

# Innhold

# Hva er Algoritmer (m.m)?

**En uformell definisjon:** En algoritme er en hvilket som helst **tydelig definert fremgangsmåte** som kan ta en verdi (eller en mengde verdier) som **input** og gir en verdi (eller en mengde verdier) som **output**. Det er en sekvens som transformerer input til output.

**Beskrivelse av løsning:** Det finnes ingen verdensomspennende standard for å beskrive en algoritme. Du kan beskrive den med naturlig språk, som et dataprogram eller til og med ved hjelp av å tegne maskinvare (hardware) som løser problemet. Det eneste kravet er at du er presis når du beskriver.

**Instanser:** Hver samling av input-verdier til et slikt problem kalles en **instans**. For eksempel kan man i en instans av problemet over ha input-verdier som er et veikart over Trondheim, og de to punktene kan være to geografiske punkter som tilsvarer NTNU Gløshaugen og NTNU Dragvoll.

**Riktig eller uriktig?**: En algoritme kan være enten **riktig** eller **uriktig**. Dersom en algoritme er *riktig*, vil den for alle tenkbare instanser (innenfor området vi har bestemt) gi riktig output; f.eks. er det bare å forvente at en sorteringsalgoritme ikke vil ha problem med å sortere alle mulige samlinger av positive heltall. Dersom den er *uriktig*, vil den ikke gjøre det. F.eks. kan du ha funnet på en algoritme som løser alle sorteringsproblemer ved å vende rekken av tall du har fått baklengs. Dette vil for noen instanser gi riktig output, men bare for en brøkdel av alle mulige instanser.

**Problem:** Et problem er en relasjon mellom input og output.

**Probleminstans:** En probleminstans er en bestemt input.

**In-place:** En algoritme er in-place når den opererer på input dataen uten å matte lage feks nye array for å løse problemet.

**Stabil:** At en algoritme er stabil vil si at hvis du sorterer en liste med tall, vil alltid tallet i forekomsten som var først i den opprinnelige listen komme først i den sorterte listen.

**Rekursjon:** Deler opp i mindre problemer, løser de og konstruerer til slutt en fullstendig løsning ut fra del-løsningene (bruker induksjonstrinn).

- Merk: Bruker et induktivt premiss om at du kan løse de mindre problemene

**Pseudopolynomialitet:** Gitt en algoritme som tar tallet  $n$  som input og har kjøretid  $O(n)$  – hvilken kompleksitetsklasse er dette?  $n$  betegner her ikke størrelsen på input, men er selv en del av inputen. Det vil si at størrelsen til  $n$  = antall bits som kreves =  $\lg(n)$ .

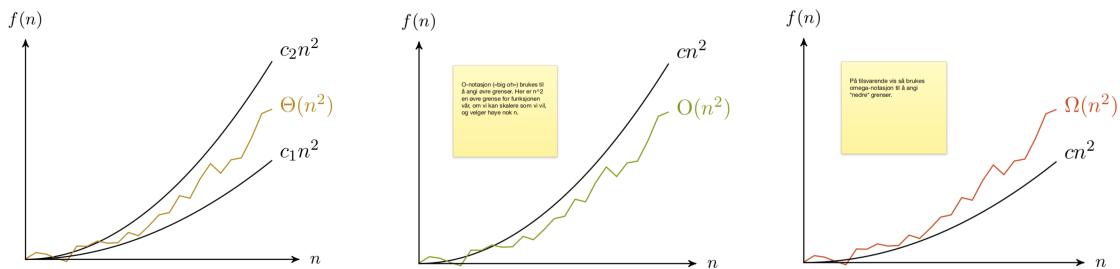
$$n = 2^{\lg(n)} = 2^\omega \Rightarrow \text{NB: Eksponentiell kjøretid!}$$

Dukker ofte opp som lureoppgave på eksamen!

**Asymptoisk notasjon:** Notasjon hvor man dropper konstanter og lavere ordens ledd

Notasjon	Kjøretid	Betydning
$o(n)$	$a < b$	<p>Strengere enn <math>O</math></p> <ul style="list-style-type: none"> <li><math>o(g(n)) = \{ f(n) : \text{for enhver positiv konstant } c &gt; 0, \text{ eksisterer det en konstant } n_0 &gt; 0 \text{ slik at } 0 \leq f(n) &lt; cg(n) \text{ for alle } n \geq n_0\}</math></li> </ul> <p>For eksempel: <math>2n = o(n^2)</math>, men <math>2n^2 \neq o(n^2)</math></p>
$O(n)$	$a \leq b$	<p>gir en øvre grense for kjøretiden:</p> <ul style="list-style-type: none"> <li><math>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\}</math></li> </ul>
$\Theta(n)$	$a = b$	<p>gir øvre og nedre grense for kjøretiden:</p> <ul style="list-style-type: none"> <li><math>\Theta(g(n)) = \{ f(n) : \text{det eksisterer positive konstanter } c_1, c_2 \text{ og } n_0 \text{ slik at } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for alle } n \geq n_0\}</math></li> </ul>
$\omega(n)$	$a > b$	<p>Strengere enn <math>\Omega</math></p> <ul style="list-style-type: none"> <li><math>\omega(g(n)) = \{ f(n) : \text{for enhver positiv konstant } c &gt; 0, \text{ eksisterer det en konstant } n_0 &gt; 0 \text{ slik at } 0 \leq cg(n) &lt; f(n) \text{ for alle } n \geq n_0\}</math></li> </ul> <p>For eksempel: <math>\frac{n^2}{2} = \omega(n)</math>, men <math>\frac{n^2}{2} \neq \omega(n^2)</math>.</p>
$\Omega(n)$	$a \geq b$	<p>gir en nedre grense for kjøretiden:</p> <ul style="list-style-type: none"> <li><math>\Omega(g(n)) = \{ f(n) : \text{det eksisterer positive konstanter } c \text{ og } n_0 \text{ slik at } 0 \leq cg(n) \leq f(n) \text{ for alle } n \geq n_0\}</math></li> </ul>

### Illustrasjon:



## Asymptotiske egenskaper:

### Symmetry:

$f(n) = \Theta(g(n))$  if and only if  $g(n) = \Theta(f(n))$ .

### Transpose symmetry:

$f(n) = O(g(n))$  if and only if  $g(n) = \Omega(f(n))$ ,

$f(n) = o(g(n))$  if and only if  $g(n) = \omega(f(n))$ .

Dersom  $f(n) = o(g(n))$  vet vi at

- $g(n) = \Theta(f(n))$
- $g(n) = \Omega(f(n))$
- $g(n) = O(f(n))$
- $f(n) = O(g(n))$
- $g(n) = \omega(f(n))$

## Kjøretid

Kjøretiden til en algoritme er et mål på hvor effektiv en algoritme er og uten tvil det viktigste.

### Om kjøretider

**Hvorfor trenger vi kjøretid?**: I en verden der datamaskiner var uendelig raske og du hadde uendelig med tilgjengelig lagringsplass, kunne du brukt en hvilken som helst *richtig* algoritme for å løse et problem – uansett hvor lite gjennomtenkt den måtte være. Men, som du sikkert har merket, er vi ikke i denne verdenen. Derfor har vi kjøretider.

**Forklaring av kjøretid**: En kjøretid beskriver det asymptotiske forholdet mellom størrelsen på problemet og hvor lang tid det vil ta å løse det.

## Noen vanlige kjøretider

Kompleksitet	Navn	Type
$\Theta(n!)$	Factorial	Generell
$\Omega(k^n)$	Eksponensiell	Generell
$O(n^k)$	Polynomisk	Generell
$\Theta(n^3)$	Kubisk	Tilfelle av polynomisk
$\Theta(n^2)$	Kvadratisk	Tilfelle av polynomisk
$\Theta(n \lg n)$	Loglineær	Kombinasjon av lineær og logaritmisk
$\Theta(n)$	Lineær	Generell
$\Theta(\lg n)$	Logaritmisk	Generell
$\Theta((\log_b n)^k)$	Polylogaritmisk	?
$\Theta(1)$	Konstant	Generell

## Kjøretidsberegning

Rekursjon er en problemløsningsmetodikk som baserer seg på at løsningen på et problem er satt sammen av løsningene til mindre instanser av samme problem. Denne teknikken kommer til å gå igjen i mange av algoritmene i pensum.

Et av de aller vanligste eksemplene på rekursivitet er Fibonaccitallene, definert som:

$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0, F(1) = 1$$

Fibonaccitall n er altså definert som summen av de to foregående Fibonaccitallene.

Gode eksempler på rekursive algoritmer er Merge og Quick Sort og binærsøk. Vi finner også ofte rekursive løsninger ved bruk av dynamisk programmering.

## Split & hersk

Divide: Del problemet opp i subproblemer som er mindre instanser av det samme problemet.

Conquer: Løs subproblemene rekursivt, når subproblemene blir små nok (base case) kan vi løse de rett frem.

Combine: Kombiner løsningene av subproblemene til en løsning av problemet. Hvis det er et problem om ikke er en del av den rekursive løsningen antas den å være en del av combine-problemet.

## Variabelskifte

Bytt ut  $m = \lg n$  for å få enklere likninger. Får da  $n = 2^m$ , og  $T(n) = T(2^m) = S(m)$ . Får da vanlige rekursjoner, uten  $\lg n$  og andre vanskelige saker. Kan løse med iterasjonsmetoden, masterteoremet eller substitusjon, og så bytte tilbake til originale variabler.

## Masterteoremet

Masterteoremet er en kokebokløsning for å finne kjøretiden til (mange) rekurrensen på formen

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$a \geq 1, b > 1$$

Merk: en viktig egenskap vi kan utnytte når vi jobber med master teoremet er at  $a^{\log n} = n^{\log a}$ .

Denne typen rekurrens oppstår gjerne i sammenheng med splitt-og-hersk algoritmer, f.eks. Merge sort. Problemet deles opp i  $a$  deler av størrelse  $n/b$ , med  $f(n)$  arbeid for å gjøre selve oppdelinga, og å sette sammen resultatet av rekursive kall etter at disse er ferdige. I eksempelet med Merge sort, er  $f(n)$  arbeidet med å splitte lista i to underlister, og å flette sammen de to sorterte listene etter at de rekursive kallene er ferdige. Det å splitte skjer i konstant tid  $\Theta(1)$ , mens det å flette sammen tar lineær tid  $\Theta(n)$ . Vi kan altså sette  $f(n)=n$ . Siden vi til enhver tid deler listen opp i to deler, hver del  $n/2$  er henholdsvis  $a=2$  og  $b=2$ . For Merge sort har vi altså:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Dersom vi ikke allerede visste kjøretiden til Merge sort, kunne vi funnet den ved å løse denne rekurrensen. Å løse rekurrensen kunne vi så brukt Masterteoremet til.

Fremgangsmåten for Masterteoremet er som følger:

1. Identifiser  $a, b, f(n)$
2. Regn ut  $\log_b a$
3. Konsulter tabellen under

Tabell over de tre tilfellene av Masterteoremet:

Tilfelle	Krav	Løsning
1	$f(n) \in O(n^{\log_b a - \epsilon})$	$T(n) \in \Theta(n^{\log_b a})$
2	$f(n) \in \Theta(n^{\log_b a} \log^k n)$	$T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$
3	$f(n) \in \Omega(n^{\log_b a + \epsilon})$	$T(n) \in \Theta(f(n))$

Merk:  $\epsilon > 0$

Vi fortsetter eksempelet med Merge sort. Vi har funnet  $a=2$ ,  $b=2$ ,  $f(n)=n$ , da må  $\log_b a = \log_2 2 = 1$ . Vi har altså tilfelle 2,  $f(n) \in \Theta(n^1)$  med løsning  $T(n) \in \Theta(n \lg n)$ .

Og på norsk betyr dette

- 1) Hvis  $f(n) < n^{\log_b a}$  med en polynomisk faktor, så er  $T(n) = \Theta(n^{\log_b a})$
- 2) Hvis  $f(n) = n^{\log_b a}$  (samme polynomiske faktor), så er  $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$
- 3) Hvis  $f(n) > n^{\log_b a}$  med en polynomisk faktor OG  $af(n/b) \leq cf(n)$ , så er  $T(n) = \Theta(f(n))$

### **Eksempel**

Enda et eksempel, hentet fra kontinuasjonseksemene 2009:

Løs følgende rekurrens. Oppgi svaret i asymptotisk notasjon. Begrunn svaret kort.

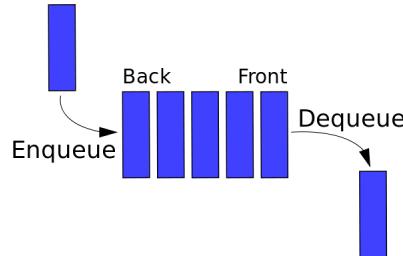
$$T(n) = 2T\left(\frac{n}{3}\right) + \frac{1}{n}$$

Vi har  $a=2$ ,  $b=3$ ,  $f(n)=n^{-1}$ . Det gir  $\log_b a=\log_3 2 \approx 0.63$ . Siden  $f(n)=n^{-1}$  er  $O(n^{63})$ , har vi tilfelle 1 og løsningen  $T(n) \in \Theta(n^{63})$ .

# Grunnleggende datastrukturer

## Stacks and Queues (Dynamic Sets)

Stakker og køer er dynamiske sett med 2 viktige metoder, PUSH og POP, som betyr hhv. å legge til og fjerne et element. Merk at et POP-kall og et DELETE-kall er det samme.



### Queue

En queue er en FIFO (First In First Out) struktur. En kø implementeres som oftest som et array (gjerne et dynamisk array, likt Java sin ArrayList). En kø har egenskapene ENQUEUE og DEQUEUE. Dette vil si at et Dequeue-kall til køen returnerer elementet som først ble satt inn.

#### Enqueue

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

#### Dequeue

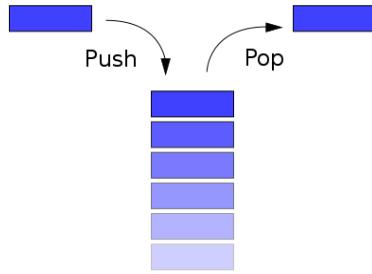
```
DEQUEUE( $Q$ )
1  $x = Q[Q.head]$ 
2 if  $Q.head == Q.length$ 
3    $Q.head = 1$ 
4 else  $Q.head = Q.head + 1$ 
5 return  $x$ 
```

Insert-metode, kallas enqueue for en kø. *Tail*-variabelen peker alltid på neste ledige plass i arrayet, så når vi enqueueer legger vi bare elementet rett inn. Vi må oppdatere *tail*-variablen, hvis den er lik lengden av lista settes den til 1, hvis ikke inkrementeres den. Man må egentlig også sjekke for overflow, altså at køen er full. Da kan en sjekke om  $head == tail + 1$  (teknisk sett er det en ledig plass i køen da, men dette defineres for å kunne avgjøre når køen er tom).

Vi sparer på det fremste elementet. Så setter vi *head* til å være 1 dersom vi er på enden av lista, hvis ikke inkrementeres *head*. Så returneres elementet som nettopp ble fjernet. Man må her egentlig sjekke for underflow-rror, altså at lista er tom. Da kan en sjekke om

$head == tail$ .

## Stack



En stakk er LIFO (Last In First Out) struktur. Stack implementeres oftest med et array, eller en lenket liste. En stack har egenskapene INSERT, PUSH og POP (DELETE). Et POP-kall til stakken returnerer elementet som sist ble satt inn.

### Stack-empty

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

Stakken har en variabel  $top$  som peker på hvor i arrayet det øverste elementet ligger. Når denne er null er stakken tom, og stack-empty returnerer true.

### Push

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

Insert-operasjon, kalles push for stakker.

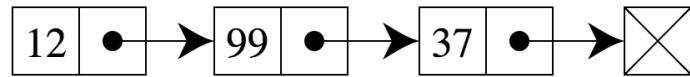
Øker  $top$ -variablen med en, og legger til det nye elementet øverst i stakken. Dersom  $top$  blir større enn størrelsen på lista vil det utløse en error (overflow).

### Pop

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

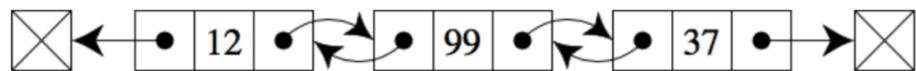
Delete-operasjon, kalles pop for stakker. Hvis stakken er tom utløses en underflow-error. Hvis ikke minkes  $top$  med en og det elementet som nettopp ble fjernet blir returnert. Merk at den ikke tar inn noen elementer å fjerne, en stakk fjerner alltid det øverste elementet.

## Lenkede lister



Objektliknende, har en verdi og en peker til neste og forrige element i lista. Lagres ikke i et array, men som enkeltelementer som har en peker til neste og forrige element. Operasjonene blir ofte enkle, da det for det meste handler om å omdirigere pekerne.

I en dobbelt-lenket liste peker det også på det forrige elementet. Hvert objekt har helst to peker-variable, next og prev.



### List-search

**LIST-SEARCH( $L, k$ )**

```

1  $x = L.\text{head}$ 
2 while  $x \neq \text{NIL}$  and  $x.\text{key} \neq k$ 
3      $x = x.\text{next}$ 
4 return  $x$ 
```

Søker lineært gjennom lista etter element med nøkkel  $k$ , hvis den ikke finnes, returneres NIL. Worst-case er at den må søke gjennom hele lista, altså  $\Theta(n)$ .

### List-insert

**LIST-INSERT( $L, x$ )**

```

1  $x.\text{next} = L.\text{head}$ 
2 if  $L.\text{head} \neq \text{NIL}$ 
3      $L.\text{head}.\text{prev} = x$ 
4      $L.\text{head} = x$ 
5      $x.\text{prev} = \text{NIL}$ 
```

Insert-metode, setter elementet først i lista. Må da sette next-variabelen til det vi skal insertere til å peke på head, så må head

sin prev peke på elementet vi skal legge til. Head må settes til å peke på elementet, og prev blir satt til NIL. Alt dette gjøres i  $O(1)$  tid.

### List-delete

**LIST-DELETE( $L, x$ )**

```

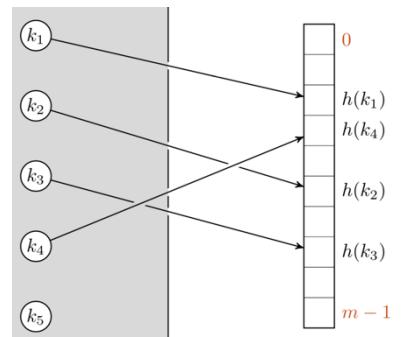
1 if  $x.\text{prev} \neq \text{NIL}$ 
2      $x.\text{prev}.\text{next} = x.\text{next}$ 
3 else  $L.\text{head} = x.\text{next}$ 
4 if  $x.\text{next} \neq \text{NIL}$ 
5      $x.\text{next}.\text{prev} = x.\text{prev}$ 
```

Omdirigere pekerne slik at elementet blir fjerna, og next og prev til elementet peker på hverandre og ikke  $x$ . Hvis vi ønsker å fjerne et element med en gitt nøkkel må vi først kalle på List-Search.

## Kjøretider

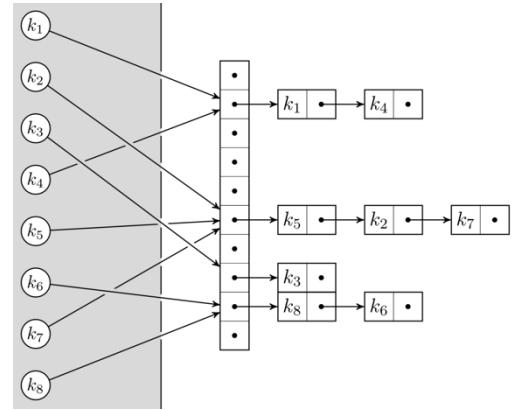
Handling	Kjøretid
Innsetting på starten	$O(1)$
Innsetting på slutten	$O(n)$
Oppslag	$O(n)$
Slette element	oppslugstid + $O(1)$

## Hashtabeller



Bruker modfisert nøkkel  $k$  som indeks  $h(k)$ . Vi ønsker å tillate store nøkler uten å ha store tabeller. Vi kan da «knøvle» nøkkelen til å bli en akseptabel, tilsynelatende tilfeldig, indeks. Obs: Input til en hashfunksjon kan være en vilkårlig bitstreng, tolket som et tall. Vi kan godt hashe andre ting som strenger og mer kompliserte objekter. Vi står opp objektene inn i hashfunksjonen og får en gyldig indeks ut. Det er mulig at samme nøkkel hasher til samme verdi, da har vi en kollisjon. Kan løses med kjedding (se under). Dersom  $x.key = m$  og  $h(m) = j$  der  $h$  er en hashfunksjon vil  $x$  være elementet,  $m$  nøkkelen og  $j$  hashen.

## Kjeding (Chaining)



Om to verdier hasher til samme indeks, sa har vi en kollisjon; for å ta vare på begge verdiene, kan vi ha en lenket liste (f.eks.) i hver celle i tabellen.

