språket L eller ikke (algoritmen bruker polynomisk tid for å avgjøre et språk). P vil også være språkene som aksepteres på polynomial tid:

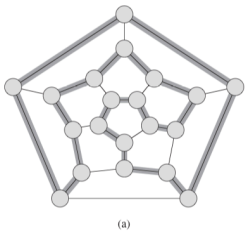
$$P = \{L : \text{de eksisterer en algoritme } A \text{ som aksepterer } L \text{ på polynomial tid}\}$$

Merk: klassen av språk som avgjøres er et subsett av klassen av språk som aksepteres av algoritmen. **Kompleksitetsklassen P vil altså inneholde språkene som kan avgjøres og aksepteres på polynomisk tid. Cobhams tese gir at det er disse problemene vi kan løse i praksis.**

34.2 Polynomial-tid verifisering

Vi skal nå se hvordan vi kan **verifisere medlemskap i språk vha sertifikater**. For eksempel for PATH problemet med instans $\langle G, u, v, k\rangle$, vil et sertifikat være banen p mellom u og v dersom vi kan vise at p er en bane i G og har lengde mindre enn k. Dette sertifikatet bekrefter at instansen er en del av PATH språket. PATH hører til P, så derfor vil det ta like lang tid å verifisere medlemskapet fra et gitt sertifikat som det vil ta å løse problemet fra bunnen av (undersøke $\langle G, u, v, k\rangle \in PATH$ vha A som avgjør PATH). For problemer som enda ikke har polynomial tid avgjørelse, vil verifikasjon være lettere.

Kjenne de NP-komplette problemene: HAM-CYCLE



Hamilton syklus (HAM-Cycle)

En Hamilton syklus i en urettet graf er en simpel syklus som inneholder alle verteksene i V (figur a). Figur b viser et eksempel på en graf som ikke inneholder en Hamilton syklus. **Hamiltonian-syklus problemet** går ut på å finne ut om en graf G inneholder en Hamilton syklus, og kan defineres på formelt språk som:

$$HAM\text{-}CYCLE = \{\langle G\rangle : G \text{ er en Hamilton syklus}\}$$

HAM-CYCLE språket vil altså kun inneholde probleminstanser $\langle G\rangle$ (standard kodet G) som en Hamilton syklus. En mulig algoritme for å avgjøre HAM-CYCLE språket (dvs. bestemme om gitt $\langle G\rangle$ er Hamilton syklus eller ikke) er å liste opp alle permutasjonene av verteksene i G og deretter sjekke om en av de er en Hamilton syklus. Dette vil ta tiden $\Omega\left(2^{\sqrt{n}}\right)$, som er superpolynomial. **Hamiltonian-syklus problemet er NP-komplett og kan lett verifiseres på polynomial tid (NP problem).**

Verifikasjonsalgoritme

Dersom vi har gitt en syklus som skal være en Hamilton syklus, kan vi lett verifisere dette ved å lage en algoritme som sjekker om syklusen inneholder alle verteksene V og om de etterfølgende kantene i syklusen er i G. Denne verifikasjonsalgoritmen kan kjøre på tiden $O(n^2)$, der n er lengden til kodingen av G. **Et bevis på at en gitt Hamiltonian syklus eksisterer i grafen, kan altså verifiseres på polynomial tid.**

Merk: sertifikat er en påstått løsning for en instans og må verifiseres

En **verifikasjonsalgoritme** A vil ha to argumenter: en vanlig input streng x og en binær streng y som kalles et **sertifikat**. Algoritmen vil **verifisere** x dersom det eksisterer en y, slik at $A(x, y) = 1$. På formelt språk:

$$L = \{x = \{0,1\}^* : \text{det eksisterer } y \in \{0,1\}^* \text{ slik at } A(x,y) = 1\}$$

En algoritme A vil altså verifisere språket L dersom det for enhver string $x \in L$ eksisterer et sertifikat y som A kan bruke for å vise at $x \in L$. For $x \notin L$ kan det ikke være noe sertifikat. I HAM-CYCLE problemet vil sertifikatet være den gitte Hamilton syklusen. Dersom grafen er Hamiltonian vil det gitte sertifikatet gi nok informasjon for å verifisere dette. Dersom grafen ikke er Hamiltonian vil vi ikke ha noe sertifikat.

Kompleksitetsklassen NP

Kompleksitetsklassen NP er klassen av språk som kan verifiseres av en polynomial-tid algoritme. Et språk vil høre til NP kun hvis det eksisterer en polynomial-tid algoritme, slik at:

$$L = \{x \in \{0,1\}^* : \text{det eksisterer et sertifikat } y \text{ med } |y| = O(|x|^c) \text{ slik at } A(x,y) = 1\}$$

Algoritme A verifiserer språket L på polynomial tid. Dersom $L \in P$, vil også $L \in NP$, altså et problem som kan løses kan også verifiseres. Vi sier at $P \subseteq NP$, men det er ukjent om $P = NP$. P består av problemer som kan løses lett, mens NP består av problemer som kan verifiseres lett. Det er ofte mer vanskelig å løse et problem fra bunnen av enn å verifisere en løsning, så derfor **tror man at NP inkluderer språk som ikke er i P** (dvs. $L \in NP$ gir ikke $L \in P$: et problem som kan verifiseres, kan ikke nødvendigvis løses). Dette betegnes som $\boldsymbol{P \neq NP}$.