- Mange kollisjoner: Lineært lange lister
  - Søk vil ta lineær tid
- Anta lineært stor tabell
- Anta jevn, «tilfeldig» fordeling
  - Konstant forventet kjøretid!
- Statisk datasett? Lag custom hashfunksjon!
  - Kan da garantere konstant kjøretid

Definerer load factor  $\alpha = n/m$ ,  $n$ =elementer,  $m$ =nøkler, altså gjennomsnittlig antall elementer per nøkkel. Worst-case hvis alle elementer hasher til samme nøkkel,  $\Theta(n)$ .

### Pseudokode:

```

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  $T[h(x.key)]$ 
```

**Amortisert analyse:** Gjennomsnittstiden en operasjon tar, fordi worst-case kan være for pessimistisk. Si worst-case har en viss kjøretid, men etter at worst-case har skjedd én gang er det veldig lenge til det kan skje igjen (f.eks. å fordoble et array for å ha plass til flere elementer). Amortisert analyse summerer kjøretidene for ulike input og situasjoner, og deler deretter på antall ganger den ble kjørt. Se på gjennomsnitt per operasjon etter at mange har blitt utført (gjennomsnitt av kjøretiden):  $\sum_{i=0}^{h-1} 2^i = 2^h - 1$

## Hashfunksjoner

### The division method

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

Vi tar altså resten etter å ha delt på m, som er størrelsen på tabellen. Er noen verdien for m man ikke burde velge, f.eks. 2p, mens et primtall ikke for nærmere 2p er ofte et godt valg.

### The multiplication method

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

A er et tall mellom 0 og 1, tar den fraksjonelle delen av kA og ganger med m, størrelsen på tabellen, runder ned til nærmeste heltall. ( $kA \bmod 1 = kA - \lfloor kA \rfloor$ ).

## Dynamiske tabeller

Tabeller som kan øke og minke i størrelse utifra hvor mange elementer det inneholder. Definerer load faktor = antall elementer i tabellen delt på antall plasser. Sier noe om hvor mye plass i tabellen som blir brukt. Vil ikke at den skal være for lav, og når den nærmer seg en må vi utvide tabellen.

Hva om en hashtabell blir for full? Eller hva med en stakk eller kø? Vi kan alllokere nytt minne og kopiere, men det tar jo lineær tid ... så vi vil gjøre det sjeldent! Vi tar i, og allokerer mye minne

### Table-insert

```

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  $2 \cdot T.size$  slots
6      insert all items in  $T.table$  into new-table
7      free  $T.table$ 
8       $T.table = new-table$ 
9       $T.size = 2 \cdot T.size$ 
10     insert  $x$  into  $T.table$ 
11      $T.num = T.num + 1$ 

```

$T.size$  = antall plasser i tabellen

$T.num$  = antall plasser brukt i tabellen

$T.table$  = tabellen

Hvis size er null oppretter vi tabellen og setter size til 1. Hvis num==size er tabellen full og vi lager en ny tabell med dobbelt så mange plasser og kopierer over elementene. Frigjør den gamle tabellen og lagrer den nye i  $T.table$ . Uavhengig av dette, til slutt settes elementet inn i tabellen og num økes med 1.

# Sortering

Stort sett kategoriserer vi sorteringsalgoritmer inn i to grupper; sammenligningsbaserte og distribuerte/ikke-sammenligningsbaserte.

## Stabilitet

En sorteringsalgoritme kan sies å være stabil dersom rekkefølgen av like elementer i listen som skal sorteres blir bevart i løpet av algoritmen. Altså at for de elementene som er like, vil det elementet som var først i input-arrayet være først i output-arrayet. Eksempel, dersom vi har listen:

$$[B1, C1, C2, A1]$$

og sorterer den med en eller annen algoritme, og de to like elementene står i samme rekkefølge etter sorteringen, har vi en stabil sorteringsalgoritme.

$$[A1, B1, C1, C2]$$

Altså har vi at selv om verdien til C1 og C2 er den samme så blir C1 sortert før C2.

## Sammenligningsbaserte sorteringsalgoritmer

Alle sorteringsalgoritmer som baserer seg på å sammenligne to elementer for å avgjøre hvilket av de to som skal komme først i en sekvens, kaller vi sammenligningsbasert. Sammenlikningsbasert sortering: baserer sorteringen kun på sammenlikninger mellom elementene. Alle sammenlikningsbaserte sorteringsalgoritmer har en nedre grense på  $\Omega(n \lg n)$ . Heapsort og merge sort er optimale sammenlikningsbaserte sorteringsalgoritmer siden de har en øvre grense  $O(n \lg n)$ . Eks. på sammenlikningsbaserte sorteringsalgoritmer er insertion sort, merge sort, quicksort.

For  $n$  elementer i en liste, finnes det  $n!$  permutasjoner. For hvert ja/nei-spørsmål vi stiller, dobler hva vi kan skille mellom (halveres hva vi står igjen med).

$$2^{T(n)} \geq n!$$

$$T(n) \geq \lg n!$$

$\lg n!$  er antall halveringer fra  $n!$  til 1. For maks uflaks kan vi ikke gjøre det bedre med sammenlikningsbaserte sorteringsalgoritmer, fordi vi må sammenlikne  $\lg n!$  ganger. Blir altså en nedre grense.

$$T_w(n) = \Omega(\lg n!) = \Omega(n \lg n) *$$

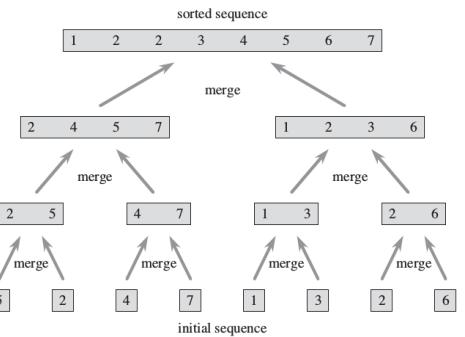
\* Stirlings approksimasjon:  $n! \geq (n/e)^n \Rightarrow \lg n! \geq n \lg n - n \lg e = \Omega(n \lg n)$ .

## Merge sort

Merge sort er en splitt-og-hersk algoritme. Merge Sort er effektiv. Deler opp problemet i stadig mindre biter, og når bitene er små nok flettes de sammen i sortert rekkefølge. Mergesort gjør altså sorteringsarbeidet før det rekursive kallet. Merk: kan ikke bruke DP på merge sort siden det ikke er noen overlappende delproblemer.

Metode: «Divide & conquer», merging, sammenligningsbasert

Best Case	Average Case	Worst Case	Stabil	Space
$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	Ja	$O(n)$



**Figure 2.4** The operation of merge sort on the array  $A = \{5, 2, 4, 7, 1, 3, 2, 6\}$ . The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

1. Sorter venstre halvdel rekursivt
  - a) Todel listen rekursivt til hver liste inneholder 1 element\*
2. Sorter høyre halvdel rekursivt
  - a) Todel listen rekursivt til hver liste inneholder 1 element\*
3. Flett sammen underlistene.

\* (En liste med ett element er per definisjon sortert)

I praksis blir dette gjerne implementert rekursivt. Merge funksjonen kjører i lineær tid. For å analysere kjøretiden til Merge sort kan vi sette opp følgende rekurrens:

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

Altså 2 ganger tiden det tar å sortere en liste av lengde  $n/2$  pluss tiden det tar å flette de to listene. Se [Masterteoremet](#) for hvordan vi kan løse denne rekurrensen og finne kjøretiden  $O(n \lg n)$ .

De fleste implementasjoner av Merge sort er stabile. På grunn av at algoritmen hele tiden lager nye underlistene, trenger den  $O(n)$  ekstra plass i minnet.

**Pseudokode:**

```

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(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

```

D&C → merge sort

## Quicksort

Quick Sort er enda en “split-og-hersk” -algoritme. Man starter gjerne Quick Sort ved å randomisere lista. Den starter med å velge et pivotelement. Den deler deretter lista i to partisjoner: en med elementene som er mindre eller lik pivoten, og en med elementene som er større enn pivot. Deretter kaller den seg selv rekursivt på de to partisjonene. Deretter fletter man sammen de to partisjonene. Quicksort gjør altså sorteringsarbeidet før det rekursive kallet. Krav om at elementene i lista må kunne sammenliknes.

Metode: «Divide & conquer», partisjonering, sammenligning, in-place

Best Case	Average Case	Worst Case	Stabil	Space
$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	Nei	$O(n \lg n)$

Konseptet med et pivot-element står sentralt i quicksort. For hver iterasjon velger en seg et pivot-element og sorterer resten av elementene basert på om de er større eller lavere enn pivot-elementet. (Hvis pivot-elementet velges tilfeldig kalles algoritmen Randomized-Quicksort.)

### Fremgangsmåte:

- Divide:** Del opp arrayer i to subarrays, slik at hvert element i det første er mindre eller lik en pivot x, og hvert element i det andre er større enn x.
- Conquer:** Sorter subarrayene utifra om de er større eller mindre enn piv, ved å rekursivt kalle quicksort
- Combine:** Arrayene er sortert, og alle elementene i det ene er mindre enn alle elementene i det andre. Hele arrayet A er dermed sortert.

Kjøretidsanalysen av Quicksort blir noe vanskeligere enn for Merge sort, ettersom størrelsen på listene som sorteres rekursivt vil avhenge av hvilken pivot vi velger. Det å velge en god pivot er en kunst i seg selv. Den naive fremgangsmåten med å slavisk velge det første elementet, kan lett utnyttes av en adversary, ved å gi en liste som er omvendt sortert som input. I dette tilfellet vil kjøretiden bli  $\Theta(n^2)$ . Vi kan motvirke dette ved å velge en tilfeldig pivot i stedet.

**QUICKSORT( $A, p, r$ )**

```

1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3       $\text{QUICKSORT}(A, p, q - 1)$ 
4       $\text{QUICKSORT}(A, q + 1, r)$ 

```

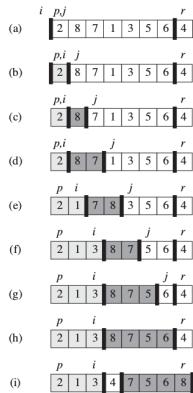
**PARTITION( $A, p, r$ )**

```

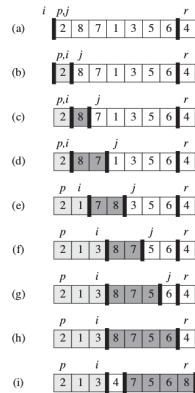
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6      exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 

```

## Pseudokode: Partition



**Figure 7.1** The operation of PARTITION on a sample array. Array entry  $A[r]$  becomes the pivot element  $x$ . Lightly shaded array elements are all in the first partition with values no greater than  $x$ . Heavily shaded elements are in the second partition with values greater than  $x$ . The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot  $x$ . (a) The initial array and variable settings. None of the elements have been placed in either of the first two partitions. (b) The value 2 is “swapped with itself” and put in the partition of smaller values. (c)–(d) The values 2 and 7 are added to the partition of larger values. (e) The values 1 and 8 are swapped, and the smaller partition grows. (f) The values 3 and 7 are swapped, and the smaller partition grows. (g)–(h) The larger partition grows to include 5 and 6, and the loop terminates. (i) In lines 7–8, the pivot element is swapped so that it lies between the two partitions.



**Figure 7.1** The operation of PARTITION on a sample array. Array entry  $A[r]$  becomes the pivot element  $x$ . Lightly shaded array elements are all in the first partition with values no greater than  $x$ . Heavily shaded elements are in the second partition with values greater than  $x$ . The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot  $x$ . (a) The initial array and variable settings. None of the elements have been placed in either of the first two partitions. (b) The value 2 is “swapped with itself” and put in the partition of smaller values. (c)–(d) The values 2 and 7 are added to the partition of larger values. (e) The values 1 and 8 are swapped, and the smaller partition grows. (f) The values 3 and 7 are swapped, and the smaller partition grows. (g)–(h) The larger partition grows to include 5 and 6, and the loop terminates. (i) In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

## Worst-case

Får et subproblem med  $n-1$  elementer i det ene arrayet og 0 i det andre. Hvis dette skjer for hvert rekursivt kall, og en partition tar  $\Theta(n)$  tid, vil kjøretiden være gitt ved

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

Dette er det samme som insertion sort, og skjer når arrayet allerede er sortert (eller omvendt sortert).

## Best-case

Hvis den deler arrayet midt på hver gang, blir størrelsen på subarrayene  $n/2$  og  $n/2-1$ , og vi får kjøretid

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

## Quicksort (Randomized)

En randomisert versjon av quicksort. Velg pivot (dele-element) tilfeldig fra arrayet istedenfor å alltid ta det bakerste. Den krever at elementene i lista må kunne sammenliknes. Antar at elementene er distinkte. Algoritmen randomiseres, slik at samme dårlige input ikke alltid gir dårlig kjøretid.

### Pseudokode:

RANDOMIZED-QUICKSORT( $A, p, r$ )

- 1 **if**  $p < r$
- 2      $q = \text{RANDOMIZED-PARTITION}(A, p, r)$
- 3     RANDOMIZED-QUICKSORT( $A, p, q - 1$ )
- 4     RANDOMIZED-QUICKSORT( $A, q + 1, r$ )

RANDOMIZED-PARTITION( $A, p, r$ )

- 1  $i = \text{RANDOM}(p, r)$
- 2 exchange  $A[r]$  with  $A[i]$
- 3 **return** PARTITION( $A, p, r$ )

Randomized-partition velger tilfeldig pivot, bytter så bakerste element med dette. Kall på originale partition.

### Worst-case og best-case

Samme som Quicksort

## Heapsort

Heapsort benytter seg av en heap som datastruktur. Bygger en max-heap av alle elementene slik at hver node sine barn er mindre enn den selv. Det øverste elementet er alltid det største. Deretter plukkes det øverste elementet ut, for så å sortere heapen igjen, og ta ut det øverste elementet igjen. Fortsetter til heapen er tom.

**Input:** Sekvens med tall,  $a_1, a_2, \dots, a_n$ .

**Output:** Permutasjon av tallene slik at  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

Metode: Seleksjon, sammenligningsbasert, prioritetskø, in-place

BC	AC	WC	SC	Stable
$\Omega(n \lg n)$	$\Theta(n \lg n)$	$O(n \lg n)$	$O(1)$	Nei

## Pseudokode

:

```

HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 for  $i = A.length$  downto 2
3   exchange  $A[1]$  with  $A[i]$ 
4    $A.size = A.size - 1$ 
5   MAX-HEAPIFY( $A, 1$ )
MAX-HEAPIFY( $A, i$ )
1  $l = \text{LEFT}(i)$ 
2  $r = \text{RIGHT}(i)$ 
3 if  $l \leq A.size$  and  $A[l] > A[i]$ 
4    $m = l$ 
5 else  $m = i$ 
6 if  $r \leq A.size$  and  $A[r] > A[m]$ 
7    $m = r$ 
8 if  $m \neq i$ 
9   exchange  $A[i]$  with  $A[m]$ 
10  MAX-HEAPIFY( $A, m$ )
BUILD-MAX-HEAP(A)
1  $A.size = A.length$ 
2 for  $i = \lfloor A.length/2 \rfloor$  downto 1
3   MAX-HEAPIFY( $A, i$ )

```

## Fremgangsmåte:

Først bygger vi en max-heap. Siden rota til heapen er det største elementet, legger vi dette bakerst i lista, dekrementerer heap-size-variabelen, som begrenser heapen til  $A[1..heap.size]$ . Nå er altså det som var rota, plassert bakerst, og ikke lengre med i heapen (selv om det er med i lista). Elementet som var bakerst, blir plassert i rota, så vi må kalle på max-heapify for denne nye rota.

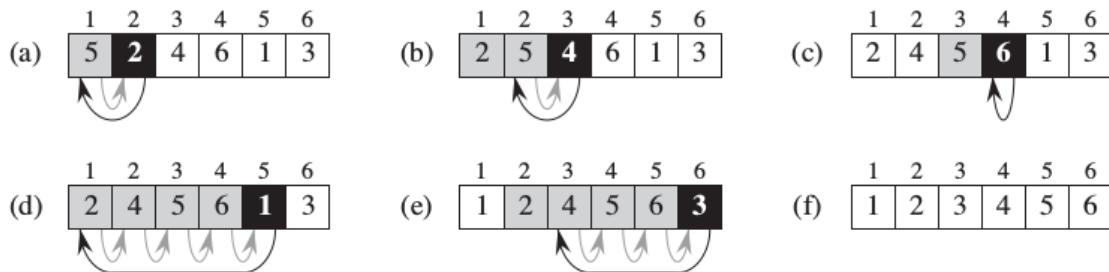
Se heaps for forklaring på Max-heapify og Build-max-heap.

## Insertion Sort

**For usortert:** Går gjennom alle elementene i A, tar ut element j, setter i til å være elementet før j. Sålenge element i er større enn element j flyttes element i et hakk til høyre i A, og trekker fra 1 fra i. Når element i ikke lenger er større enn element j, settes element j inn i A rett etter i. For små inputstørrelser og nesten sorterte lister er den veldig rask, men for større inputstørrelser og lite sortert blir den veldig treg, og man burde heller bruke andre algoritmer. Kan feks. brukes i slutten på en Quick Sort algoritme. Insertion sort krever at elementene som skal sorteres må kunne sammenliknes.

Metode: innsetting, in-place

**For sortert:** Setter inn element i på rett plass i den allerede sorterte lista av i-1 elementer.



**Figure 2.2** The operation of `INSERTION-SORT` on the array  $A = \{5, 2, 4, 6, 1, 3\}$ . Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. (a)–(e) The iterations of the `for` loop of lines 1–8. In each iteration, the black rectangle holds the key taken from  $A[j]$ , which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. (f) The final sorted array.

### INSERTION-SORT( $A$ )

```

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

BC	AC	WC	SC	Stable
$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	Ja

## Pseudokode

## Ikke-sammenligningsbaserte sorteringsalgoritmer

Sorteringsalgoritmer som ikke er sammenligningsbaserte er ikke begrenset av  $O(n \lg n)$  som nedre grense for kjøretid.

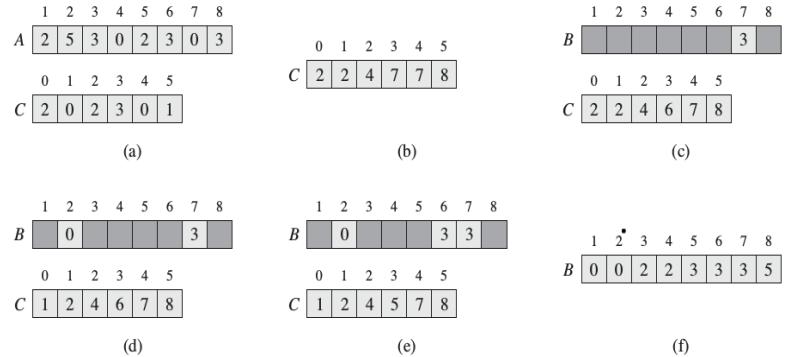
### Counting sort

Counting Sort antar at input er et heltall  $N$  mellom 0 og  $k$ . Lager en liste med verdier fra 0 til  $k$ . Videre teller counting sort hvor mange elementer som er lavere eller lik det elementet som skal sorteres, og putter det på riktig plass i lista. Dette er en stabil søkealgoritme. Fungerer best når verdiene på tallene som sorteres ligger tett etter hverandre og  $k$  ikke er for høy. Antar at inngangselementene  $n$  er heltall mellom 0 og  $k$ , for et heltall  $k$ . Når  $k = O(n)$ , bruker algoritmen  $\Theta(n)$  tid.

Metode: Ikke-sammenligningsbasert

BC	AC	WC	SC	Stable
$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$	Ja

Dersom  $k = O(n)$ , får vi kjøretid  $\Theta(n)$



**Figure 8.2** The operation of COUNTING-SORT on an input array  $A[1..8]$ , where each element of  $A$  is a nonnegative integer no larger than  $k = 5$ . (a) The array  $A$  and the auxiliary array  $C$  after line 5. (b) The array  $C$  after line 8. (c)–(e) The output array  $B$  and the auxiliary array  $C$  after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array  $B$  have been filled in. (f) The final sorted output array  $B$ .

### Pseudokode:

```

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 for i = 1 to k
7   C[i] = C[i] + C[i - 1]
8 for j = A.length downto 1
9   B[C[A[j]]] = A[j]
10  C[A[j]] = C[A[j]] - 1
    