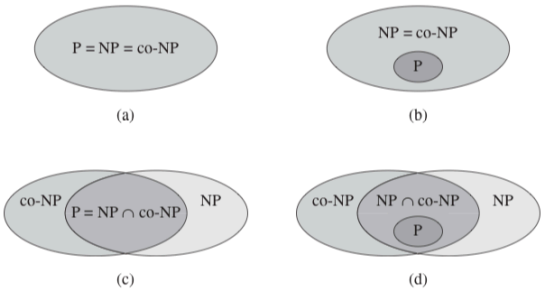
Forstå den konvensjonelle hypotesen om forholdet mellom P og NP

Det er ukjent om NP er lukket under komplementet, altså om komplementet til et problem i klassen fortsatt er i klassen, noe som betegnes med: $L \in NP$ gir at $\overline{L} \in NP$. **Vi definerer kompleksitetsklassen co-NP (complement NP) som settet av språk L, slik at $\overline{L} \in NP$**. co-NP vil altså inneholde komplementer av problemer funnet i NP. For eksempel siden NP inneholder HAM-CYCLE som sier at grafen er en syklus, vil co-NP inneholde et språk som sier at grafen ikke er en Hamilton-syklus. NP ønsker å verifisere om vi kan svare "ja", mens co-NP ønsker å verifisere om vi kan svare "nei". **For NP problemer er det lett å verifisere at det er en løsning, mens for co-NP problemer er det lett å verifisere at det ikke er en løsning.**

Forstå definisjonen av co-NP

Det ukjente spørsmålet om $L \in NP$ gir at $\overline{L} \in NP$, vil derfor bli om $\boldsymbol{NP = co\text{-}NP}$ (et problem som kan verifiseres, vil ha komplement som også kan verifiseres)**.** Vi har at $P \subseteq NP$ og $P \subseteq co\text{-}NP$ og vet at P er lukket under komplementet (problem kan løses betyr at komplement også kan løses). Det følger at $P \subseteq NP \cap co\text{-}NP$, altså om et problem kan løses så kan problemet og komplementet verifiseres. Figuren viser fire ulike forhold, og man er usikker hvilket forhold som er riktig:

a) $\boldsymbol{P = NP = co\text{-}NP}$ – hvis et problem eller komplementet kan verifiseres kan det også løses, noe som er usannsynlig
b) $\boldsymbol{NP = co\text{-}NP}$ – hvis et problem kan verifiseres så kan også komplementet verifiseres
c) $\boldsymbol{P = NP \cap co\text{-}NP}$ – hvis problemet og komplementet kan verifiseres (dvs. verifisere at vi kan svare "ja" på problemet og "nei" på komplementet) kan problemet løses.
d) $\boldsymbol{P \neq NP \cap co\text{-}NP}$ **og** $\boldsymbol{NP \neq co\text{-}NP}$ – verifikasjon av problemet og komplementet betyr ikke at problemet kan løses. Dersom problemet kan verifiseres betyr det ikke at komplementet kan verifiseres. Dette er mest sannsynlig.



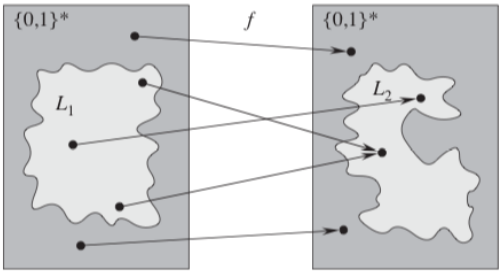
34.3 NP-kompletthet og redusibilitet

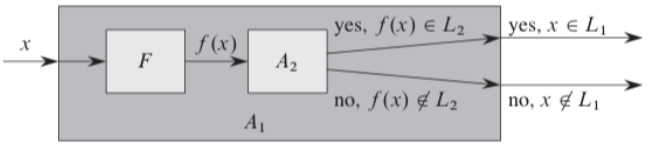
$P \neq NP$ betyr at selv om det finnes en effektiv måte å sjekke om en gitt løsning er korrekt, betyr det ikke at det finnes en effektiv måte å finne løsningen på. NP-komplette problem styrker dette utsagnet. **Denne klassen har egenskapen av at dersom ett NP-komplett problem kan løses på polynomial tid, så kan alle problem i NP ha en polynomial-tid løsning, altså $\boldsymbol{P = NP}$**. Språket HAM-CYCLE vil være ett NP-komplett problem. Hvis vi finner en algoritme som avgjør HAM-CYCLE på polynomial tid (dvs. avgjør om en graf er en Hamilton syklus eller ikke på tiden $O(n^k)$), vil vi kunne løse alle problem i NP på polynomial tid. **De NP-komplette språkene er de hardeste språkene i NP.**

Forstå *redusibilitets-relasjonen* $\leq_P$

Redusibilitet

Et språk $\boldsymbol{L_1}$ er polynomial-tid redusibel til et språk $\boldsymbol{L_2}$ dersom $\boldsymbol{x \in L_1}$ kun hvis $\boldsymbol{f(x) \in L_2}$, der $\boldsymbol{f}$ er en polynomial-tid beregnende reduksjonsfunksjon. Denne funksjonen blir funnet av en polynomial-tid algoritme F, kalt **reduksjonsalgoritmen**. Vi bruker betegnelsen $\boldsymbol{L_1 \leq_P L_2}$ for å gi at L_1 kan reduseres til L_2 på polynomial-tid. Figuren illustrerer dette konseptet. L_1 og L_2 er begge subsett av $\{0,1\}^*$, og **$\boldsymbol{f}$ gir en kartlegging slik at hvis $\boldsymbol{x \in L_1}$, så vil $\boldsymbol{f(x) \in L_2}$.** Hvis $x \notin L_1$ vil $f(x) \notin L_2$. Reduksjonsfunksjonen vil kartlegge en instans av språket L_1 til en instans av språket L_2. Dersom vi kan vise at $f(x)$ instansen er i L_2 språket vil vi derfor ha vist at x instansen er i L_1 språket.





Lemma 1

Hvis $L_1, L_2 \subseteq \{0,1\}^*$ er språk slik at $L_1 \leq_P L_2$, så vil $L_2 \in P$ gi at $L_1 \in P$. Dvs. dersom vi kan redusere L_1 til L_2 på polynomial tid og vi finner en polynomial-tid løsning på L_2, vil vi også ha en polynomial løsning på L_1. Dersom A_1 er en polynomial-tid algoritme som løser L_1 kan vi se på figuren hvordan denne fungerer. For en gitt input x vil A_1 bruke polynomial-tid reduksjonsalgoritmen F for å transformere x til $f(x)$. Deretter kan A_2 brukes for å teste om $f(x) \in L_2$ på polynomial tid. Output til A_1 vil bestemmes av output til A_2. A_1 har polynomial kjøretid siden både F og A_2 kjører på polynomial tid.

NP-kompletthet

Hvis $L_1 \leq_P L_2$, så kan ikke L_1 være mer enn én polynomial faktor hardere enn L_2, fordi ellers kan ikke L_1 reduseres til L_2 på polynomial tid. Vi kan nå definere settet av NP-komplette språk, som er de hardeste problemene i NP.

Et språk $L \subseteq \{0,1\}^*$ er NP-komplett (NPC) hvis:
1. $L \in NP$
2. $L' \leq_P L$ for alle $L' \in NP$

Forstå definisjonen av NP-*hardhet* og NP-*kompletthet*

Altså, et NP-komplett problem vil være et NP problem og det vil eksistere et annet NP problem som kan på polynomial-tid reduseres til dette NP problemet. **Et språk som tilfredsstiller egenskap 2, men ikke nødvendigvis egenskap 1, sies å være NP-hard**. Altså vi har to definisjoner:

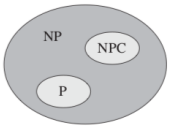
- **NP-hardhet** – L vil være NP-hard hvis det eksisterer et NP problem som kan reduseres til L på polynomial tid. L er ikke nødvendigvis et NP problem selv, altså er det ikke garantert at NP-harde problem kan verifiseres på polynomial tid. Dette er problemer som er minst like harde som de hardeste problemene i NP, så hvis disse kan løses på polynomial-tid kan alle NP problem løses.
- **NP-kompletthet** – betyr at problemet er både NP og NP-hard, altså kan en løsning verifiseres lett, men det er minst så hardt som det hardeste problemet i NP. Hvis vi kan løse disse, kan vi løse alle NP problem og løsningen kan verifiseres på polynomial tid.

For å vise NP-kompletthet må vi altså vise at det er både NP og NP-hardt. For å vise at et problem er NP-hardt må vi vise at alle andre NP-problem kan reduseres til dette NP-komplette problemet på polynomial tid.

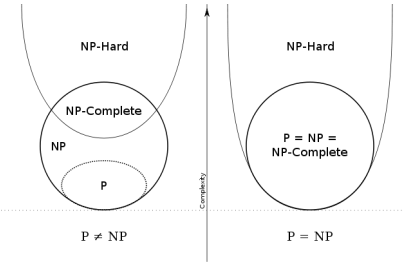
Forstå den konvensjonelle hypotesen om forholdet mellom P og NP

Teorem

Hvis ett NP-komplett problem kan løses på polynomial tid, så vil $P = NP$. Hvis ingen NP-komplette problem kan løses på polynomial tid, så vil $P \neq NP$. *Bevis:* Dersom $L \in P$ og $L \in NPC$, så vil vi for $L' \in NP$ ha at $L' \leq_P L$ av definisjonen til NPC. Dermed vil Lemma 1 gi at $L' \in P$. Det motsatte gjelder for $L \notin P$.



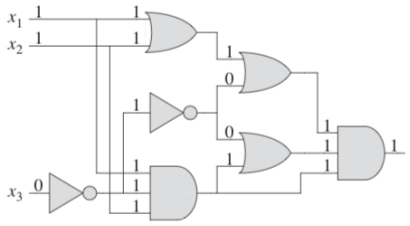
De fleste tror at $P \neq NP$ som fører til forholdet på figurene.