```

C er et array som holder styr på hvor mange elementer som er mindre enn x. A er input og B er output. Først teller vi hvor mange ganger A[j] forekommer i A, lagrer dette i C, slik at  $C[A[j]]$  inneholder antall ganger A[j] forekommer i A. Så summerer vi  $C[i] = C[i] + C[i-1]$ , altså hvor mange elementer som kommer før C[i], dette lagres igjen i C. Så kan vi plassere tallene i B. Gå gjennom A baklengs, tallet i A[j] skal stå på posisjon  $C[A[j]]$  i B, og så må vi dekrementere tallet i tilfelle det var flere av samme tall.

## Radix sort

Radix Sort sorterer tall i et gitt tallsystem (her titallsystemet) etter minst signifikante siffer. Radix sort antar at input er  $n$  elementer med  $d$  siffer, der hvert siffer kan ha opp til  $k$  forskjellige verdier. Algoritmen ser som regel på det minst signifikante sifferet, sorterer med hensyn på dette sifferet, og gjentar så med det nest minst signifikante sifferet, osv. Om sortseringen på hvert siffer er basert på en algoritme som sorterer stabilt på  $\Theta(n+k)$  (feks counting sort), vil hele algoritmen bruke  $\Theta(d(n+k))$  tid.

Sorterer først på minst signifikante nummer, så nest minste osv til mest signifikante nummer. Slik får vi sortering innad i sorteringen.

Metode: Ikke-sammenligningsbasert

BC	AC	WC	SC	Stable
$\Omega(d(n+k))$	$\Theta(d(n+k))$	$O(d(n+k))$	$O(n+k)$	Ja*

**Pseudokode:**

```
RADIX-SORT(A, d)
1  for  $i = 1$  to  $d$ 
2      sort* A by digit  $d$ 
```

\* Radix sort krever en stabil subroutine for selv å være stabil.

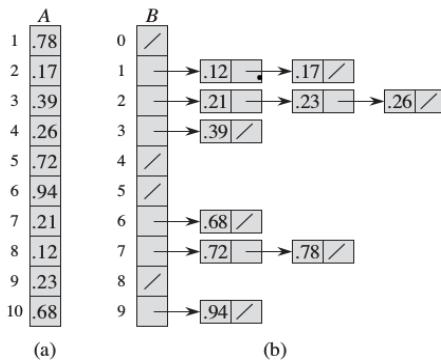
Gitt  $n$   $d$ -siffer tall der hvert siffer kan ha  $k$  ulike verdier, er kjøretiden  $\Theta(d(n+k))$ . For  $d$  siffer, sorteres lista for hvert siffer, så det kommer an på sorteringsalgoritmen brukt. Det er naturlig for dette tilfellet å bruke counting sort, som bruker  $\Theta(n+k)$  tid, gjør vi dette  $d$  ganger får vi en kjøretid på  $\Theta(d(n+k))$ . For konstant  $d$  og  $k=O(n)$  får den lineær kjøretid.

## Bucket sort

Veldig lik Counting Sort, men bruker sakalte "bøtter" man putter tallene i basert på tallenes relative størrelse. For eksempel [5, 6) betyr at verdier større eller lik 5, men mindre enn 4 skal i bøtta. Bucket sort antar at inputen er generert fra en tilfeldig (random) prosess som fordeler elementene uniformt og uavhengig over et intervall. Bucket sort deler intervallet inn i  $n$  like store bøtter" (intervaller), og fordeler så de  $n$  inputtallene i bøttene. Hver bøtte sorteres så for seg ved å bruke en sorteringsalgoritme: insertion sort (vanligvis) eller en annen algoritme. Om man bruker Bucket-Sort på  $n$  uavhengig uniformt fordelt tall i området  $[0, 1)$ , så får man en kjøretid på  $O(n)$ . Forventet kjøretid for insertion-sort vil i dette tilfellet være  $O(2 - \frac{1}{n})$ , evt  $O(1)$ .

Metode: ikke-sammenligningsbasert

BC	AC	WC	SC	Stable
$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$	Ja



**Figure 8.4** The operation of BUCKET-SORT for  $n = 10$ . (a) The input array  $A[1..10]$ . (b) The array  $B[0..9]$  of sorted lists (buckets) after line 8 of the algorithm. Bucket  $i$  holds values in the half-open interval  $[i/10, (i + 1)/10)$ . The sorted output consists of a concatenation in order of the lists  $B[0], B[1], \dots, B[9]$ .

## Pseudokode

:

```

BUCKET-SORT(A)
1   $n = A.length$ 
2  create  $B[0..n - 1]$ 
3  for  $i = 1$  to  $n$ 
4      make  $B[i]$  an empty list
5  for  $i = 1$  to  $n$ 
6      add  $A[i]$  to  $B[\lfloor nA[i] \rfloor]$ 
7  for  $i = 0$  to  $n - 1$ 
8      sort list  $B[i]$ 
9  concatenate  $B[0] \dots B[n - 1]$ 

```

Går gjennom tallene, og legger de i en av  $n$  like store subintervaller, bøtter. Deretter sorteres bøttene hver for seg, med insertion sort siden det ikke er mange elementer i hver bøtte (siden det er uniform distribusjon). Så setter den sammen listene og så er det sortert. Hver bøtte blir en linket liste.

Krever uniform distribusjon, og ikke for mange elementer, da vi bruker insertion sort for å sortere innad i bøttene.

Kan også få til lineær tid uten uniform distribusjon, hvis inputen har den egenskapen at kvadratet av bøtte-størrelsene er lineære i forhold til totalt antall elementer.

## Søking

Sortering og søking er to problemstillinger som går igjen enten som problem i seg selv, eller som underproblem. Med sortering mener vi det å arrangere en mengde med objekter i en spesifikk rekkefølge. Med søking mener vi det å finne ett spesifikt objekt i en mengde objekter. Det å finne et spesifikt element i en datastruktur er et utbredt problem. I en liste av tall kan vi f.eks. lete etter medianen eller bare et bestemt tall. I en graf kan vi f.eks. lete etter en måte å komme oss fra ett element til et annet. Søkealgoritmer for grafer (DFS og BFS) står under [Graf-algoritmer](#).

For å søke i lister, har vi primært to fremgangsmåter; brute force og binærssøk.

### Brute force

Denne fremgangsmåten er ganske selvforklarende. Gå gjennom lista (sortert eller ikke) fra begynnelse til slutt og sammenlign hvert element mot elementet du ønsker å finne. Med  $n$  elementer i lista blir kjøretiden  $O(n)$

### Binærssøk

Er en effektiv algoritme for å finne ønsket tall. Krav: listen  $L$  er sortert  
Metode kort oppsummert: Er det du leter etter i første halvdel

- 1) Hvis ja: Let videre der
- 2) Hvis nei: Let videre i den andre halvdelen

Si at vi leter etter en verdi  $a$  i en sortert liste  $L$ . Vi begynner med å finne det midterste elementet i  $L$  og sjekker om det er lik elementet vi leter etter. Dersom ja, returner indeksen, dersom ikke, avgjør om elementet i leter etter er større enn eller mindre enn det midterste, og gjenta deretter prosedyren i den underlista der elementet må være, dersom det i det hele tatt ligger i lista.

Siden vi for hver iterasjon todeler listen, og kan forkaste halvparten får vi følgende kjøretider:

Best Case	Average Case	Worst Case
$O(1)$	$O(lgn)$	$O(lgn)$

### Pseudokode

:

Rekursiv:

```
BISECT(A, p, r, v)
1 if  $p \leq r$ 
2    $q = \lfloor (p + r)/2 \rfloor$ 
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 return NIL
```

Iterativ: Genrelt mer effektiv, da vi slipper ekstra kostnader med funksjonskall

```
BISECT'(A, p, r, v)
1 while  $p < r$ 
2    $q = \lfloor (p + r)/2 \rfloor$ 
3   if  $v \leq A[q]$ 
4      $r = q$ 
5   else  $p = q + 1$ 
6 return  $p$ 
```

## Randomized select

Finner det i-ende minste elementet i  $A[p \dots r]$ . Det velger ett element som en pivott og partisjoner data deretter. Men i stedet for å rekursivt splitte begge dellistene, går den bare inn i siden med elementet vi søker etter.

«Quicksort som binærsøk» - Bruker rekursjon induktivt og antar mindre instanser kan løses.

**Input:** Et sett A av n distinkte tall og et heltall i, der  $1 \leq i \leq n$ .

**Output:** Et element  $x \in A$  som er større enn  $i-1$  elementer i A. (Element nr i for sortert A)

Metode: in-place, «divide & conquer»

### Kode:

```
RANDOMIZED-SELECT(A, p, r, i)
1 if  $p == r$ 
2   return  $A[p]$ 
3    $q = \text{RANDOMIZED-PARTITION}(A, p, r)$ 
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,  $q + 1, r, i - k$ )
```

Merk: samme partition som i randomized-quicksort

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

Hvis  $p = r$ , er det bare ett element i lista, base case, vi returnerer dette. Hvis ikke deler vi opp lista i to på en tilfeldig heltall, beregner antall elementer i første halvdel av delinga.

Hvis  $i = k$ , har vi funnet det  $i$ -te største elementet, og kan returnere dette. Hvis ikke gjør vi et rekursivt kall på den siden av lista som inneholder det  $i$ -te elementet.

Worst-case skjer hvis vi deler opp lista rundt det største gjenværende elementet i lista. For  $n$  elementer i lista, og oppdeling tar  $n$  tid, blir da kjøretiden  $\Theta(n^2)$ .

## Select

Fungerer på samme måte som randomized-select, bare at den deler arrayet i to på en mer strukturert måte enn tilfeldig. Select brukes rekursivt for å finne det  $i$ -ende minste elementet i en lav/høy side (avhengig av størrelsen på partisjonen til medianen av medianen). Bruker partisjoneringsalgoritmen fra quicksort, men bruker elementer fra partisjonene som input. Den bestemmer det  $i$ -ende minste elementet i en input-array. Dette gjør den ved å dele arrayen i grupper på  $n/5$  med 5 elementer, finner deres respektive medianer, og finner igjen medianen av de  $n/5$  forskjellige medianene. Partisjonerer så arrayen rundt medianen-av-medianen. Fordel: Bedre deling av arrayet enn randomized select.

**Input:** Et sett  $A$  av  $n$  distinkte tall og et heltall  $i$ , der  $1 \leq i \leq n$ .

**Output:** Et element  $x \in A$  som er større enn  $i-1$  elementer i  $A$ . (Element nr  $i$  for sortert  $A$ )

Metode: «Divide & conquer», partisjoning

**Pseudokode:**

```

SELECT(A, p, r, i)
1 if  $p == r$ 
2     return  $A[p]$ 
3  $q = \text{GOOD-PARTITION}(A, p, r)$ 
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,  $p + 1, r, i - k$ )

```

```

GOOD-PARTITION(A, p, r)
1  $n = r - p + 1$ 
2  $m = \lceil n/5 \rceil$ 
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  $x = \text{SELECT}(B, 1, m, \lfloor m/2 \rfloor)$ 
9 return PARTITION-AROUND(A,  $p, r, x$ )

```

```

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

```

- 1) Få inn et array av størrelse  $n$ . Del disse inn i  $\lceil n/5 \rceil$  grupper med 5 elementer i hver, bortsett fra siste som får  $n \bmod 5$  elementer.
- 2) Sorter hver subliste med insertion sort. Finn medianen i hver av listene.
- 3) Kall på Select rekursivt for å finne medianen av medianene.
- 4) Kall på Partition (modifisert til å ta inn dele-elementet) med medianen av medianene.
- 5) Hvis  $i = \text{delelementet}$ , returner dette. Ellers, bruk Select rekursivt for å finne det  $i$ -te minste elementet i et av subarrayene fra Partition.

## Kompleksitet og Stabilitet for Sortering- og Søkealgoritmer

- n er antallet elementer som skal jobbes med.
- k er den høyeste verdien tallene kan ha.
- d er maks antall siffer et tall kan ha.
- BC == Best Case
- AC == Average Case
- WC == Worst Case

Algoritme	BC	AC	WC	In-place	Stable
Merge sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	Nei	Ja
R-Quick	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	Ja	Nei
Heap	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	Ja	Nei
Bubble	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	Ja	Ja
Insertion	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	Ja	Ja
Selection	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	Ja	Nei
Bucket	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	Nei	Ja
Counting	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	Nei	Ja
Radix	$\Omega(d(n+k))$	$\Theta(d(n+k))$	$O(d(n+k))$	Nei	Ja*
Select	$O(n)$	$O(n)$	$O(n)$	NA	NA
R-Select	$O(n)$	$O(n)$	$O(n^2)$	NA	NA

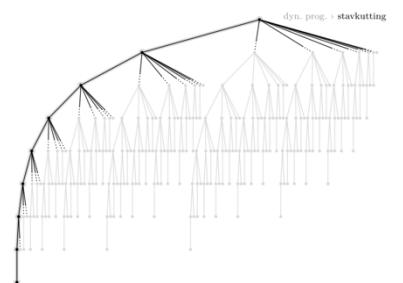
\* Radix sort krever en stabil subroutine for selv å være stabil.

## Algoritmeoversikt (Sorterings- og seleksjonsalgoritmer)

(Se bl. a. side 150 i boka)

	Stabil	In place	Best case	Av. case	Worst case
Bubble sort	Ja	Ja	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	Ja	Ja	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	Ja	Nei	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	Nei	Ja	$\Theta(n * \lg n)$	$\Theta(n * \lg n)$	$\Theta(n^2)$
Counting sort	Ja	Nei	$\Theta(k+n)$	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	Ja	Nei	$\Theta(d*(n+k))$	$\Theta(d*(n+k))$	$\Theta(d*(n+k))$
Bucket sort	Ja	Nei	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2)$
Heapsort	Nei	Ja	$O(n)$	$O(n \lg n)$	$O(n \lg n)$
Selection sort	Nei (Ja, m/lenket liste)	Ja	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Select			$O(n)$	$O(n)$	$O(n)$
Randomized select			$\Theta(n)$	$\Theta(n)$	$\Theta(n^2)$

## Dynamisk programmering



Dynamisk programmering brukes når delproblemene overlapper. Hvis en på visse problemer bruker standard splitt-og-hersk vil en løse samme problem flere ganger og dermed gjøre unødvendig arbeid.

Istedentfor å løse de samme problemene flere ganger mellomlagrer vi svarene til subproblemene, så vi ikke trenger å løse de mer enn én gang. Bruker det ofte til optimaliseringssproblemer (eks. korteste sti).

- 1) Karakteriser strukturen av en optimal løsning

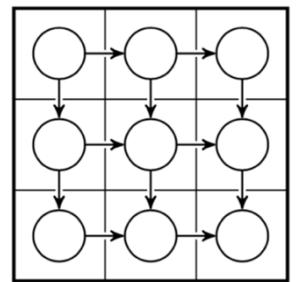
- 2) Rekursivt definer verdien til den optimale løsningen
- 3) Beregn verdien, ofte bottom-up (iterativt)
- 4) Konstruer optimal løsning fra de beregnede verdiene

For at vi skal kunne gjør dette må problemet ha optimal substruktur. Vanlige problemstillinger som kan løses vha DP - Longest common subsequence - Rod cutting  
 $DP \approx \text{rekursjon} + \text{memoisering} \Rightarrow \text{tid}/\text{subproblemer} = \Theta(1)$  (teller ikke rekursjoner)

Merk: at delproblemene løses rekursivt der de bygger på beste delløsning

Prøver å løse situasjoner der vi får overlappende delinstanser

De mørke linjene i bildet ovenfor er problemene løst én gang, de lyse er de samme problemene som de mørke, ser at vi sparer mye tid på memoisering.

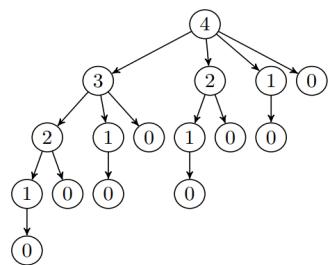


Viktig å tenke på «**Time-memory tradeoff**» - Tenk rutenett

Forenkling er å sette delproblemene i rutenett: Lar oss lagre løsninger i en tabell

Merk: Delproblemene kan bare løses i en rekkefølge (se piler for eksempel)

## Delproblemgraf



Graf som viser at et problem har de samme delproblemene, så det lønner seg å bruke dynamisk programmering. Alt med 0 er det samme problemet, alt med 1 osv. Blir tydelig at det samme problemet løses mange ganger, og at vi burde bruke dynamisk programmering.

## Optimal substruktur

Et problem har optimal substruktur hvis en optimal løsning inneholder optimale løsninger på delproblemer. Merk at størrelsen på delproblemene ikke er spesifisert, en kan altså ikke redusere de til deres enkleste komponenter som i [merge sort](#). Konstruerer optimal løsning basert på optimale delløsninger.

## Memoisering: top-down

Holde styr på tidligere beregnede verdier i en tabell, som sjekkes før man faktisk beregner det. Begynner som oftest med en stort problem, som deretter blir mindre og mindre utover i kjøringen. Kan finne løsningen raskere i spesialtilfeller hvis algoritmen ikke rekurserer til absolutt alle løsninger.

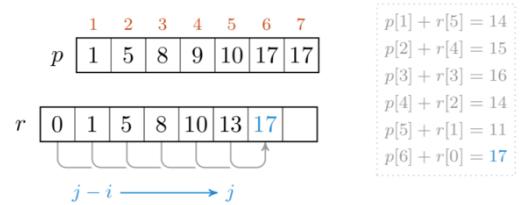
## Iterasjon: bottom-up

Løser de minste subproblemene først, og så de større, som er avhengig av de små. Vi har derfor løst alle subproblemene fra før når vi løser det store problemet. Mindre funksjonskall, så mindre overhea (de konstante faktorene er mindre).

## Overlappende delproblemer

Delproblemene er de samme for ulike problemer. Med Divide-and-Conquer vil vi da løse de samme problemene flere ganger. Dynamisk programmering løser dette ved å kun løse det én gang, og deretter lagre svaret til neste gang det samme problemet må løses. Sparer masse tid.

## Rod-cutting



Gitt en stav med lengde  $n$ , og en liste med priser for alle lengder kortere enn  $n$ . Avgjør hvordan staven kan kuttes opp som gir maksimal fortjeneste.

Input: En lengde  $n$  og priser  $p_i$  for lengder  $i = 1, \dots, n$

$$r[j] = \max_{i=1 \dots j} (p[i] + r[j-i])$$

Output: Lengder  $l_1, \dots, l_k$  der summen av lengder  $l_1, \dots, l_k$  er  $n$  og totalprisen  $r_n = p_{l1} + \dots + p_{lk}$  er maksimal

```
CUT-ROD( $p, n$ )
1  if  $n == 0$ 
2    return 0
3   $q = -\infty$ 
4  for  $i = 1$  to  $n$ 
5     $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ 
6  return  $q$ 
```

## Rekursiv top-down implementasjon

Vi går inch by inch, og finner maksimal pris ved å sammenlikne høyeste pris hittil funnet med prisen for en  $i$  inch stav pluss prisen for gjenværende del av staven, som vi finner rekursivt. Kjøretiden er eksponential,  $2^n$ , fordi metoden kaller på seg selv først  $n$  ganger, deretter  $n-1$  ganger osv, med de samme parameterne. Dette kan forbedres med dynamisk programmering.

## Longest common subsequence (LCS)

Y										dyn. prog. → lcs
	s	t	o	r	m	k	a	s	t	
0	0	1	2	3	4	5	6	7	8	9
a	0	0	0	0	0	0	0	1	1	1
t	0	0	1	1	1	1	1	1	1	2
o	0	0	1	2	2	2	2	2	2	2
m	0	0	1	2	2	3	3	3	3	3
m	0	0	1	2	2	3	3	3	3	3
a	0	0	1	2	2	3	3	4	4	4
k	0	0	1	2	2	3	4	4	4	4
t	0	0	1	2	2	3	4	4	4	5

X

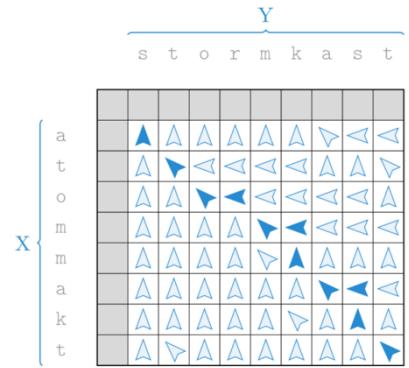
hvis  $x_i = y_j$

velg maks

Løses med dynamisk programmering fra bunnen og opp, ved å se på det siste elementet i hver liste.

Input: To sekvenser X med lengde m og Y med lengde n

Output: En sekvens Z med maksimal lengde k



Når man finner felles bokstaver for hver rad/rekke beholder man verdien på ruten ovenfor til venstre og inkrementerer plassen ij med 1. Man beholder ellers verdien i neste rute på måten beskrevet ved siden av.

Når man til slutt skal finne samsvarende ord ser man på hvilke delløsninger som bidro til løsningen (n, m) som på figuren til venstre. Deretter velger man ut elementene der ij er like og setter de sammen til et ord.

Fremgangsmåte:

1) Karakterisere lengste felles subsekvens

- a) Hvis  $x_m = y_n$ , så er  $z_k=x_m=y_n$  og  $Z_{k-1}$  er en LCS av  $X_{m-1}$  og  $Y_{n-1}$ .
- b) Hvis  $x_m \neq y_n$ , så impliserer  $z_k \neq x_m$  at  $Z$  er en LCS av  $X_{m-1}$  og  $Y$ .
- c) Hvis  $x_m \neq y_n$ , så impliserer  $z_k \neq y_n$  at  $Z$  er en LCS av  $X$  og  $Y_{n-1}$ .

Dette beviser at LCS har en optimal substruktur.

2) En rekursiv løsning

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases}$$

Der  $c$  er lengden av LCS,  $i$  er antall elementer i  $x$  og  $j$  er antall elementer i  $y$ . Den tar

for seg de tre tilfellene over, der den øverste er om enten  $x$  eller  $y$  har lengde null, den

i midten tar for seg tilfelle a. og den nederste tar for seg tilfelle b. og c., der LCS er den største av de to.

3) Beregne lengden til LCSen