CIRCUIT-SAT – krets-tilfredsstilhet problemet

Når vi har bevist at ett problem er NP-komplett kan vi bruke polynomial-tid redusibilitet for å vise at andre problem er NP-komplett. Vi lar dette problemet være CIRCUIT-SAT problemet. En boolean krets er tilfredsstillende (*SAT = satisfiable*) hvis den kan gis variabelverdier (0/1) slik at output til kretsen blir 1 (se figur). **Krets-tilfredsstilhet**

Kjenne de NP-komplette problemene: CIRCUIT-SAT



problemet er: "gitt en boolean krets med logiske porter og en output, er den tilfredsstillende?". I formelt språk blir dette:

$$CIRCUIT\text{-}SAT = \{\langle C \rangle : C \text{ er en tilfredsstillende boolean krets}\}$$

CIRCUIT-SAT språket vil altså inneholde standard kodinger av kretser som er tilfredsstillende, dvs. har 1 som output. En mulig algoritme for å avgjøre CIRCUIT-SAT (dvs. bestemme om gitt krets gir output 1) er å undersøke alle mulige tildelinger av variabelverdier. Dersom det er k input vil det være 2^k mulige tildelinger og derfor vil dette ta tiden $\Omega(2^k)$, som er superpolynomial. **CIRCUIT-SAT problemet er NP-komplett**, som vi nå skal bevise ved å se at det er NP og NP-hardt:

- **CIRCUIT-SAT problemet hører til NP klassen** – kan bevises ved å finne en polynomial-tid verifiseringsalgoritme A som kan verifisere CIRCUIT-SAT. Denne algoritmen vil ha to input: (1) en boolean krets C og (2) et sertifikat som gir fordelingen av variabelverdier til kretsen C. Dersom denne fordelingen gir at C får output 1, vil algoritmen få output 1, ellers vil den få output 0. Dette kan gjøres på polynomial tid, så derfor vil CIRCUIT-SAT $\in NP$
- **CIRCUIT-SAT problemet er NP-hardt** – vi må vise at alle språk i NP kan reduseres til CIRCUIT-SAT på polynomial tid. Dersom L er ethvert språk i NP skal vi finne en reduksjonsfunksjon f som vil kartlegge alle binære strenger x til en krets $C = f(x)$, slik at $x \in L$ kun hvis $C \in CIRCUIT\text{-}SAT$. Merk at dette er definisjonen av $L \leq_P CIRCUIT\text{-}SAT$. Vi må vise at det eksisterer en reduksjonsalgoritme som finner riktig f, slik at C er tilfredsstillene kun når det eksisterer et sertifikat y som gir at $A(x, y) = 1$ og vi må vise at dette gjøres på polynomial tid.

34.4 NP-kompletthet bevis

Vi har bevist at CIRCUIT-SAT problemet er NP-komplett ved å direkte bevise at $L \leq_P CIRCUIT\text{-}SAT$ for alle språk $L \in NP$. Nå skal vi se hvordan vi kan bevise NP-kompletthet uten å direkte redusere alle språk i NP til det gitte språket.

Forstå hvordan NP-kompletthet kan bevises ved én reduksjon

Lemma 2

Hvis L er et språk slik at $L' \leq_P L$ for en $L' \in NPC$, så vil L være NP-hardt. Hvis i tillegg $L \in NP$, så vil $L \in NPC$. Vi har et kjent NP-problem L' som betyr at alle NP-problem kan reduseres til L'. Dersom L' kan reduseres til L, vil transitivitet gi at alle NP-problem kan reduseres til L. Hvis vi så kan vise at L er et NP-problem selv, har vi vist at L er et NP-komplett problem. Dette er altså metoden når vi kjenner et annet NP-komplett problem som lar oss bevise NP-kompletthet med én reduksjon. **Hvis ikke er vi nødt til å vise at $L' \leq_P L$ for alle $L' \in NP$, som krever mange reduksjoner.**

Metoden for å vise at L er NP-komplett er:

Være i stand til å konstruere enkle NP-kompletthetsbevis

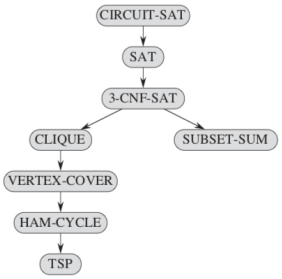
1. **Bevis $L \in NP$**
2. **Velg et kjent NP-problem L'**
3. **Beskriv en algoritme som regner ut funksjonen f som kartlegger alle instanser x i L' til instanser $f(x)$ i L. Dvs. finn reduksjonsalgoritmen**
4. **Vis at funksjonen f tilfredsstiller $x \in L'$ kun hvis $f(x) \in L$ for alle x**
5. **Vis at reduksjonsalgoritmen bruker polynomial tid for å finne f**

Steg 2-5 viser at L er NP-hardt med én reduksjon.

OBS: grunnen til at vi kan vise NP-kompletthet med én reduksjon er at vi kan ta utgangspunkt i et kjent NP-komplett problem og vise at dette kan reduseres til problemet vi ser på. Transitivitet gir at alle NP problem kan da reduseres til problemet vi ser på, og dersom det i tillegg er et NP problem, vil det da være et NPC problem. Dermed slipper vi å redusere alle språk i NP for å bevise NP kompletthet.

34.4 -34.5 NP-komplette problem

Vi skal nå gi NP-kompletthet bevis til ulike problemer. På figuren kan vi se hvilket NP-komplett språk som brukes i beviset på et annet NP-komplett språk (språket som reduseres peker ned mot NP-komplette problem som skal bevises). Som vi kan se er CIRCUIT-SAT ved roten, fordi denne løste vi ved å redusere alle språk i NP, og derfor brukte vi ingen andre NP-komplette problem for å vise denne. Siden det er pensum å *kjenne* til disse problemene, har jeg valgt å utelukke selve utregningen i bevisene.



SAT

SAT (tilfredsstillhet) problemet går ut på å bestemme om en boolean, logisk formel kan være tilfredsstillende, dvs. kan gi output 1 for bestemte variabelverdier. I SAT problemet vil instansen være en logisk formel, og spørsmålet er om formelen kan være sann (om bestemte variabelverdier vil gi output 1) :

Merk: SAT tilsvarer CYCLIC-SAT men vi ser på en formel i stedet for en krets.

$$SAT = \{\langle\phi\rangle : \ \phi \text{ er en tilfredsstillende boolean formel}\}$$

SAT vil altså inneholde standard kodinger av boolean formler som er tilfredsstillende. For eksempel kan $\phi = \Big((x_1 \to x_2) \vee \neg\big((\neg x_1 \leftrightarrow x_3) \vee x_4\big)\Big) \wedge \neg x_2$. Dersom tildelingen av variabler er $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$ vil denne formelen returnere 1 og derfor vil denne formelen ϕ være en del av SAT. **SAT av boolean formler er NP komplett**. *Bevis:*

- $\boldsymbol{SAT \in NP}$ – et sertifikat som består av variabelverdier som tilfredsstiller ϕ kan verifiseres på polynomial tid.
- **SAT er NP-hard fordi $CIRCUIT\text{-}SAT \leq_P SAT$** – må vise at enhver instans av CIRCUIT-SAT kan reduseres til en instans av SAT på polynomial-tid. Dette gjøres ved å vise at en krets C kan omformes til en boolean formel ϕ, på en slik måte at C er tilfredsstilt kun hvis ϕ er tilfredsstilt

Merk: $L_1 \leq_P L_2$ dersom $x \in L_1$ kun hvis $f(x) \in L_2$

3-CNF SAT

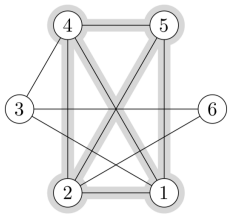
3-CNF blir ofte brukt som NP-komplett problem for å bevise andre NP-komplette problem, fordi den er mer begrenset en CIRCUIT-SAT. CNF er en konjunksjon (AND) av disjunkte klausuler (OR). En boolean formel er k-CNF dersom den er en CNF av nøyaktig k variabler, dvs. det er k variabler i hver OR-klausulene. For eksempel vil en 3-CNF være $(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$. **3-CNF SAT problemet går ut på å bestemme om en logisk formel som er 3-CNF er tilfredsstillende, dvs. kan gi output 1 for bestemte variabelverdier.** I 3-CNF SAT problemet vil instansen være en logisk formel på 3-CNF form, og spørsmålet er om formelen kan være sann (om bestemte variabelverdier vil gi output 1) :

$$3\text{-}CNF\ SAT = \{\langle\phi\rangle : \ \phi \text{ er en tilfredsstillende boolean formel på formen } 3\text{-}CNF\}$$

3-CNF SAT vil altså inneholde standard kodinger av boolean formler som er tilfredsstillende og har 3-CNF form. **3-CNF SAT er NP komplett**. *Bevis:*

- **3-CNF $\boldsymbol{SAT \in NP}$** – et sertifikat som består av variabelverdier som tilfredsstiller ϕ kan verifiseres på polynomial tid.
- **3-CNF SAT er NP-hard fordi $SAT \leq_P$ 3-CNF SAT** – må vise at enhver instans av SAT kan reduseres til en instans av 3-CNF SAT på polynomial-tid. Dette gjøres ved å vise at formelen ϕ kan omformes til en formel ϕ' med 3-CNF form, på en slik måte at ϕ vil være tilfredsstilt (output 1) kun hvis ϕ' er tilfredsstilt.