```
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  $c[i - 1, j] \geq c[i, j - 1]$ 
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  $b$ 
```

Oppretter to tabeller, der b peker mot det optimale subproblemet som blir valgt, og c

er lengden til LCS-en. Går gjennom tabellene, og sjekker om hvilke av de tre tilfellene som er gjeldende. Piler peker altså mot den største av de tre indeksene som er til venstre, over eller skrått mot venstre. Der den peker skrått, den verdien skal være med i LCSen.

### Eksempel: Ryggsekk

Tenk at du har en ryggsekk med begrenset plass/kapasitet W og en mengde objekter du ønsker å ha i denne. Basert på verdiene  $v_1, \dots, v_n$  og tilhørende vekt/plass  $w_1, \dots, w_n$  hvert objekt tar ønsker du en optimal løsning hvor du får med mest verdi i sekken. Du må derfor gjøre en trade-off på verdi i forhold til vekt for hvert element. Om mulig: tenk inventory i Role Playing Game.

Kan løses både med DP og grådige valg. Her er det brukt LSC-algoritme:

```

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( $x, y$ )

```

	0	1	2	3	4	5	$w$	$v$
0	0	0	0	0	0	0	1	1
1	0	1	1	1	1	1	2	5
2	0	1	5	6	6	6	1	4
3	0	4	5	9	10	10	3	3
4	0	4	5	9	10	10	1	2
5	0	4	6	9	11	12	2	6
6	0	4	6	10	12	15		

Evt rekursiv:

```

KNAPSACK( $n, W$ )
1 if  $n == 0$ 
2   return 0
3  $x = \text{KNAPSACK}(n - 1, W)$ 
4 if  $W < w_n$ 
5   return  $x$ 
6 else  $y = \text{KNAPSACK}(n - 1, W - w_n) + v_n$ 
7   return max( $x, y$ )

```

Returnerer den maksimale verdien. Finner først den maksimale verdien dersom element n ikke er med, lagres i x. Hvis totalvekta er mindre enn vekta til element n, returneres da x. Hvis ikke, finner vi den maksimale verdien dersom element n er med, da må vi trekke fra vekta til n fra totalvekta, og så kalle Knapsack rekursivt, og legge til verdien til n til slutt. Vi returnerer så den maksimale verdien av disse to verdiene.

### 0-1 Knapsack-rekurrensen:

$$m[i, w] = m[i - 1, w] \text{ if } w < w_i$$

$$m[i, w] = \max(m[i - 1, w], m[i - 1, w - w_i] + v_i) \text{ if } w \geq w_i$$

## Memoisering DP-algoritme

Memoisere betyr i hovedsak å huske og gjennbruke løsninger fra subproblemer for å løse problemet: tid = # subproblemer

Et eksempel er en rekursiv fibonacci hvor man bruker en dictionary til å holde på tidligere utregnede subproblemer.

Legg merke til linje 6 (der skjer memorisering) i følgende generelle DP-algoritme:

```
FUNCTION'(A)
1 if F[A] == NIL
2     S = DIVIDE(A)
3     n = S.length
4     let R[1..n] be a new array
5     for i = 1 to n
6         R[i] = FUNCTION'(S[i])
7     F[A] = COMBINE(R)
8 return F[A]
```

# Grådige algoritmer

Noen ganger er dynamisk programmering overkill. Grådige algoritmer velger den delløsningen som ser best ut akkurat nå, og går videre. Den trenger altså ikke vite løsningen på alle mulige delproblemer før den kan gjøre det grådige valget. Dette vil føre til den optimale løsningen hvis problemet har Greedy-choice property: Det vil si at den velger en lokalt optimalt løsning i høve om at den også skal være globalt optimal. I tillegg må problemet ha optimal substruktur. Vanlige problemer som kan løses av GA er: - Aktivitetsutvalg - Huffman-kode

Merk: Løser det mest lovende delproblemet rekursivt og bygger løsningen på denne del-løsningen

Gradig valg + optimal delløsning = optimal løsning

Fordelen med grådige algoritmer fremfor DP er at algoritmen kan være enklere å implementere og ha bedre kjøretid.

## Greedy-choice property

Vi kan ta et valg som virker best i det øyeblikk og løse subproblemene som dukker opp senere. Valget gjort av grådige valg kan ikke være avh. av fremtidige valg/alle løsningene til problemet så langt

## Aktivitetsutvalg

Velg så mange ikke-overlappende aktiviteter som mulig. Anta at de er sortert etter sluttid og bruk en grådig algoritme til å velge dem. Merk: Vi trenger ikke se på alle disse delproblemene!

- Det vil alltid lønne seg å ta med intervallet som slutter først!

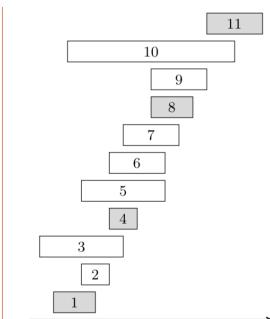
Rekursiv løsning

Iterativ omskriving

GREEDY-ACTIVITY-SELECTOR( $s, f$ )

```
1   $n = s.length$ 
2   $A = \{a_1\}$ 
3   $k = 1$ 
4  for  $m = 2$  to  $n$ 
5    if  $s[m] \geq f[k]$ 
6       $A = A \cup \{a_m\}$ 
7       $k = m$ 
```

```
REC-ACT-SEL( $s, f, k, n$ )
1   $m = k + 1$ 
2  while  $m \leq n$  and  $s[m] < f[k]$ 
3     $m = m + 1$ 
4  if  $m \leq n$ 
5     $S = \text{REC-ACT-SEL}(s, f, m, n)$ 
6    return  $\{a_m\} \cup S$ 
7  else return  $\emptyset$ 
```



## Huffmans algoritme

Huffmans algoritme er en grådig algoritme med formål å minimere lagringsplassen (uten tap) til en kjent sekvens med symboler. Hvert symbol kan representeres av en binær kode av lik lengde. Ved å telle antall hendelser av hvert symbol og bygge et binærtre basert på frekvensene kan man redusere antall bits som trengs til å lagre sekvensen med 20% -90%

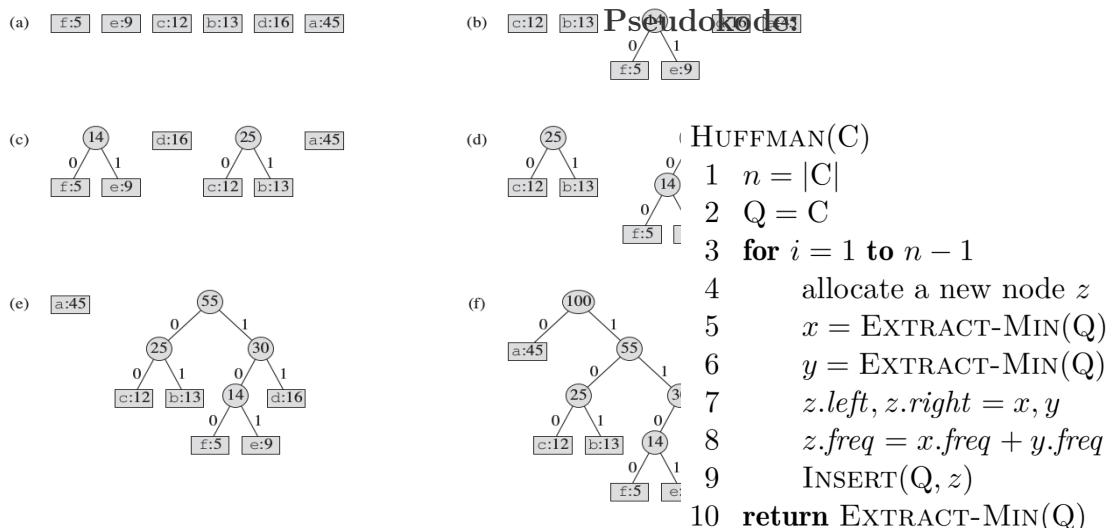
Fremgangsmåte:

1. Lag en node for hver unike bokstav og bygg en min heap av alle nodene.
2. Hent så ut to og to de nodene med lavest frekvens fra min heapen.
3. Lag en ny initiel node med frekvens lik summen av frekvensen til de to nodene. Legg alltid nodene med lavest frekvens til venstre og legg de til i en min heap.
4. Gjenta 2 og 3 til heapen inneholder bare en node. Den gjenværende noden er rotten.

Ved å videre å la en gren til venstre være 0 og en gren til høyre være 1 vil alle symbolene kunne representeres med en unik binær kode. De symbolene som brukes hyppigst vil få en kortere kode en de som brukes sjeldnere.

Best case	Average case	Worst case
$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$

En versjon av huffman kan i praksis se slik ut:



**Figure 16.5** The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of their children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. (a) The initial set of  $n = 6$  nodes, one for each letter. (b)-(e) Intermediate stages. (f) The final tree.

**Resultat:**

# Grafer

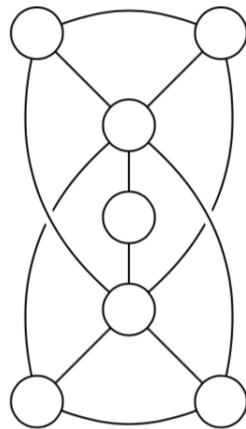
En graf modellerer parvise relasjoner mellom objekter. En graf er en slags helhetlig oversikt over mange små enkelte relasjoner. En graf består av noder(vertices) og kanter (edges). Fordi mange problemer kan modelleres som grafer er denne datastrukturen noe av det viktigste i faget.

Matematisk betegner vi ofte nodesetet  $V$  for vertices og kantsettet  $E$  for edges. En graf blir dermed

$$G = (V, E)$$

En graf kan være både rettet (directed) og urettet (undirected). I en urettet graf er det en kant  $(v, u)$  hvis det er en kant  $(u, v)$ . I en rettet graf er dette ikke nødvendigvis sant.

En graf (i dette tilfellet: urettet) kan se ut f.eks. som dette:



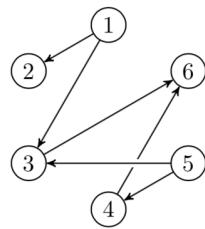
## Representasjon av grafer

Når en skal representere en graf på en datamaskin krever en et rammeverk som støtter pekerne "og "objektenes som en graf beskriver. De 2 vanligste måtene å gjøre dette er ved bruk av nabolister eller nabomatriser.

**Nabomatriser:** Raskere, men tar mer plass om få kaner. Egner seg til direkte oppslag. Representeres ved en  $(|V| \times |V|)$ -matrise. Kanter angis med et entall dersom det går en kant mellom to noder. Nabomatriser kan vi også modifisere til å representer vektede grafer ved å sette verdier annet enn 1 og 0.

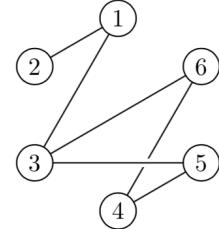
## Rettet

	1	2	3	4	5	6
1	1	1				
2						
3					1	
4					1	
5		1	1			
6						



Urettet: symmetrisk;  $A(u,v) = A(v,u)$

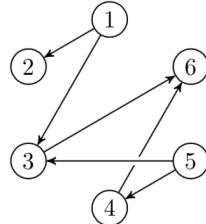
	1	2	3	4	5	6
1		1	1			
2	1					
3	1				1	1
4					1	1
5			1	1		
6			1	1		



**Nabolister:** Tar mindre plass om få kanter. Egner seg til traversering. Dette angir at det går en kant fra noden før kolon til hver av nodene i listen etter kolon.

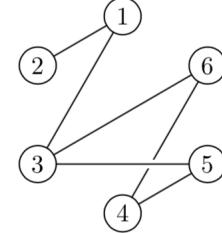
## Rettet

1	•	→	2	3
2	•			
3	•	→	6	
4	•	→	6	
5	•	→	3	4
6	•			

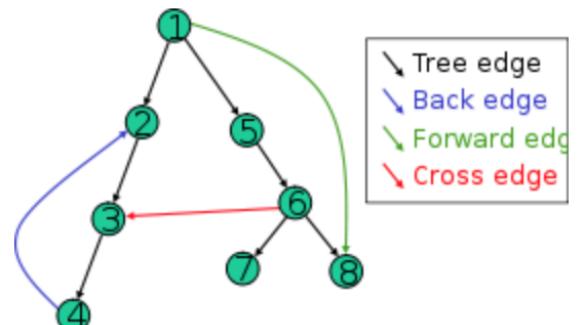


Urettet: Kanter går begge veier

1	•	→	2	3
2	•		1	
3	•	→	6	1
4	•	→	6	5
5	•	→	3	4
6	•		3	4



## Kant-klassifisering



Tree-Edge er en kant til en direkte etterfølger.

Forward-Edge er en kant til en indirekte etterfølger.

Back-Edge er en kant til en forgjenger.

Cross-Edge er en kant til en node som hverken er en etterfølger eller forgjenger.

Merk:

Møter en hvit node Tre-kant

Møter en går node Bakoverkant

Møter en svart node Forover- eller krysskant

## Parentesteoremet

For ethvert dybde-først søk av en graf  $G = (V, E)$ , for hvilke som helst noder  $u$  og  $v$ , er nøyaktig en av de følgende egenskapene sanne

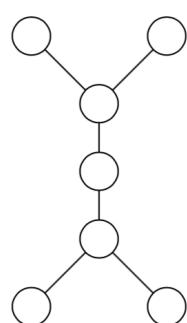
- Intervallene  $[u.d, u.f]$  og  $[v.d, v.f]$  er disjunkte, og verken  $u$  eller  $v$  er etterkommer av den andre.
- Intervallet  $[u.d, u.f]$  er fullstendig inneholdt i  $[v.d, v.f]$ , og  $u$  er en etterkommer av  $v$  i et dybde-først-tre
- Intervallet  $[v.d, v.f]$  er fullstendig inneholdt i  $[u.d, u.f]$ , og  $v$  er en etterkommer av  $u$  i et dybde-først-tre

## Trær

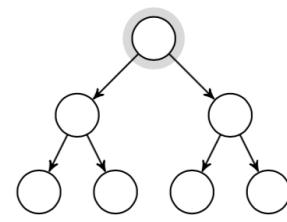
Et tre er en begrenset gruppe type graf. Trær har retning (parent / child forhold) og inneholder ikke sykler. Trær kan også defineres som grafer der, for hvert par noder  $u, v \in V$ , så eksisterer det én (og bare en) enkel sti mellom  $u$  og  $v$ .

Hvis hver node i et tre også har kun en forelder så har vi en DAG (Directed Acyclic Graph).

Fritt tre



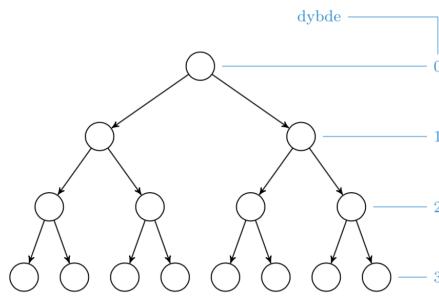
Rotfast tre



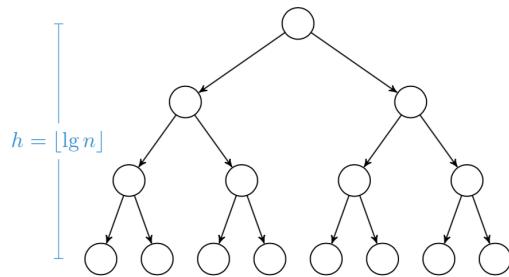
En sti kobler hvert par; en kant unna usammenhengende eller syklisk; sammenhengende eller asyklist med  $|E| = |V| - 1$

Ser gjerne på rotfaste trær som «rettede» grafer, med rettede stier vekk fra rota

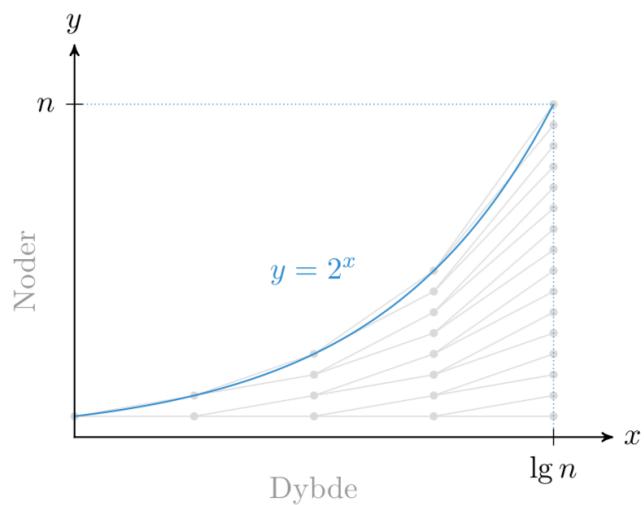
Dybde: Antall kanter unna rota



Høyde: maksimal dybde



Sammenheng mellom dybde og høyde:  $n = 2^x$



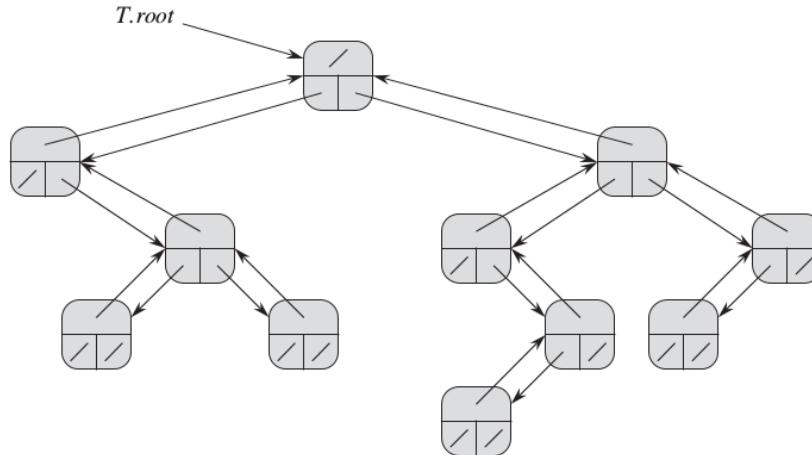
## Binærtrær

For binære trær, har hver node en nøkkel, left, right og parent. Et tre T har en peker T.root som peker på rota. Hver node peker på sine barn og sin forelder. Hvis forelder-pekeren er NIL, er dette rota. Hvis barne-pekeren er null, har ikke noden noe barn.

Fullt binærtre: Alle interne har to barn

Balansert binærtre: Alle løvnoder har ca. samme dybde

Komplett binærtre: Alle løvnoder har nøyaktig samme dybde

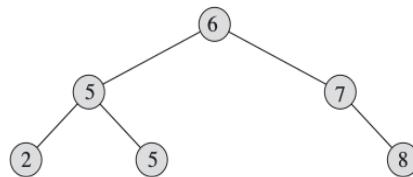


**Figure 10.9** The representation of a binary tree  $T$ . Each node  $x$  has the attributes  $x.p$  (top),  $x.left$  (lower left), and  $x.right$  (lower right). The  $key$  attributes are not shown.

## Binært søketre

Binært tre, der hver node er et objekt med attributtene `left`, `right`, `parent` og `key`. Alle binære søkertrær må tilfredsstille den binære søkeretre-egenskapen:

**Binært-søketre-egenskapen:** La  $x$  være en gitt node i søkerreetet. Hvis  $y$  er en node i det venstre subtree til  $x$  ma  $y$  sin verdi være mindre eller lik ( $\leq$ )  $x$  sin verdi. Tilsvarende for høyre subtree. Her ma  $y$  sin verdi være større enn eller lik ( $\geq$ )  $x$  sin verdi.



## Operasjoner på søkertrær

Operasjon	Average	Worst
Plass (bit)	$O(\log n)$	$O(n)$
Søk	$O(\log n)$	$O(n)$
Sett inn	$O(\log n)$	$O(n)$
Slett	$O(\log n)$	$O(n)$

## Inorder-Tree-Walk