Merk: $L_1 \leq_P L_2$ dersom $x \in L_1$ kun hvis $f(x) \in L_2$



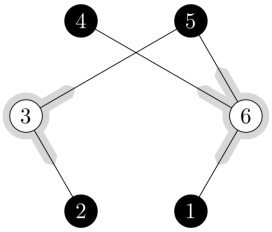
CLIQUE

En **clique** i en urettet graf $G = (V, E)$ er et subsett $V' \subseteq V$ der det er et vertekspar mellom alle verteksene. (se figur). Det er altså en komplett delgraf og størrelsen er antall vertekser den inneholder. **CLIQUE problemet går ut på å bestemme om en gitt graf G inneholder en clique med størrelse k** (optimaliseringsproblem: finne clique med maksimum størrelse). I CLIQUE problemet vil instansen være en urettet graf G og et heltall k, og spørsmålet er om G har en komplett delgraf med k vertekser:

$$CLIQUE = \{\langle G, k\rangle : \ G \text{ inneholder en clique av størrelse } k\}$$

CLIQUE vil altså inneholde standard kodinger av grafen G og k som inneholder en clique av størrelse k. **CLIQUE er NP komplett**. *Bevis:*

- $\boldsymbol{CLIQUE \in NP}$ – et sertifikat for grafen $G = (V, E)$ vil være en clique $V' \subseteq V$ som kan verifiseres ved å sjekke at størrelsen er k og at for hvert par $u, v \in V'$ vil kanten $(u, v) \in E$. Dette kan gjøres på polynomial tid.
- $\boldsymbol{CLIQUE}$ **er NP-hard fordi 3-CNF** $\boldsymbol{SAT \leq_P CLIQUE}$ – må vise at enhver instans av 3-CNF SAT kan reduseres til en instans av CLIQUE på polynomial-tid. Dette gjøres ved å vise at en boolean formel ϕ med 3-CNF form kan omformes til en graf G, der k er antall klausuler i ϕ. Vi har at ϕ vil være tilfredsstillende kun hvis G har en clique med størrelse k



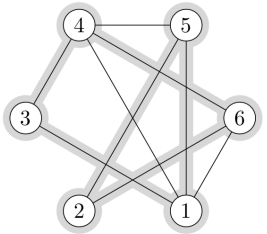
VERTEX-COVER

Et **verteksdekke** (*vertex cover*) i en urettet graf $G = (V, E)$ er et subsett $V' \subseteq V$ der $(u, v) \in E$ gir at $u \in V'$ og/eller $v \in V'$. På figuren ser vi at $V' = \{3, 6\}$ og dette "dekker" alle kantene i G. Størrelsen til verteksdekket er antall vertekser. **VERTEX-COVER problemet går ut på å bestemme om en gitt graf G inneholder et verteksdekke med størrelse k** (optimaliseringsproblem: finne verteksdekke med minimum størrelse). I VERTEX-COVER problemet vil instansen være en urettet graf G og et heltall k, og spørsmålet er om G har et verteksdekke med k vertekser:

$$VERTEX\text{-}COVER = \{\langle G, k\rangle : \ G \text{ inneholder et verteksdekke av størrelse } k\}$$

VERTEX-COVER vil altså inneholde standard kodinger av grafen G og k som inneholder et verteksdekke av størrelse k. **VERTEX-COVER er NP komplett**. *Bevis:*

- $\boldsymbol{VERTEX\text{-}COVER \in NP}$ - et sertifikat for grafen $G = (V, E)$ vil være et verteksdekke $V' \subseteq V$ som kan verifiseres ved å sjekke at størrelsen er k og at for hver kant $(u, v) \in E$ vil $u \in V'$ og/eller $v \in V'$. Dette kan gjøres på polynomial tid.
- $\boldsymbol{VERTEX\text{-}COVER}$ **er NP-hard fordi CLIQUE** $\boldsymbol{\leq_P VERTEX\text{-}COVER}$ – må vise at enhver instans av CLIQUE kan reduseres til en instans av VERTEX-COVER på polynomial-tid. Denne reduksjonen avhenger av komplementet $\overline{G}$, og vi har at graf G vil inneholde en clique av størrelse k kun vis graf $\overline{G}$ har et verteksdekke av størrelse $|V| - k$.



HAM-CYCLE

En Hamilton syklus i en urettet graf er en simpel syklus som inneholder alle verteksene i V. **HAM-CYCLE problemet går ut på å finne ut om en graf G inneholder en Hamilton syklus**. I HAM-CYCLE problemet vil instansen være en urettet graf G og spørsmålet er om G inneholder en syklus som inneholder alle nodene:

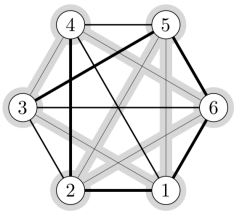
$$HAM\text{-}CYCLE = \{\langle G\rangle : G \text{ er en Hamilton syklus}\}$$

HAM-CYCLE vil altså inneholde standard kodinger av grafen G som inneholder en Hamilton syklus. **HAM-CYCLE er NP komplett**. *Bevis:*