```
INORDER-TREE-WALK( $x$ )
1  if  $x \neq \text{NIL}$ 
2    INORDER-TREE-WALK( $x.\text{left}$ )
3    print  $x.\text{key}$ 
4    INORDER-TREE-WALK( $x.\text{right}$ )
```

Heter inorder fordi den printer først venstre-barna, så seg selv, så høyrebarna. Algoritmen printer verdiene i stigende rekkefølge, ved at den begynner til venstre nederst, og jobber seg oppover nodene.

## Tree-Search

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

Tar inn en rot-node og en nøkkel den skal lete etter. Hvis den er funnet eller vi er kommet til bunnen, returneres henholdsvis noden eller NIL. Ellers, hvis nøkkelen vi leter etter er mindre enn nøkkelen til noden vi står i, går vi inn i venstre subtre. Hvis ikke går vi inn i høyre subtre. Kjøretiden blir  $O(h)$ , der  $h$  er høyden av treet, siden den i verste tilfelle traverserer en enkel sti fra rotnoden til dypeste løvnode.

## Iterative-Tree-Search

```
ITERATIVE-TREE-SEARCH( $x, k$ )
1  while  $x \neq \text{NIL}$  and  $k \neq x.\text{key}$ 
2    if  $k < x.\text{key}$ 
3       $x = x.\text{left}$ 
4    else  $x = x.\text{right}$ 
5  return  $x$ 
```

Samme som Tree-Search, bare iterativt. Er på de fleste datamaskiner mer effektivt siden vi ikke bruker tid/minne på funksjonskall.

## Tree-Minimum

```

TREE-MINIMUM( $x$ )
1 while  $x.left \neq NIL$ 
2      $x = x.left$ 
3 return  $x$ 

```

Gå til venstre helt til man når bunnen, dette er det minste elementet. Tar  $O(h)$  tid. Binære søketreegenskapen garanterer at dette gir rett svar.

## Tree-Maximum

```

TREE-MAXIMUM( $x$ )
1 while  $x.right \neq NIL$ 
2      $x = x.right$ 
3 return  $x$ 

```

Symmetrisk som minimum

## Tree-successor

```

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

```

En successor til  $x$  er den noden som kommer etter  $x$  i en inorder tree walk. Hvis noden har et høyrebarn er successor minimum av dette høyrebarnet. Hvis noden ikke har noe høyrebarn derimot, da er successoren den foreldrenoden vi støter på hvis vi går oppover i treet som ikke er høyrebarnet til sin forelder.

Kjøretiden er  $O(h)$ , siden vi enten går en enkel sti oppover i treet, eller en enkel sti nedover i treet.

## Tree-predecessor

Symmetrisk til successor, bare at vi går andre veien  
 TREE-PREDECESSOR

1. if  $x.left \neq NIL$
2.     Return TREE-MAXIMUM( $x.left$ )
3.      $y = x.p$
4.     While  $y \neq NIL$  and  $x == y.left$
5.          $x = y$
6.          $y = y.p$

7. Return y

## Tree-insert

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

Går nedover i treet for å finne den løvnoden som z skal være barnet til. Sjekker da om nøkkelen er større enn eller mindre enn noden, og avhengig av dette går vi inn i høyre eller venstre subtre helt til vi finner NIL. Vi setter da at forelderen til z er denne noden, og så må vi sjekke om nøkkelen er større enn eller mindre enn foreldrenoden sin nøkkel, og sette z som barnet til foreldrenoden, på riktig plass.

Kjøretiden er  $O(h)$ , fordi vi går en enkel sti nedover i treet.

## Transplant

Subroutine for Tree-Delete.

### TRANSPLANT( $T, u, v$ )

```
1  if  $u.p == \text{NIL}$ 
2     $T.root = v$ 
3  elseif  $u == u.p.left$ 
4     $u.p.left = v$ 
5  else  $u.p.right = v$ 
6  if  $v \neq \text{NIL}$ 
7     $v.p = u.p$ 
```

Bytter ut  $u$  med  $v$ . Hvis  $u$  er et venstrebarne, settes  $v$  til å være venstrebarne til foreldrenoden til  $u$ , samme med høyrebarn. Hvis  $v$  ikke er NIL, så settes foreldrenoden til  $v$  til å være foreldrenoden til  $u$ . Høyre- og venstrebarna til  $v$  settes ikke her, det må kalleren til Transplant gjøre. Kjøretid er  $O(1)$ , fordi alt er konstant-tid operasjoner.

## Tree-delete

Fire caser for sletting

1. Hvis venstrebarnet til  $z$  er null, flytter vi bare opp høyrebarnet til å ta over  $z$  sin plass.
2. Hvis høyrebarnet til  $z$  er null, flytter vi bare opp venstrebarnet til å ta over  $z$  sin plass.
3. Hvis  $z$  har både et venstrebarne og høyrebarn, finner vi suksessoren til  $z$ , som er minimum i det høyre subtreeet til  $z$ .

a. Hvis suksessoren til  $z$ ,  $y$ , ikke er et direkte barn av  $z$ , må vi først sette  $y$  sitt høyrebarn til å ta over plassen til  $y$  ( $y$  har ikke noe venstrebarne siden det er minimum). Så må høyrebarnet til  $y$  settes til å være lik høyrebarnet til  $z$ , og foreldren til høyrebarnet til  $y$  må settes til å være  $y$  (og ikke  $z$ ).

Uansett om  $y$  er et direkte barn av  $z$  eller ikke, settes så  $y$  inn for  $z$ , og  $y$  sitt venstrebarne blir satt til å være  $z$  sitt venstrebarne, og dette venstrebarne sin forelder blir satt til å være  $y$ .

```

TREE-DELETE( $T, z$ )
1  if  $z.left == \text{NIL}$ 
2      TRANSPLANT( $T, z, z.right$ )
3  elseif  $z.right == \text{NIL}$ 
4      TRANSPLANT( $T, z, z.left$ )
5  else  $y = \text{TREE-MINIMUM}(z.right)$ 
6      if  $y.p \neq z$ 
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      $y.left.p = y$ 

```

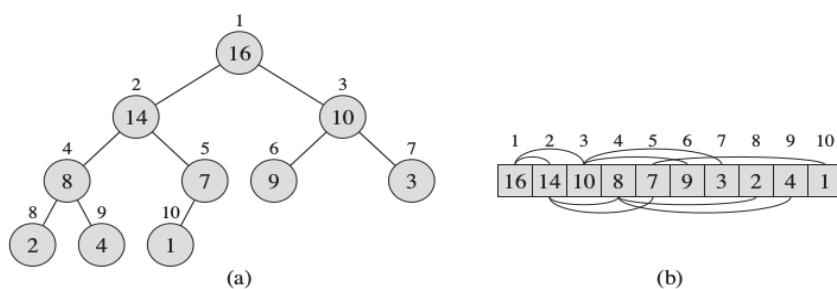
Alle linjer bortsett fra Tree-Minimum tar  $O(1)$  tid, og følgelig blir kjøretiden  $O(h)$ .

## Heaps

En heap er en spesiell trestruktur, som tilfredstiller heap-egenskapen. En Max-heap er et komplett binærtre som har Max-heap-egenskapen, nemlig at alle barn har mindre verdi. En Min-heap er tilsvarende. Denne egenskapen bruker en når en tar ut det største elementet fra heapen. En vet at det øverste elementet er størst, og for så å sette det nest-øverste elementet på toppen av heapen må en gjøre  $\log(n)$  sammenligninger. På grunn av denne egenskapen brukes en heap i prioritetskøer og heapsort.

**Heap-egenskapen:** I en heap må hele treeet være fullstendig fylt ut, bortsett fra muligens det laveste nivaet, som fylles ut fra venstre.

- Rotnode:  $i = 1$
- Foreldrenode( $i$ ):  $i/2$
- Høyre-barn:  $(2i + 1)$ , Venstre-barn:  $(2i)$



**Figure 6.1** A max-heap viewed as (a) a binary tree and (b) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

Metodene for heap inkluderer:

Metode	Kjøretid
Build-max-heap	$O(n)$
Extract-max	$O(\log n)$
Max-heapify	$O(\log n)$
Max-heap-insert	$O(\log n)$
Heap-increase-key	$O(\log n)$
Heap-maximum	$\Theta(1)$

### Vedlikehold av heaps

```

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

```

### Bygg og fiks heap

```

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

```

**Returner max/min** (avh av type heap):  
øverste element

**Parent**

```

HEAP-MAX(A)
1  return A[1]

```

### Hente ut max/min

### HEAP-EXTRACT-MAX(A)

```

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

```

### Øke verdi til en nøkkel i heapen

```

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

```

### Sette inn en verdi i heapen

```

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

```

**Left**

```

PARENT(i)
1  return ⌈i/2⌉

```

<b>LEFT(<math>i</math>)</b>	<b>Right</b>
1 <b>return</b> $2i$	<b>RIGHT(<math>i</math>)</b>

1   **return**  $2i + 1$

## Max-heapify

Finner det største elementet av left, right og i, dersom i er størst er heap-propertien opprettholdt og vi er ferdig. Hvis ikke bytter vi i med det største av barna, og kjører max-heapify på dette subtreet.

## Build-max-heap

Vi kan bruke max-heapify for å bygge en max-heap. Hvis vi ser på representasjonen av en heap i et array, ser vi at alle elementene fra index  $n//2 + 1$  til  $n$  er løv, altså har de ingen barn og kan ikke sammenliknes med noen, men de vil bli sammenliknet med foreldrene sine når de blir plassert riktig. Vi kan derfor bare kjøre max-heapify på de  $n//2$  første elementene i lista, så er heapen bygget.

## Heap-extract-max

Returnerer elementet med høyest prioritet, og fjerner det fra heapen. Setter bakerste element i heapen til å være ny rot-node, siden vi skal extracte denne, og deretter dekrementeres heap-size, så det siste ikke lenger er med i heapen. Kjører max-heapify på den nye rota, og returnerer så den gamle. Alle operasjoner bortsett fra max-heapify tar  $\Theta(1)$  tid, så kjøretiden blir  $O(\lg n)$ .

## Heap-increase-key

Tar inn en indeks og en ny nøkkelen, som må være større enn den gamle nøkkelen. Setter den nye nøkkelen, og så må den oppdatere heapen. Det er nå en mulighet for at nøkkelen er større enn sin forelder. Må gå oppover fra noden mot rota og sammenlikne node i med foreldrenoden til i. Dersom foreldrenoden er mindre enn node i, må de to byttes om. Fortsetter helt til node i ikke er større enn foreldrenoden sin, eller til vi når rota.

## Max-heap-insert

Setter inn et nytt objekt i heapen. Setter den først bakerst med en nøkkel på minus uendelig. Kaller så på heap-increase-key med den faktiske nøkkelen.

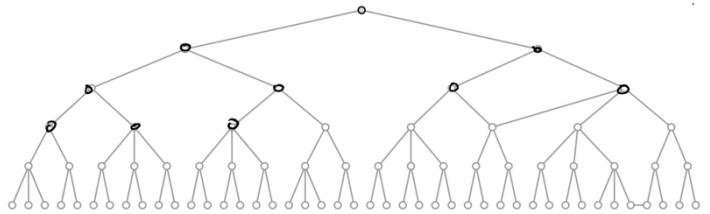
OBS! Alt med MAX skal også kunnes med MIN. Blir basically bare å bytte ut alle max-ord med min.

# Søk i Graf / Traversering

Vi ønsker en veldefinert måte å gå gjennom alle nodene i en graf, nemlig å traversere en graf. De to vanligste måtene å gjøre dette på heter dybde først søk og bredde først søk.

For både Bredde-Først søker (BFS) og Dybde-Først søker (DFS) kan vi terminere søkeret når vi finner elementet vi leter etter.

## Bredde først (BFS)



Bredde først søker er en FIFO graftraverseringsalgoritme/søkealgoritme. For hver node en besøker legger en alle nodens barn i en kø. Man starter på foreldrenoden og legger inn alle dens barn i køen. Nar alle naboer til node x er oppdaget, fjernes den fra køen og man tar den neste noden i køen, og legger alle dens barn inn i køen. Nar køen er tom, sjekker man ikke videre om det er ubesøkte noder.

Metode: FIFO-kø

Mer konkret:

1. Legg til startnode i køen vår, Q.
2. Hent ny aktiv node gjennom et POP-kall til køen.
3. Legg den aktive nodens barn til i Q, så fremt de ikke allerede er besøkt.
4. Gå til steg 2 og gjenta til Q er tom.

Kjøretid:  $O(|V| + |E|)$

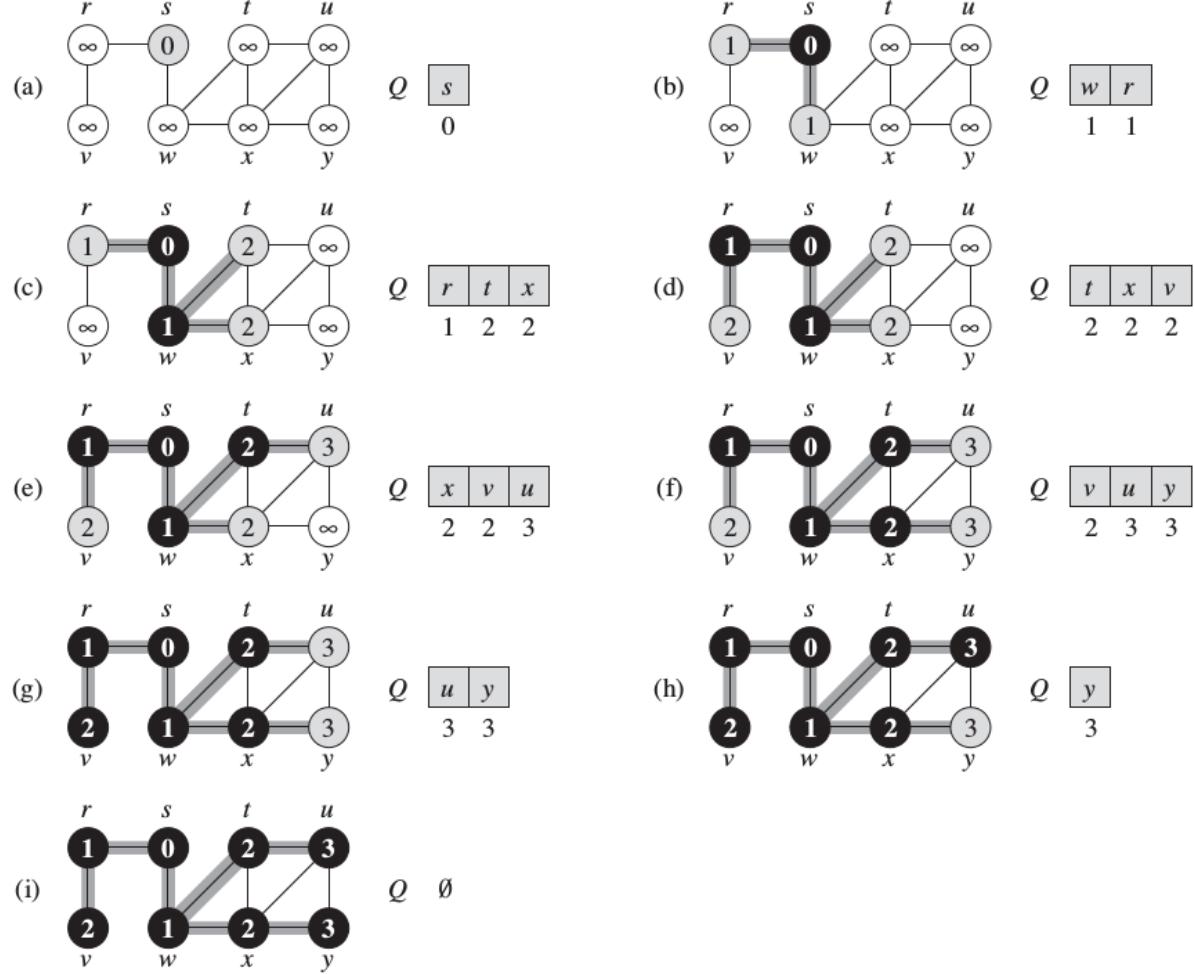
Pseudokode:

**BFS( $G, s$ )**

```
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
```

## Forklaring

Setter alle fargene til hvit, bortsett fra rotnoden s. Legger s inn i køen. Sålenge det fortsatt er grå noder igjen, velger vi den første grå noden fra Q, går gjennom alle kantene som går fra denne, hvis den er oppdaget, farger vi den grå, setter avstanden fra s til avstanden fra s til v pluss en, og setter foreldrevariablene til v. Så legger vi denne inn i Q. Til slutt farger vi noden svart, for nå er den utforsket.



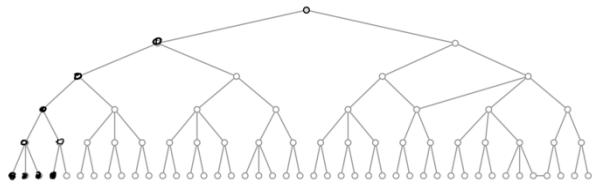
**Figure 22.3** The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. The value of  $u.d$  appears within each vertex  $u$ . The queue  $Q$  is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear below vertices in the queue.

Alle noder blir lagt inn i Q én gang, så kø-operasjonene tar  $O(V)$  tid. Vi går gjennom hver node én gang, og da også hver node sin nabolist, siden summen av lengdene av nabolistene er antall kanter, tar dette til sammen  $O(E)$  tid, så til sammen tar prosedyren  $O(V+E)$  tid.

## Korteste vei uten vekter

BFS finner korteste vei fra rotnoden  $s$  til alle andre noder. Avstanden lagret i hver node etter å ha kjørt BFS er lik antall kanter fra startnoden. Dette må være korteste sti, fordi vi går bredde først, så når vi kommer til noden første gang, og setter d-variabelen, har vi traversert  $k$  kanter fra startnoden, og alle senere ganger vi treffer noden vil vi ha traversert  $k + x$  kanter,  $x > 0$ .

## Dybde først (DFS)



Dybde først søker er en LIFO graftraverseringsalgoritme/søkealgoritme. For hver node en besøker legger en nodens barn i en stack. Kan man ikke gå videre vil den “backtrace” til forrige node, og se etter en mulig vei videre. Hvis stacken er tom, sjekker man om alle noder er besøkt. Hvis ikke, starter man på nytt fra ubesøkte noder.

Metode: LIFO-kø(stack)/rekursjon

Mer konkret:

1. Legg til startnoden i stakken, S.
2. Hent ny aktiv node gjennom POP-kall til stakken.
3. Legg den aktive nodens barn til i S, så fremt de ikke allerede er besøkt.
4. Gå til steg 2 og gjenta til S er tom.

Kjøretid:  $O(|V| + |E|)$

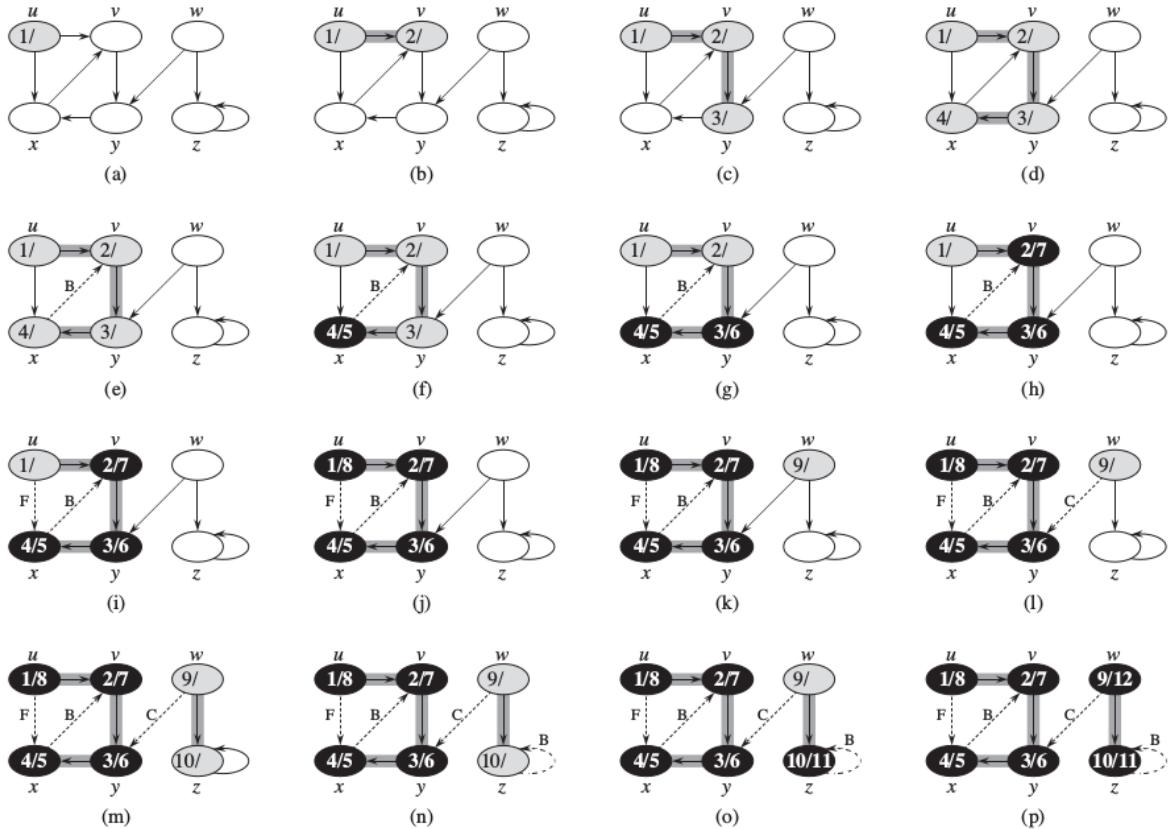
### $\text{DFS}(G)$

```
1  for each vertex  $u \in G.V$ 
2       $u.\text{color} = \text{WHITE}$ 
3       $u.\pi = \text{NIL}$ 
4       $time = 0$ 
5  for each vertex  $u \in G.V$ 
6      if  $u.\text{color} == \text{WHITE}$ 
7           $\text{DFS-VISIT}(G, u)$ 
```

### $\text{DFS-VISIT}(G, u)$

```
1   $time = time + 1$            // white vertex  $u$  has just been discovered
2   $u.d = time$ 
3   $u.\text{color} = \text{GRAY}$ 
4  for each  $v \in G.\text{Adj}[u]$     // explore edge  $(u, v)$ 
5      if  $v.\text{color} == \text{WHITE}$ 
6           $v.\pi = u$ 
7           $\text{DFS-VISIT}(G, v)$ 
8   $u.\text{color} = \text{BLACK}$         // blacken  $u$ ; it is finished
9   $time = time + 1$ 
10  $u.f = time$ 
```

**Pseudokode:** Vi går gjennom alle nodene, og hvis den er hvit kaller vi på DFS-visit. Her er det gitt at noden er hvit. Vi farger den grå og setter discovery-variabelen. Så går vi gjennom alle kantene ut fra denne, setter noden som foreldrenoden deres, og tar DFS-visit på de igjen. Til slutt settes fargen til svart når vi er ferdig med å utforske noden, øker tidsvariabelen, og setter finish-variabelen til tiden.



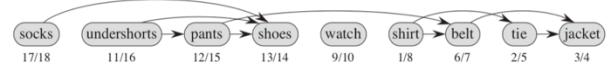
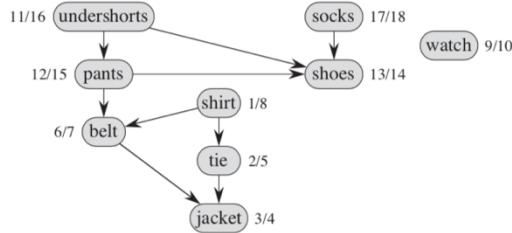
**Figure 22.4** The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.

Vi går gjennom alle nodene én gang, så kjøretiden, eksklusivt kallet til DFS-visitt er  $\Theta(V)$ . For hvert kall til DFS-visitt besøkes hver node i nabolista til noden, og summen alle disse er som kjent  $\Theta(E)$ . Derfor blir kjøretiden  $\Theta(V+E)$ .

## Topologisk sortering

Sortering av en graf, slik at alle kanter går til høyre. Hvis grafen  $G$  inneholder en kant  $(u,v)$ , så kommer  $u$  foran  $v$  i rekkefølgen. Dvs arbeidet med å sortere nodene i en graf slik at naboer listes i rett innebyrdes orden (foreldre før barn). Det forutsettes at to noder bare har en rettet kant seg imellom, og at grafen er asyklistisk. Krav om at kun ta en topologisk sortering når vi har en DAG (directed acyclic graph).

En topologisk sortert DAG  $G = (V, E)$  er en lineær sortering av alle sine node slik at hvis  $G$  inneholder en kant  $(u, v)$ , så vil  $u$  stå før  $v$  i sorteringen.



Ved å merke start- og slutt-tider kan DFS brukes til topologisk sortering, men dette krever da at grafen er en DAG (Directed Acyclic Graph). Man begynner i noden som ikke har noen kanter inn til seg. Finnes det en kant  $(u,v)$ , skal noden  $u$  komme før  $v$  i ordningen. DFS brukes til å finne denne ordningen.

Hvis det er mulig å lage en topologisk sortering (grafen er rettet og asyklistisk), kan en kjøre **DAG-shortest-path**, den mest effektive løsningen av korteste vei en til alle.

### Pseudokode:

#### TOPOLOGICAL-SORT( $G$ )

- 1 call  $\text{DFS}(G)$  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

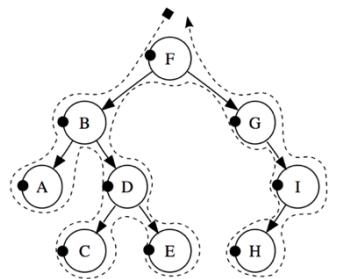
Kjøretiden er det samme som DFS, siden vi bare setter inn en operasjon som tar  $O(1)$  tid inn i DFS. Kjøretiden blir altså  $\Theta(V+E)$ .

## Traverseringstrær

Et traverseringstre er et tre der vi har traversert hver node i grafen. Vi har to ulike traverseringstrær, BFS og DFS. BFS = bredde først søk, som går breddevis utover og finner noder. DFS = Dybde først søk, som går dybdevis nedover, opp og ned. Selv om treet er nodene i grafen koblet sammen av kantene vi finner når vi først oppdager noden. Hver node har en forgjenger, og det er kantene mellom noder og deres forgjengere som blir kantene i treet. For DFS får vi ikke et tre, men en skog.

### Traversering med vilkårlig prioritetskø

Algoritmene er egentlig veldig like, det handler bare om i hvilken prioritet vi tar ut elementer fra prioritetskøen. BFS prioriterer de eldste, DFS prioriterer de nyeste. Dijkstra prioriterer de med kortest avstandsestimat, og Prim prioriterer de som har en lett kant til en av de utforskede (sorte) nodene.

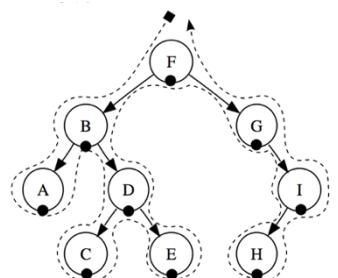


**In-Order, Pre-Order og Post-Order traversering** En grei måte å huske det på, er å tenke når vi legger nodene inn i lista over besøkte noder. Vi starter alltid i toppnoden og går nedover venstre side av grafen sånn som i visualiseringa under.

I **pre-order** rekkefølge legger vi noden til lista når vi passerer dem på venstre side. Her printer man ut nodens verdi før dens barn, venstre og deretter høyre.

rot, venstre, høyre

|||||||

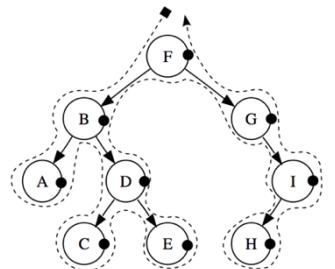


I **in-order** rekkefølge legger vi noden til lista når vi er rett på undersida av noden. Her

printer man venstre barn, noden, og deretter høyre barn (om ikke det er noe venstre barn, print noden før høyre barn).

venstre, rot, høyre

|||||||



I **post-order** rekkefølge legger vi noden til lista når vi passerer dem på høyre side. Her printer man nodens verdi etter man har printet venstre og høyre barn.

venstre, høyre, rot

# Grafalgoritmer

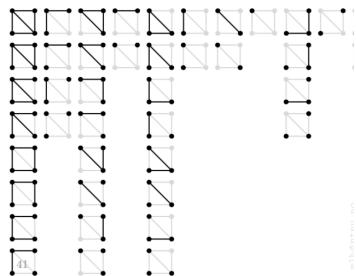
## Minimale spennrør

Et minimalt spennrør er et tre som er innom alle nodene nøyaktig én gang, og som har den lavest mulige kombinerte kantvekten. Merk: Hvis alle kantene i en sammenhengende graf har forskjellige vekter, vil grafen kun ha ett minimalt spennrør.

**Graf:**



**Delgraf:** Alle kombinasjoner av grafen



41

**Spenngraf:** Er en delgraf med samme nodesett som orginalgrafen

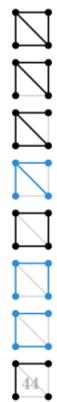


42

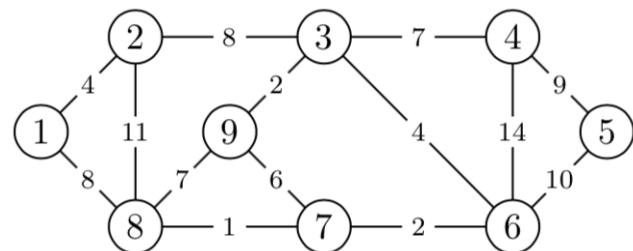
**Spennskog:** Er en asyklistisk spenngraf



**Spenntre:** Er en sammenhengende spennskog



Innfører **vekter** på kantene: omtales som lengder eller kostnader



Def:

**Invariant:** Kantmengden utgjør en del av et minimalt spenntre

«**Trygg kant**»: En kant som bevarer invarianten

### Generic-MST

**Input:** En urettet graf  $G = (V, E)$  med en vektfunksjon  $w : E \rightarrow \mathbb{R}$

**Output:** En asyklig delmengde  $T$  bestående av kanter fra  $E$  som kobler sammen alle nodene i  $V$  og minimerer vektsummen.

```

GENERIC-MST( $G, w$ )
1    $A = \emptyset$ 
2   while  $A$  does not form a spanning tree
3       find an edge  $(u, v)$  that is safe for  $A$ 
4        $A = A \cup \{(u, v)\}$ 
5   return  $A$ 

```

En trygg kant betyr en kant vi kan legge til i  $A$  som opprettholder invarianten (se under).

Merk: En kant er en **lett kant** som krysser snittet hvis vekten til kanten er minimal av kantene som krysser snittet. Lette kanter = trygge kanter

## Skog -> disjunkte menger

En disjunkt datastruktur inneholder en samling  $\{ s_1, s_2, \dots, s_k \}$  av disjunkte dynamiske sett. Hvert sett har en representativ, som er et medlem av settet. Det er viktig at det samme elementet blir returnert hver gang man ber et sett om sin representativ.

En rotfast trestruktur har to “triks” for å bli en optimal implementasjon av disjunkte sett:

- 1) Union etter rank: Hver node har en variabel, rank, som er en øvre grense på høyden til noden i treet. Når vi tar union av to trær, blir den rota med høyest rank den nye rotnoden, mens den rota med lavere rank peker direkte på den nye rota. (Hvis de to trærne har lik rank, velges tilfeldig en av rotnodene som ny rot, og ranken må inkrementeres).
- 2) Stikompresjon: Under Find-set-operasjoner får vi hver node til å peke direkte til rota, slik komprimerer vi treet til å ha så lav høyde som mulig, som gjør det raskere å finne rota. Den endrer ikke ranken til nodene.

## Make-set

```

MAKE-SET( $x$ )
1    $x.p = x$ 
2    $x.rank = 0$ 

```

Oppretter et nytt sett med én node, foreldrenoden settes til seg selv og ranken er 0.

## Union

```
UNION( $x, y$ )
1  LINK(FIND-SET( $x$ ), FIND-SET( $y$ ))
```

Finner de to representativelementene og kaller på hjelpe-metoden Link.

## Find-set

```
FIND-SET( $x$ )
1  if  $x \neq x.p$ 
2     $x.p = \text{FIND-SET}(x.p)$ 
3  return  $x.p$ 
```

Kaller rekursivt på seg selv med foreldrenoden som parameter helt til den finner rota. Rekurerer da tilbake og setter foreldrenoden til alle nodene til å peke på rot-noden (stikompresjon).

## Kruskal

Kruskals algoritme lager treet ved å finne de minste kantene i grafen en etter en, og lage en skog av trær. Deretter settes disse trærne gradvis sammen til ett tre, som blir det minimale spennetreet. Først finnes kanten i grafen med lavest vekt. Denne kanten legges til et tre. Deretter ser algoritmen etter den neste laveste kantvekten. Er ingen av nodene til denne kanten med i noe tre, så lages et nytt tre. Er én av nodene knyttet til et tre, så legges kanten til i det eksisterende treet. Er begge nodene knyttet til hver sitt tre settes de to trærne sammen. Er begge nodene knyttet til samme tre ignoreres kanten. Sånn fortsetter det til vi har ett tre. Kruskal godtar ikke sykliske koblinger. Merk: Kruskal er en grådig algoritme. Hvis man ikke sorterer kantene vil man ende opp med et spennetre.

### Pseudokode:

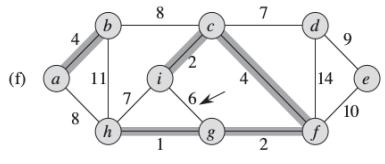
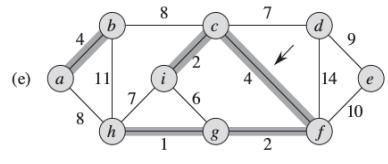
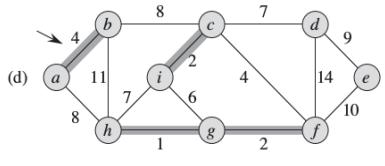
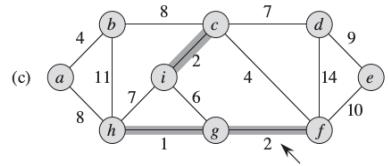
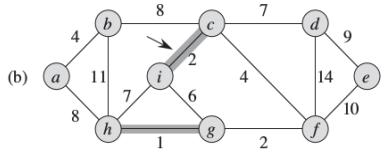
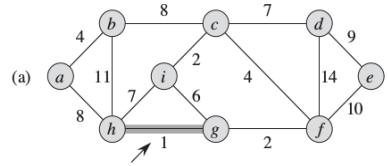
<pre> MST-KRUSKAL(G, w) 1 A = ∅ 2 for each vertex <math>v \in G.V</math> 3   MAKE-SET(<math>v</math>) 4 sort G.E by <math>w</math> 5 for each edge <math>(u, v) \in G.E</math> 6   if FIND-SET(<math>u</math>) ≠ FIND-SET(<math>v</math>) 7     A = A ∪ {(<math>u, v</math>)} 8     UNION(<math>u, v</math>) 9 return A </pre>	<pre> MAKE-SET(<math>x</math>) 1 <math>x.p = x</math> 2 <math>x.rank = 0</math>  FIND-SET(<math>x</math>) 1 if <math>x \neq x.p</math> 2   <math>x.p = \text{FIND-SET}(x.p)</math> 3 return <math>x.p</math> </pre>
--	---

---

Best Case	Average Case	Worst Case
		$O(E \log V)$

---

Først lager vi oss  $|V|$  sett, og sorterer de etter vekt. Så går vi gjennom settene, etter hvem som har lavest vekt. Hvis u har et annet representativelement en v, dvs. u og v er i to ulike disjunkte sett, så slår vi dem sammen, og legger til kanten i A. Vi binder sammen u og v, slik at de er i samme tre, og Find-Set vil returnere samme representativelement, så de ikke blir koblet sammen på noen annen måte senere, og dermed danner en sykel.



## Prim

Prims algoritme lager treet ved å starte i en vilkårlig node, og så legge til den kanten knyttet til noden som har lavest verdi. Deretter velges kanten med lavest verdi som er i knyttet til én av nodene som nå er en del av treet. Dette fortsetter til alle nodene er blitt en del av treet. Prim bruker en min-prioritet-kø med noder der prioriteten er vekten på den letteste kanten mellom noden og treet. Kjøretiden avhenger av datastrukturen som velges, pensum bruker en binærheap. Merk: Prim er en grådig algoritme.

## Pseudokode

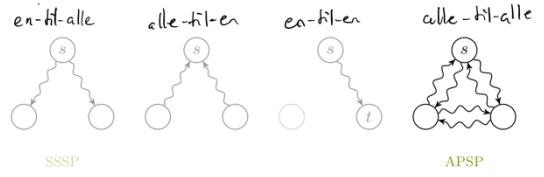
:

```
MST-PRIM(G, w, r)
1  for each  $u \in G.V$ 
2     $u.key = \infty$ 
3     $u.\pi = \text{NIL}$ 
4   $r.key = 0$ 
5   $Q = G.V$ 
6  while  $Q \neq \emptyset$ 
7     $u = \text{EXTRACT-MIN}(Q)$ 
8    for each  $v \in G.Adj[u]$ 
9      if  $v \in Q$  and  $w(u, v) < v.key$ 
10          $v.\pi = u$ 
11          $v.key = w(u, v)$ 
```

Best Case	Average Case	Worst Case
		$O(E\log V)$

Setter først alle key-attributtene til uendelig, og foreldre-noder til NIL. Startnoden r sin nøkkel er 0 (null vekt å gå til seg selv), og G er lik alle nodene V i G. Sålenge Q er ikke-tom, tar vi ut den noden som har minst key (første gang blir dette r, siden alle andre har key = uendelig). Vi går så gjennom alle kantene til noden, og hvis denne noden er i Q (har ikke blitt utforsket enda og er dermed en del av spennetreet) og vekten til kanten er mindre enn den kanten som til nå har bundet noden til spennetreet (funnet en kant med lavere vekt), så oppdaterer vi noden slik at foreldrenoden blir satt og vekt-verdien blir satt.

## Korteste Vei En til Alle (SSSP)



Gitt en vektet, rettet graf med en vektfunksjon, ønsker vi en sti fra node u til node v der summen av vektene er minimert (vekten til stien er minimert). Å finne korteste vei i en graf kan modellere veldig mange situasjoner i virkeligheten. Generelt deles korteste vei-problemer inn i mindre delproblemer, der en ser på korteste delstrekninger. Altså har problemet optimal substruktur.

Visse egenskaper ved grafen vil påvirke hvilken algoritme vi bør bruke. Er grafen rettet og uten sykler? -> Vi kan bruke DAG. Inneholder grafen kun positive kanter? -> Vi kan bruke Dijkstra \* Dannes det negative sykler? Korteste"vei er udefinert og vi må bruke Bellman-Ford.

Def:

**Pulling:** Bottom-up kantslakking av inn-kanter i topologisk sortert rekkefølge

**Reaching:** Bottom-up(?) kantslakking av ut-kanter i topologisk sortert rekkefølge

**Kantslakking** er oppspalting av minimums-operasjonen. Hver gang vi finner en snarvei synker estimatet av avstanden s u.

```

RELAX( $u, v, w$ )
1  if  $v.d > u.d + w(u, v)$ 
2       $v.d = u.d + w(u, v)$ 
3       $v.\pi = u$ 
    
```

**Relax** er en hjelpe-metode for korteste-sti-algoritmene. Den tar inn node u og v (som det finnes en kant mellom,  $(u,v)$ ), og en vektfunksjon w. Målet er å slakke v-noden. Dersom avstanden fra s til v er større enn avstanden fra node s til u etterfulgt av vekten til kanten  $(u,v)$ , så er denne nye avstanden bedre, og vi bytter ut stien til v med den nye stien som går via u. Hvis dette ikke er tilfellet gjør den ingenting. Metoden tar konstant tid,  $O(1)$ .



**Sti-slakking-egenskapen:** Om p er en korteste vei fra s til v og vi slakker kantene til p i rekkefølge, så vil v få riktig avstandsestimat. Det gjelder uavhengig av om andre

slakninger forekommer, selv om de kommer innimellom. Altså, når forgjengere har rett svar så har nåværende node rett svar.

En **enkel sti** er en sti uten sykler. En **kortest sti** er alltid enkel.

Negativ sykel? Ingen sti er kortest!

Det finnes fortsatt en kortest enkel sti. (Uløst hvordan (NP-hardt))

### **Smartere slakking(?)**

Strategi 1: Slakk kanter ut fra noden i topologisk sortert rekkefølge(krever en rettet asyklig graf).

Fungerer fordi: Når alle inn-kanter er slakket kan ikke noden forbedres, og kan trygt velges som neste.

Strategi 2: Velg den gjenværende med lavest estimat (fungerer med sykler, men ikke negative kanter) Dijkstra

Fungerer fordi: Gjenværende noder kan kun forbedres ved slakking fra andre gjenværende.  
Det laveste estimatet kan dermed ikke forbedres.

### **Alle til en**

Ved å snu alle kantene i en graf kan man nne korteste alle-til-én veier, gitt at man vet hvordan man nner korteste en-til-alle veier.

## Bellman-Ford

Bellman-Ford er den tregeste algoritmen, og finner korteste vei med negative kanter, men ikke negative sykler. I tilfelle negative sykler vil den håndtere de ved å returnere false. I en graf med negative sykler er korteste vei udefinert, men BF vil da vite med sikkerhet at grafen inneholder minst en negativ sykel som kan nås fra opprinnelsesnoden (origin). Returnerer false dersom det finnes en negativ sykel i grafen. Går igjennom alle kantene og bruker RELAX på hver av dem  $i = |V| - 1$  ganger. Deretter sjekker den etter negative sykler ved å sjekke om veien fra startnoden til node  $w$  via node  $v$  blir mindre enn den vi fant under søket.

**Input:** En vekta retta graf med kilde  $s$  og en vektfunksjon  $w$ .

**Output:** Boolean som indikerer om det finnes en negativ sykel eller ikke (True for ikke sykel). Hvis True, produserer den også en korteste sti fra  $s$  til alle andre noder.