- $\boldsymbol{HAM\text{-}CYCLE \in NP}$ - et sertifikat for grafen $G = (V, E)$ vil være sekvensen av $|V|$ vertekser som utgjør en Hamilton syklus. Verifiseringen består av å sjekke at sekvensen inneholder alle verteksene nøyaktig én gang og at den danner en syklus med kanter som er i E. Dette kan gjøres på polynomial tid.
- $\boldsymbol{HAM\text{-}CYCLE}$ **er NP-hard fordi** $\boldsymbol{VERTEX\text{-}COVER \leq_P HAM\text{-}CYCLE}$– må vise at enhver instans av VERTEX-COVER kan reduseres til en instans av HAM-CYCLE på polynomial-tid. Dette gjøres ved å vise at en graf G med heltall k kan omformes til en graf G', slik at G vil ha et verteksdekke av størrelse k kun hvis G' inneholder en Hamiltonian syklus

Merk: $L_1 \leq_P L_2$ dersom $x \in L_1$ kun hvis $f(x) \in L_2$

TSP (Traveling-Salesman Problem)



En selger ønsker å dra på en rundtur for å besøke alle bøyene n nøyaktig én gang og ende opp i byen der turen startet (se figur). Dette kan modelleres som en komplett graf med n vertekser, der kantene har en kostnad $c(i, j)$. **TSP problemet går ut på å bestemme om grafen G inneholder en rundtur med maksimal kostnad k** (optimaliseringsproblem: finne en rundtur med minst mulig totalkostnad). I TSP problemet vil instansen være en urettet graf G, en kostnadsfunksjon c og et heltall k, og spørsmålet er om G har en rundtur med totalkostnad $\leq k$:

$$TSP = \{\langle G, c, k\rangle : G \text{ inneholder en rundtur med maks totalkostnad } k\}$$

TSP vil altså inneholde standard kodinger av grafen G, c og k som inneholder en rundtur med maks totalkostnad k. **TSP er NP komplett**. *Bevis:*

- $\boldsymbol{TSP \in NP}$ - et sertifikat for grafen $G = (V, E)$ vil være en sekvens av $n = |V|$ vertekser i en rundtur. Verifiseringen består av å sjekke at sekvensen inneholder alle verteksene nøyaktig én gang og at summen av kostnaden til kantene er maks k. Dette kan gjøres på polynomial tid.
- $\boldsymbol{TSP}$ **er NP-hard fordi** $\boldsymbol{HAM\text{-}CYCLE \leq_P TSP}$ – må vise at enhver instans av $HAM\text{-}CYCLE$ kan reduseres til en instans av TSP på polynomial-tid. Dette gjøres ved å omforme instansen til HAM-CYCLE G til en graf $G' = (V, E')$ med kostnadsfunksjon c, slik at grafen G inneholder en Hamilton syklus kun hvis G' inneholder en rundtur med maks totalkostnad k.

Merk: $L_1 \leq_P L_2$ dersom $x \in L_1$ kun hvis $f(x) \in L_2$

SUBSET-SUM

Et endelig sett S består av positive heltall, og subsett-summen vil være summen av heltallene i et subsett $S' \subseteq S$. **SUBSET-SUM problemet går ut på å bestemme om settet S inneholder et subsett som har sum lik en target verdi $t > 0$**. I SUBSET-SUM problemet vil instansen være en mengde positive heltall S og et positivt heltall t, og spørsmålet er om det finnes en delmengde $S' \subseteq S$ som har sum t:

$$SUBSET\text{-}SUM = \{\langle S, t\rangle : \text{det eksisterer et subsett } S' \subseteq S \text{ slik at } t = \textstyle\sum_{s \in S'} s\}$$

SUBSET-SUM vil altså inneholde standard kodinger av settet S og heltall t, som har et subsett med sum t. **SUBSET-SUM er NP komplett**. *Bevis:*

- $\boldsymbol{SUBSET\text{-}SUM \in NP}$ - et sertifikat for settet S vil være et subsett $S' \subseteq S$, slik at $t = \sum_{s \in S'} s$, noe som kan verifiseres på polynomial tid.

- **$SUBSET\text{-}SUM$ er NP-hard fordi 3-CNF $SAT \leq_P SUBSET\text{-}SUM$** – må vise at enhver instans av 3-CNF SAT kan reduseres til en instans av $SUBSET\text{-}SUM$ på polynomial-tid. Dette gjøres ved å omforme formelen ϕ med k klausuler til en instans $\langle S, t \rangle$ slik at ϕ er tilfredsstillende kun hvis S har et subsett med sum t

0-1 Ryggsekk- og longest-path problemet

Tidligere i kompendiet har det blitt nevnt at 0-1 ryggsekkproblemet og longest-path problemet ikke kan løses på polynomisk tid. Vi skal nå se at disse er NP-harde. Et problem er NP-hardt dersom $L' \leq_P L$ for alle $L' \in NP$. Dette kan bevises ved å vise at alle NP problem kan reduseres til L eller ved å vise at ett NP-komplett problem kan reduseres til L.