```

BELLMAN-FORD( $G, w, s$ )
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2 for  $i = 1$  to  $|G.V| - 1$ 
3   for each edge  $(u, v) \in G.E$ 
4     RELAX( $u, v, w$ )
5   for each edge  $(u, v) \in G.E$ 
6     if  $v.d > u.d + w(u, v)$ 
7       return FALSE
8 return TRUE

```

```

INITIALIZE-SINGLE-SOURCE( $G, s$ )
1 for each vertex  $v \in G.V$ 
2    $v.d = \infty$ 
3    $v.\pi = \text{NIL}$ 
4    $s.d = 0$ 

```

```

RELAX( $u, v, w$ )
1 if  $v.d > u.d + w(u, v)$ 
2    $v.d = u.d + w(u, v)$ 
3    $v.\pi = u$ 

```

Initialiserer alle nodene. Går så gjennom alle kantene  $V-1$  ganger og slakker alle (etter linjene 1-4 skal alle nodene ha sin endelige koststand). og en ytterlig iterasjon (linje 5-7) skal ikke gi utslag. Hvis det gjør det har vi en negativ sykel., returnerer i henhold til dette.

Best Case	Average Case	Worst Case
		$O( V   E )$

## Dijkstras algoritme

Grådig algoritme, velger alltid den noden med laveste avstandsestimat, slakker alle kantene ut fra denne noden. Dijkstras algoritme er raskere enn Bellman-Ford, men terminerer bare hvis kantvektene er ikke-negative. Den tillater positive sykler. Den velger noder en etter en fra hvor nærmest de er startnoden. Dijkstra er en gradig algoritme. Dijkstras er mest effektiv når det brukes en heap som prioritetskø, og det er også denne kjøretiden vi har listet.

Pseudokode:

```

DIJKSTRA( $G, w, s$ )
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S = \emptyset$ 
3  $Q = G.V$ 
4 while  $Q \neq \emptyset$ 
5    $u = \text{EXTRACT-MIN}(Q)$ 
6    $S = S \cup \{u\}$ 
7   for each vertex  $v \in G.Adj[u]$ 
8     RELAX( $u, v, w$ )

```

RELAX( $u, v, w$ )

```

1 if  $v.d > u.d + w(u, v)$ 
2    $v.d = u.d + w(u, v)$ 
3    $v.\pi = u$ 

```

INITIALIZE-SINGLE-SOURCE( $G, s$ )

```

1 for each vertex  $v \in G.V$ 
2    $v.d = \infty$ 
3    $v.\pi = \text{NIL}$ 
4    $s.d = 0$ 

```

1. Lag en min-prioritetskø  $Q$  av alle nodene i grafen. (linje 3)

2. Hent ut noden fra Q som har lavest kostnad. (linje 5)
3. Slakk (relax) nabonodene til den aktive og oppdater avstandene deres. (linje 8)
4. Legg den aktive noden til i besøkte noder. (linje 6)
5. Gå til steg 2 og gjenta fram til Q er tom.

Best Case	Average Case	Worst Case
		$O( E  +  V  \log  V )$

Kjøretiden av avhengig av hvordan vi implementerer prioritetskøen Q. Vi kaller insert én gang per node når vi initialiserer køen, extract-min én gang per node når vi tar den ut av køen, og decrease-key én gang for hver kant i køen.

Array: Lagrer v.d i indeks v i arrayet, insert og decrease-key tar  $O(1)$  tid og extract-min tar  $O(V)$  tid. Kjøretiden blir da  $O(V^2 + E) = O(V^2)$ .

Binær min-heap: Hvis E er liten, altså  $E = o(V^2/\lg V)$ . Extract-Min tar da  $O(\lg V)$  tid. Det tar  $O(V)$  tid å lage heapen. Decrease-key tar  $O(\lg V)$ . Kjøretiden blir da  $O(V \lg V + E \lg V + V) = O(E \lg V)$  hvis alle nodene kan nås fra startnoden.

Fibonacci-heap: Extract-min tar  $O(\lg V)$ , decrease-key tar  $O(1)$  tid. Dette gir da en kjøretid på  $O(V \lg V + E)$

## DAG shortest path

Den raskeste algoritmen for å løse problemet SSSP, DAG-Shortest-Path er en korteste vei, en-til-alle algoritme. For å kjøre denne må man ha en DAG (Directed Acyclic Graph). Tillater ikke negative kanter, og kan selvfølgelig ikke ha sykler, da det er en DAG. Gjør topo- logisk sortering av DAGen og besøker hver node en gang før å kjøre RELAX på nodene foran. Da kan vha **DFS** lage en topologisk sortering og slakke kantene i denne rekkefølgen.

Kobling til dynamisk programmering: Når alle innkanter er slakket, kan ikke noden få bedre avstandsestimat. Vi løser dermed alle delproblemene (kan jeg få en bedre avstand ved å gå via denne kanten) før vi kommer til noden, bottom-up iterativ dynamisk programmering.

**Input:** En vekta DAG (retta asyklist graf).

**Output:** Korteste sti fra en kilde s til alle andre noder.

**Pseudokode:**

DAG-SHORTEST-PATH( $G, w, s$ )

- 1 topologically sort the vertices of  $G$
- 2 INITIALIZE-SINGLE-SOURCE( $G, s$ )
- 3 **for** each vertex  $u$ , in topsort order
- 4     **for** each vertex  $v \in G.Adj[u]$
- 5         RELAX( $u, v, w$ )

INITIALIZE-SINGLE-SOURCE( $G, s$ )

- 1 **for** each vertex  $v \in G.V$
- 2      $v.d = \infty$
- 3      $v.\pi = \text{NIL}$
- 4      $s.d = 0$

RELAX( $u, v, w$ )

- 1 **if**  $v.d > u.d + w(u, v)$
- 2      $v.d = u.d + w(u, v)$
- 3      $v.\pi = u$

Best Case	Average Case	Worst Case
$\Theta( V  +  E )$	$\Theta( V  +  E )$	$\Theta( V  +  E )$

Sorterer grafen i topologisk rekkefølge. Går gjennom nodene i denne rekkefølgen, og slakker alle kantene ut fra denne noden.

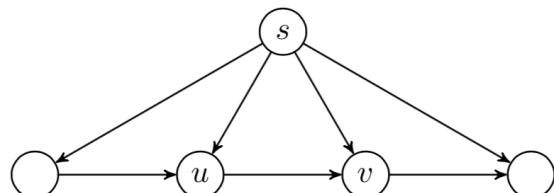
Lineær kjøretid, veldig positivt. Krever ingen sykler og retta graf, ikke så positivt.

Topologisk sortering tar  $\Theta(V+E)$  tid, initialisering av nodene tar  $\Theta(V)$  tid. Vi går gjennom alle nodene, og til sammen slakker vi kantene nøyaktig én gang hver. Relax tar  $O(1)$  tid, så kjøretiden blir  $\Theta(V+E)$ .

## Korteste Vei Alle til alle (APSP)

Dette problemet er en direkte forelengelse av problemet korteste vei én til alle, for en kan jo selvfølgelig kjøre Bellman-Ford eller Dijkstra for hver node. Da får en hhv. kjøretiden  $|V|O(|E||V|) = O(|E||V|^2)$  og  $|V|O(|E| + |V|\log|V|) = O(|V||E| + |V|^2\log|V|)$ . Altså vil vi i en dense graf med negative kanter og mange kanter få en kjøretid på  $O(V^4)$ , fordi  $|E| = |V^2|$ . Floyd-Warshall reduserer dette til  $O(V^3)$ . Merk at i en graf med negative sykler er korteste vei ikke definert og vi kan heller ikke bruke Floyd-Warshall.

## Johnson



Johnson finner korteste vei i en alle-til-alle graf. Den kombinerer Dijkstra og Bellman-Ford så man kan ha negative kanter i grafen. Bruker Dijkstra for hver node dersom spinkle/enkle grafer. Hvis den støter på noen negative kanter brukes Bellman-Ford først. Dette kan gjennomføres ved å legge til noden s midlertidig og kan nå sikre  $w(u,v) + h(u)$  ( $s, v$ ), der  $h(u) = h(s) + w(s,u)$  og  $h(v) = h(s) + w(s,v)$ .

JOHNSON( $G, w$ )

```

1 construct G' with start node s
2 BELLMAN-FORD(G', w, s)
3 for each vertex v ∈ G.V
4      $h(v) = v.d$ 
5 for each edge  $(u, v) \in G.E$ 
6      $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$ 
7 let D =  $(d_{uv})$  be a new  $n \times n$  matrix
8 for each vertex u ∈ G.V
9     DIJKSTRA(G,  $\hat{w}$ , u)
10    for each vertex v ∈ G.V
11         $d_{uv} = v.d + h(v) - h(u)$ 
12 return D

```

```

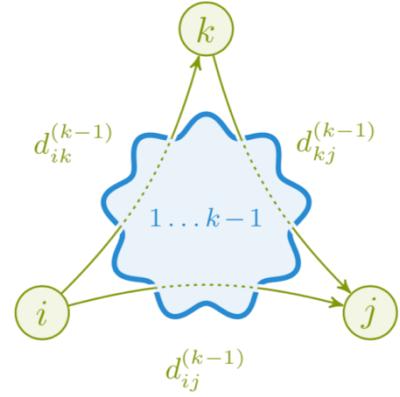
1    JOHNSON(G, w)
2    compute  $G^*$ , where  $G^*.V = G.V \cup \{s\}$ 
3     $G^*.E = G.E \cup \{(s, v) : v \in G.V\}$ , and  $w(s, v) = 0$  for all  $v \in G.V$ 
4    if BELLMAN_FORD( $G^*$ , w, s) == false
5        print "graph with negative-weight cycle"
6    else
7        for each vertex  $v \in G^*.V$ 
8            set  $h(v)$  to the value of  $\delta(s, v)$ 
9                computed by the Bellman-Ford algorithm
10       for each edge  $(u, v) \in G^*.E$ 
11            $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$ 
12       let D = ( $d_{uv}$  be a new  $n \times n$  matrix
13       for each vertex  $u \in G.V$ 
14           run DIJKSTRA( $G$ ,  $\hat{w}$ , u) to compute  $\hat{\delta}(u, v)$  for all  $v \in G.V$ 
15           for each vertex  $v \in G.V$ 
16                $d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$ 
17   return D

```

Forelesning - forenkling: Ignorerer returverdi fra Bellman-Ford = Antar vi

## Pseudokode:

## Floyd-Warshall



Funker hvis det finnes negative kanter, men ingen negative sykler. Nodene må være lagret som en nabomatrise, ikke en nabolist. Den bruker DP. Lager en nabomatrise for alle noder og hvor det gar kanter. Kostnaden mellom disse blir verdien av kanten. Er det ikke en direkte vei mellom to noder settes verdien til  $\infty$ . Deretter velges den en node a og sjekker om veien fra u til v er kortere via a. Deretter finner den en ny node, og sjekker om det er kortere veier om man benytter seg av denne. Slik fortsetter den til den har besøkt alle noder. Målsetting med FW: tillater negative kanter, har lavere asymptotisk kjøretid, lavere konstantledd

Problemet som skal løses for hver kant  $(i, j)$  er altså:

$$d_{ij}^{(k)} = \min \left( d_{ij}^{(k-1)}, \left( d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) \right)$$

Forskjellen fra transitiv lukking er at vi ikke bare ser om det er en sti, men ser også på vekten/avstanden d mellom kantene.

**Input:** Graf med en vektfunksjon w

**Output:** To matriser: en for korteste vekt mellom to noder, og en for forgjengeren til noden på den korteste stien.

**Psuedokode fra forelesning:**

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D(0) = W
3  for k = 1 to n
4      let D(k) = (d(k)ij) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d(k)ij = min (d(k-1)ij, d(k-1)ik + d(k-1)kj)
8  return D(n)
```

**Forenklet versjon uten D<sup>k</sup>**

```
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 dij > dik + dkj
7                  dij = dik + dkj
8                  πij = πkj
9  return D, Π
```

For hver tildeling av nodene i, j og k sjekker den om det finnes en raskere vei fra i til j som

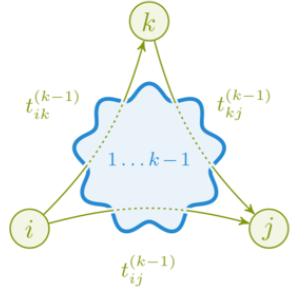
går gjennom k.

Best Case	Average Case	Worst Case
$\Theta( V ^3)$	$\Theta( V ^3)$	$\Theta( V ^3)$

Algoritmen lager matriser  $D^{(k)}$  der  $D^{(k)}[i][j]$  gir korteste vei fra i til j som kun passerer noder nummerert k eller lavere.

$D^{(0)}$  blir bare vektene, og forgjengeren blir i for stien (i,j). Regner ut formelen for  $d_{ij}$  som forklart over, for alle noder 1 til k, for hver startnode, for hver sluttnode, sjekk om stien er kortere om vi går innom node k, altså sjekk om stien vi har fra i til j er lengre enn stien fra i til k pluss stien fra k til j. Floyd-Warshall' dropper k-en, vi trenger bare en matrise, som blir skrevet over. Den tar også med forgjengeren, slik at vi kan finne tilbake til den korteste stien.

## Transitive-Closure



Gjør akkurat det samme som Floyd-Warshall, men sjekker om det finnes en vei fra i til j eller ikke, den er altså ikke opptatt av vektene. Samme kjøretid som FW. Bra når vi har få kanter, f.eks  $E = O(V^2)$ .

Hvis den går en sti mellom i og j i en graf G, så går det en kant mellom i og j i den transitive lukningen med verdi 1. Vi kan finne den transitive lukningen ved å kjøre Floyd-Warshall med kantvekter 1, og hvis  $d_{ij}$  er mindre enn n går det en sti, hvis ikke er den uendelig. Eller så kan vi bytte ut + med AND og OR, og definere

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left( t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right)$$

Som problemet som skal løses for hver kant (i, j).

Merk:  $t_{ij}^{(k)} = \text{det går en vei fra } i \text{ til } j \text{ via noder } \{1, \dots, k\}$

### Pseudokode:

Forenklet: Bruker bare én tabell

Utfyllende

```

TRANSITIVE-CLOSURE'(G)
1  n = |G.V|
2  initialize T
3  for k = 1 to n
4      for i = 1 to n
5          for j = 1 to n
6              t_{ij} = t_{ij} \vee (t_{ik} \wedge t_{kj})
7  return T

```

```

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)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})
13     return T^{(n)}

```

## Kjøretid:

Trippel for-løkke,  $\Theta(n^3)$ .

## Maksimal flyt

Kort fortalt: Ønsker å finne maks flyt fra source s til sink t. Maksimal flyt er nådd hvis og bare hvis residualnettverket ikke har flere flytforøkende stier.

**Flytnettverk** Et flytnettverk er en rettet graf, der alle kantene har en ikke-negativ kapasitet. I tillegg er det et krav at dersom det finnes en kant mellom u og v, finnes det ingen kant motsatt fra v til u rettet. Det er likevel lov å ha sykler. Et flytnettverk har en kilde, s, og et sluk, t. Kilden kan sees på som startnode, og sluket som sluttnode. Grafen er ikke delt, så for alle v finnes en vei  $s \sim v \sim t$ . Alle kanter bortsett fra s har en kant inn. En node, bortsett fra kilden og sluket, har like mye flyt inn som den har flyt ut (Kirchoffs første lov).

Et flytnettverk kan ha mange kilder og sluk. For å eliminere problemet, lager vi en superkilde og/eller et supersluk. Superkilden har en kant til hver av kildene, og kapasiteten på de kantene setter vi som uendelig. På samme måte lager vi supersluken. En kant fra hver av slukene, og setter kapasiteten til uendelig. Da er det et nytt nettverk, med kun en kilde og en sluk, og vi kan løse problemet som vanlig.

- Rettet graf  $G = (V, E)$
- Hver kant har en ikke-negativ kapasitet  $c(u, v)$
- $(u, v) \in E \Rightarrow (v, u) \notin E$
- Hvis  $(u, v) \notin E \Rightarrow c(u, v) = 0$
- Selv-løkker er ikke lov
- $s = \text{source}$ ,  $t = \text{sink}$
- Alle noder i grafen ligger på en sti fra s til t, dvs. grafen er sammenhengende og hver node unntatt s har minst én inngående kant.

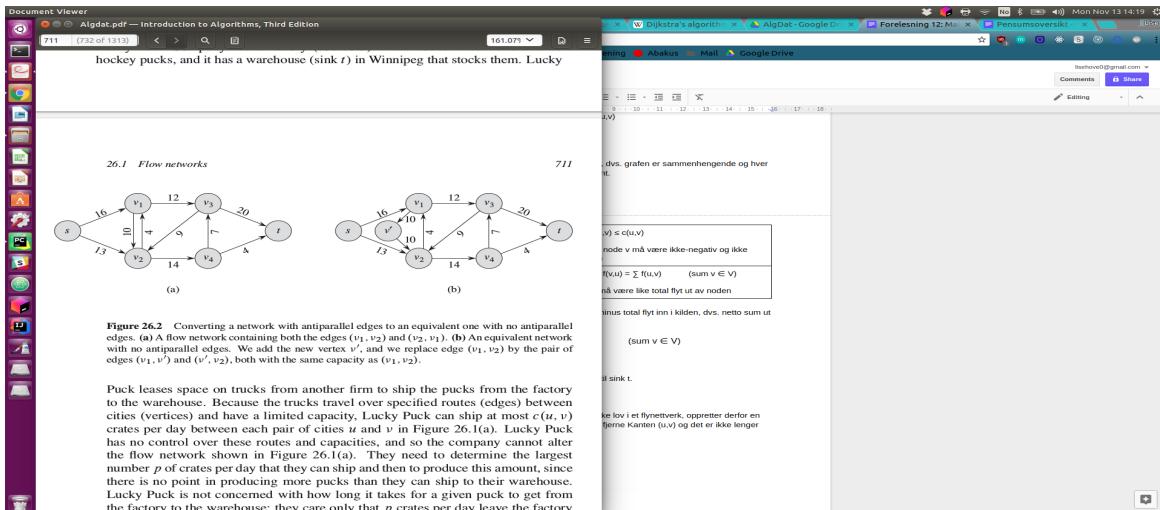
**Flyt** Flyt = funksjon  $f : V \times V \rightarrow \mathbb{R}$  som tilfredsstiller

Capacity constraint	For alle $u, v \in V$ , $0 \leq f(u, v) \leq c(u, v)$ Flyt fra en node $u$ til en node $v$ må være ikke-negativ og ikke overgå kapasiteten
Flow conservation	For alle $u \in V - \{s, t\}$ , $\sum f(v, u) = \sum f(u, v)$ Total flyt inn i en node (utenom kilde og sluk) må være like total flyt ut av noden

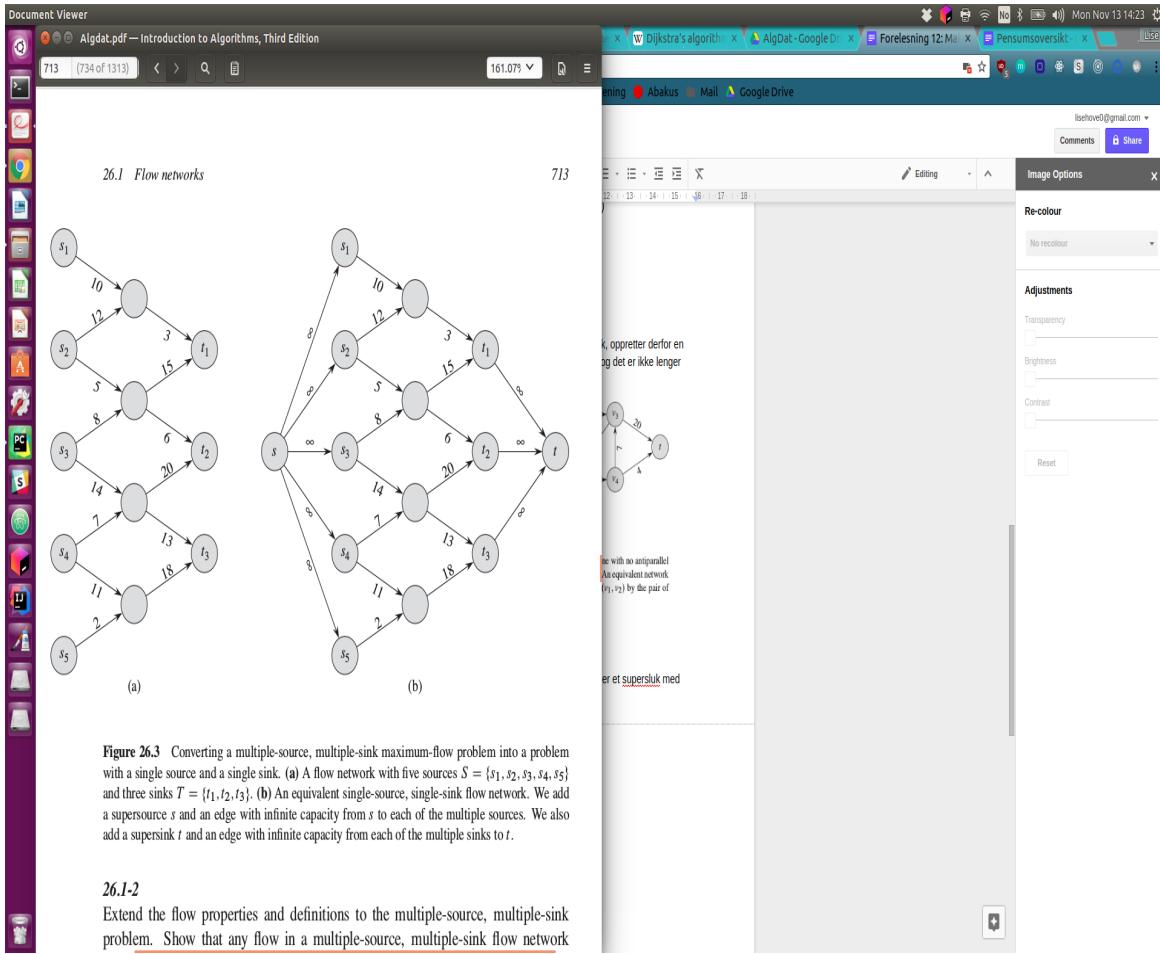
Verdien  $|f|$  er definert som total flyt ut av kilden minus total flyt inn i kilden, dvs. netto sum ut av kilden  $s$ . (Total flyt inn i kilden er ofte null)

$$|f| = \sum f(s, v) - \sum f(v, s) \quad (\text{sum } v \in V)$$

**Antiparalelle kanter** Kantene  $(u, v)$  og  $(v, u)$  er antiparalelle. Dette er ikke lov i et flynettverk, oppretter derfor en mellomnode  $x$  slik at  $c(u, x) = x(x, v) = x(v, u)$ . Kan da fjerne kanten  $(u, v)$  og det er ikke lenger noen antiparalelle kanter i flynettverket.