Forstå at 0-1-ryggsekk-problemet er NP-hardt og hvorfor løsningen ikke er polynomisk

0-1 Ryggsekkproblemet – NP-hardt

På side 92 og 98 så vi at 0-1 ryggsekkproblemet går ut på at en tyv raner en butikk med n varer og vil enten ta eller ikke ta varen når ryggsekken kan bære totalt W gram. Vi har funnet en algoritme som løser dette problemet, med kjøretid $\Theta(nW)$. Dette er polynomisk kjøretid som funksjon av n og W. Dersom vi ikke oppgir noen eksplisitte parametre derimot, så antas kjøretiden å være en funksjon av input-størrelsen. Hvordan vi måler denne størrelsen avhenger av problemet, men når vi regner på om ting kan løses i polynomisk tid holder vi oss til antall bits i input. Dette skyldes at dataprogrammet krever at probleminstansene representeres på en måte som dataprogrammet forstår, og derfor vil probleminstansene kodes til binære strenger (s. 152). Dersom W har m antall bits, vil kjøretiden bli:

$$T(n, m) = \Theta(n2^m)$$

Som vi kan se vil ikke dette være en polynomisk kjøretid, men heller en superpolynomial. **Vi skal nå vise at 0-1 ryggsekkproblemet er NP-komplett.**

Vi begynner med å formulere problemet som et språk. 0-1 Ryggsekkproblemet er et optimaliseringsproblem der vi ønsker å finne hvilke varer som skal tas for å maksimere verdien. Dette kan omformes til et beslutningsvalg ved å inkludere et heltall k, slik at **problemet blir å bestemme hvordan vi kan velge varer ut fra n mulige slik at vekten ikke er større enn W og verdien overgår en gitt verdi k.** Probleminstansen vil derfor være antall varer n, vekten W og verdien k:

$$\text{0-1 } KNAPSACK = \{\langle n, W, k \rangle : \text{det eksisterer et valg av varer fra } n \text{ mulige med vekt} < W \text{ og verdi} > k\}$$

0-1 KNAPSACK vil altså inneholde standard kodinger av n, W og et heltall k, slik at varer kan velges med vekt mindre eller lik W og verdi over k. **0-1 KNAPSACK er NP komplett**. *Bevis:*

- **0-1 $KNAPSACK \in NP$** - et sertifikat vil være en mengde varer som har vekt mindre eller lik W og verdi større enn k. Verifiseringen består av å summere vekten til de oppgitte varene og sjekke at dette ikke er større enn W og å sjekke at den samlede verdien overgår k. Dette kan gjøres på polynomial tid.
- **0-1 $KNAPSACK$ er NP-hard fordi $SUBSET\text{-}SUM \leq_P$ 0-1 $KNAPSACK$**– må vise at enhver instans av $SUBSET\text{-}SUM$ kan reduseres til en instans av 0-1 $KNAPSACK$ på polynomial-tid. Instansen til SUBSET-SUM vil være et subsett ($S' = \langle s_1, s_2, \dots s_n \rangle$) og en verdi t som subsettet skal summeres til. Denne kan

reduseres til en instans av 0-1 KNAPSACK problemet ved å la $s_i = c_i$ og $t = k$, der c_i representerer verdien til vare i. Da vil S ha et subsett med sum t kun hvis valget av varer har verdi som overgår k.

Dermed har vi vist at 0-1 KNAPSACK problemet er NP-komplett (og derfor NP-hardt)!

Longest-path problemet – NP-hardt

På side 88 så vi at longest-path problemet har ingen polynomial-tid løsning. Longest-path problemet er NP-hardt, noe som kan vises at en reduksjon av HAM-CYCLE problemet.

Forstå at lengste enkle-vei-problemet er NP-hardt

Vi skal nå vise at longest-path problemet NP-komplett.

Vi begynner med å formulere problemet som et språk. Longest-path problemet er et optimaliseringsproblem der vi ønsker å finne hvilken bane av vertekser som består av størst antall kanter. Dette kan omformes til et beslutningsvalg ved å inkludere et heltall k, slik at **problemet blir å bestemme om banen i grafen har lengde større enn $k \leq |V|$.** Probleminstansen vil derfor være en graf G og verdien k:

$LONGEST\text{-}PATH = \{\langle G, k\rangle : G$ inneholder en bane som har lengde større enn $k\}$

LONGEST-PATH vil altså inneholde standard kodinger av G og et heltall k, der G har en bane med lengde større enn k. **LONGEST-PATH er NP komplett**. *Bevis:*

- $\boldsymbol{LONGEST\text{-}PATH \in NP}$ - et sertifikat vil være en graf G som inneholder en bane med lengde større enn k. Verifiseringen går ut på å sjekke at banen inneholder samme verteks kun én gang, består av kanter som er i E og at den har lengde større enn k. Dette kan gjøres på polynomial tid.
- $\boldsymbol{LONGEST\text{-}PATH}$ **er NP-hard fordi** $\boldsymbol{HAM\text{-}CYCLE \leq_P LONGEST\text{-}PATH}$– må vise at enhver instans av $HAM\text{-}CYCLE$ kan reduseres til en instans av $LONGEST\text{-}PATH$ på polynomial-tid. Instansen til HAM-CYCLE vil være en urettet graf $G = (V, E)$ som vi kan redusere til en urettet graf $G' = (V', E')$ der $k = |V|$. Da vil G inneholde en Hamilton syklus kun hvis G' har en longest-path med lengde k (siden da vil det være en bane som besøker alle verteksene).

Dermed har vi vist at LONGEST-PATH problemet er NP-komplett (og derfor NP-hardt)!