**Fleire kilder og sluk** Hvis det er flere kilder og/eller sluk kan vi innføre en superkilde og/eller et supersluk med  $c(\text{super, kilde}) = c(\text{sluk, super}) = \text{uendelig}$



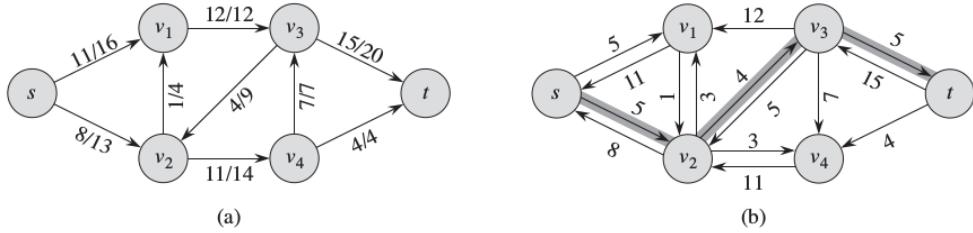
**Residualnettverk/Restnett** Representerer hvordan vi kan endre flyten. Ekstra flyt defineres som kapasiteten minus flyten. Hvis denne er positiv, legger vi inn denne “kanten” i residualnettverket med kapasitet

$$c_f(u,v) = c(u,v) - f(u,v)$$

Vi har kanter i begge retninger, “riktig retning” som representerer hvor mye mer flyt vi kan legge til, og “feil retning” som representerer hvor mye flyt vi kan reversere (kansellere flyten som allerede er der). På grunn av dette defineres residualkapasiteten

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Eksempel på (a) flytnettverk og (b) korresponderende residualnettverk:



**Augmenting path/Forøkende sti** En flytøkende sti (augmenting path) er en sti fra starten til en node, som øker total flyt i nettverket. En augmenting path er en enkel sti fra start  $s$  til sluk  $t$  i residualnettverket. Per definisjon av residualnettverket kan vi øke flyten  $f(u,v)$  i en augmenting path med  $c_f(u,v)$  uten å gå over begrensningene. Langs bakoverkanter må flyten omdirigeres.

En forøkende sti  $p$  er en enkel sti fra  $s$  til  $t$  i residualnettverket  $G_f$ . Vi kan øke flyten gjennom hver kant med den minste kant-verdien i stien, denne verdien heter residualkapasiteten av stien  $p$  og er gitt ved

$$c_f(p) = \min(c_f(u,v) : (u,v) \text{ er på stien } p)$$

Vi definerer en funksjon  $f_p$

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p, \\ 0 & \text{otherwise.} \end{cases}$$

altså, det vi kan øke stien  $p$  med for å få en ny sti nærmere maksimum flyt. Dvs.

$$|f \uparrow f_p| = |f| + |f_p| > |f|$$

Det bevises i boka at  $|f \uparrow f'| = |f| + |f'|$

**Flytopphaving** Vi kan «sende flyt baklengs langs kanter der det allerede går flyt. Vi opphever da flyten, så den omdirigeres til et annet sted det er dette bakoverkantene i restnettet representerer.

**Snitt/kutt** Et snitt  $(S,T)$  av et flytnettverk  $G = (V,E)$  er en oppdeling(snitt) av  $V$  i  $S$  og  $T = V - S$ , der kilde  $s \in S$  og sluk  $t \in T$ . Flyten over snittet er definert

$$f(S, T) = \sum_{s \in S} \sum_{t \in T} f(u, v) - \sum_{s \in S} \sum_{t \in T} f(v, u)$$

Antall mulige kutt totalt i et nettverk med  $n$  noder:

$$|C| = 2^{n-2}$$

altså summen av kantene fra S til T minus summen av kantene fra T til S (husk at kantene er rettet). For ethvert snitt er flyten f lik  $|f|$ , dvs. verdien av flyten.

**Minimalt snitt** Et minimalt snitt er et snitt der snitt-kapasiteten er minimum over alle snitt i nettverket. Av alle de mulige kuttene, ønsker vi å se på det kuttet som har minst flyt, da dette er flaskehalsen "i nettverket.

**Maks-flyt/min-snitt-teoremet** Maksimum flyt = minimalt snitt

Hvis f er flyten i nettverket  $G = (V, E)$  med kilde s og sluk t, er følgende ekvivalenser samme

- 1) f er maksimum flyt
- 2) Residualnettverket  $G_f$  har ingen forøkende stier
- 3)  $|f| = c(S, T)$  for et snitt  $(S, T)$  av G

Altså dersom f er maksimal flyt gjennom et nettverk, så er det minimale snittet sin snitt-kapasitet lik  $|f| =$  størrelsen på flyten.

## Maksimum bipartitt matching

### Matching

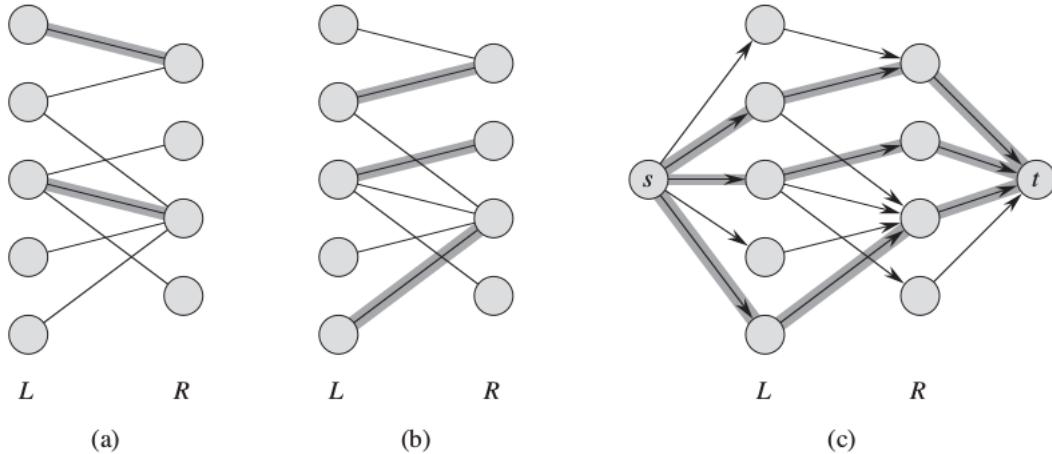
Matching er en delmengde av alle kaneter, der hver node er tilknyttet maks en kant fra delmengden; For en urettet graf  $G=(V,E)$  er en matching et subset av kantene,  $M \subseteq E$ , slik at for alle noder  $v \in V$  finnes det maks én kant assosiert med v. Vi sier at en node matcher med M dersom det finnes en kant i M assosiert med v, ellers er v unmatched.

### Maximal matching

Matching med maksimal kardinalitet, dvs. med maks antall kanter.  $|M| \geq |M'|$  for alle mulige matchinger  $M'$ .

## Bipartitt graf

Nodene kan bli partisjonert inn i  $L$  og  $R$ , der  $L$  og  $R$  er disjunkte og alle kantene  $E$  går mellom  $L$  og  $R$ . Antar også at alle nodene har minst én assosiert kant. Maksimum bipartitt matthing går ut på å finne flest mulige kanter mellom  $L$  og  $R$ , der ingen kant kan gå til eller fra samme kant.



**Figure 26.8** A bipartite graph  $G = (V, E)$  with vertex partition  $V = L \cup R$ . **(a)** A matching with cardinality 2, indicated by shaded edges. **(b)** A maximum matching with cardinality 3. **(c)** The corresponding flow network  $G'$  with a maximum flow shown. Each edge has unit capacity. Shaded edges have a flow of 1, and all other edges carry no flow. The shaded edges from  $L$  to  $R$  correspond to those in the maximum matching from (b).

## Ford-Fulkersons metode

I hver iterasjon av Ford-Fulkerson finner vi en flytforøkende sti  $p$ , og bruker  $p$  til å modifisere  $f$ . Merk at Ford-Fulkersons ikke spesifiserer hvordan dette implementeres. Ford-Fulkerson metoden finner maksimal flyt i et flytnettverk. Hver iterasjon forsøker å finne en flytforøkende sti, og setter på all den flyten som er mulig. Deretter leter den etter en ny flytforøkende sti, og gjentar prosessen. Når det ikke er flere flytforøkende stier har man oppnådd maksimal flyt. Den benytter seg av DFS for å finne flytforøkende sti.

- Dersom  $(u,v) \notin E$  antar vi at  $(u,v).f = 0$ .
- Antar at vi har kapasitetene  $c(u,v)$ , og at  $c(u,v)=0$  dersom  $(u,v) \notin E$
- $c_f(u,v)$  er definert som tidligere

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Fungerer ikke hvis kantkapasitetene er irrasjonelle tall.

Hvis maksimal flyt  $f^*$  er veldig høy kan algoritmen bli veldig treg. Eks. der det finnes en vei med flaskehals 1 som blir valgt som første forøkende sti. Da kan man ende opp med å øke flyten med 1 for hver iterasjon, og med veldig høy  $f^*$  vil dette resultere i dårlig kjøretid.

## Pseudokode:

```

FORD-FULKERSON-METHOD(G, s, t)
1 initialize flow  $f$  to 0
2 while there is an augm. path  $p$  in  $G_f$ 
3     augment flow  $f$  along  $p$ 
4 return  $f$ 

```

```

FORD-FULKERSON(G, s, t)
1 for each edge  $(u,v) \in G.E$ 
2      $(u,v).f = 0$ 
3 while there is a path  $p$  from  $s$  to  $t$  in  $G_f$ 
4      $c_f(p) = \min \{c_f(u,v) : (u,v) \text{ is in } p\}$ 
5     for each edge  $(u,v)$  in  $p$ 
6         if  $(u,v) \in E$ 
7              $(u,v).f = (u,v).f + c_f(p)$ 
8         else  $(v,u).f = (v,u).f - c_f(p)$ 

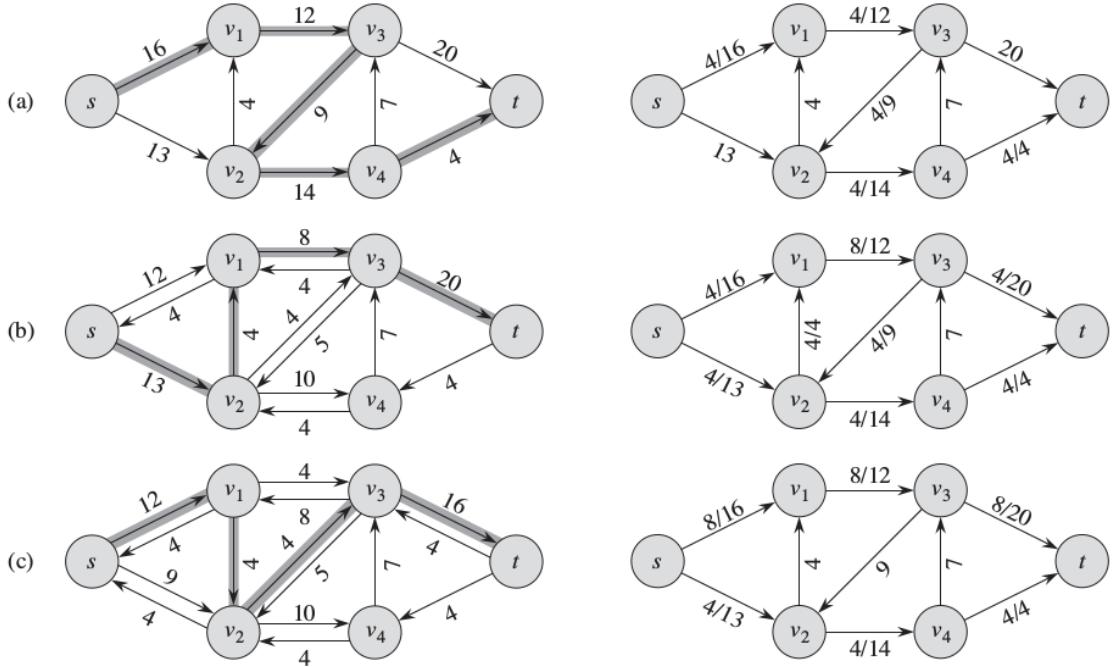
```

---

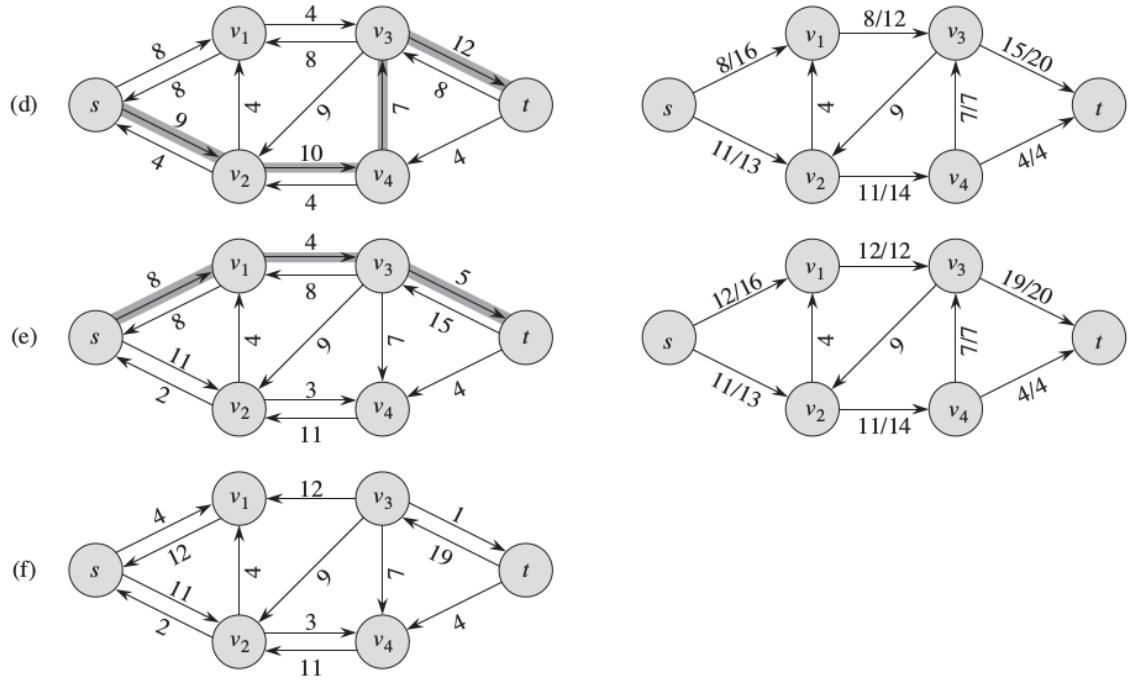
Best Case	Average Case	Worst Case
$O(VE^3)$		$O(Ef)$

---

Først initialiserer vi flyten mellom u og v,  $(u,v).f$ , til å være null for alle kanter i flytnettverket. Imens det fortsatt finnes en sti p fra s til t i residualnettverket er det fortsatt mulig å øke flyten. Vi finner residualkapasiteten til denne stien. Så går vi gjennom hver kant i stien p, dersom kanten er en kant i flytnettverket øker vi flyten med residualkapasiteten, hvis ikke minker vi flyten med residualkapasiteten (finner ut som vi skal øke flyt, eller sende flyt tilbake).



**Figure 26.6** The execution of the basic Ford-Fulkerson algorithm. (a)–(e) Successive iterations of the **while** loop. The left side of each part shows the residual network  $G_f$  from line 3 with a shaded augmenting path  $p$ . The right side of each part shows the new flow  $f$  that results from augmenting  $f$  by  $f_p$ . The residual network in (a) is the input network  $G$ .



**Figure 26.6, continued** (f) The residual network at the last **while** loop test. It has no augmenting paths, and the flow  $f$  shown in (e) is therefore a maximum flow. The value of the maximum flow found is 23.

## Edmonds-Karp

Edmonds-Karp er Ford-Fulkersons metode der BFS brukes for å finne augmenting path. Her finner vi flaskehalsen underveis. Det er da viktig å holde styr på hvor mye flyt vi får frem til hver node. Traverser bare noder vi ikke har nådd frem til enda. Dette gir en kjøretid på  $O(VE^2)$ .

**Pseudokode:**

```

1  EDMONDS_KARP(G, s, t)
2  for each edge (u, v) ∈ G.E
3    (u, v).f = 0
4  repeat until t.a == 0
5    for each vertex u ∈ G.V
6      u.a = 0 (reaching u in  $G_f$ )
7      u.π = NIL
8    s.a = ∞
9    Q = ∅
10   Enqueue(Q, s)
11   while t.a == 0 and Q ≠ ∅
12     u = DEQUEUE(Q)
13     for all edges (u,v), (v,u) ∈ G.E
14       if (u,v) ∈ G.E
15          $c_f(u,v) = c(u,v) - (u,v).f$ 
16       else
17          $c_f(u,v) = (u,v).f$ 
18       if  $c_f(u,v) > 0$  and v.a == 0
19         v.a = min(u.a,  $c_f(u,v)$ )
20         v.π = u
21         ENQUEUE(Q, v)
22     u, v = t.π, t (merk: t.a =  $c_f(p)$ )
23     while u ≠ NIL
24       if (u, v) ∈ G.E
25         (u, v).f = (u, v).f + t.a
26       else
27         (u, v).f = (u, v).f - t.a
28     u, v = u.π, u
29 until t.a == 0

```

**Heltallsteoremet** Om alle kapasiteter i flytnettverket er heltall, vil Ford-Fulkersons metode finne en maks flyt med en heltallsverdi, og flyten mellom alle nabonoder vil ha en heltallsverdi.

# Kjøretid graf-algoritmer

	Kjøretid	Brukso mråde	Fordeler
BFS	$O(V+E)$	Søk	
DFS	$\Theta(V+E)$	Søk	
MST-Kruskal	$O(E \lg V)$	MST	
MST-Prim	$O(E \lg V) / O(E+V \lg V)$	MST	
Bellman-Ford	$O(VE)$	SSSP	Kan ha negative <u>kantverkter</u> , og vil rapportere negative sykler.
DAG	$O(V+E)$	SSSP	
Djikstra	$O(E \lg V)$	SSSP	Kan bare ha positive kantverkter
Floyd-Warshall	$O(V^3)$	APSP	
Ford-Fulkerson	$O(E f^* )$	Max-Flo w	$f^*$ = maksimal flyt. Implementert med BFS blir det $O(VE^2)$
Topologisk sort	$\Theta(V+E)$		
Huffman	$O(n \lg n)$	PBC	

MST=Minimal spend tre, SSSP = Korteste vei én til alle, APSP = Korteste vei alle til alle, PBC = Finn den korteste binære prefifikskoden som representerer en tekst

# Problemkompleksitet

## P, NP, NPC

### P

Et sett av konkrete beslutningsproblemer som kan løses i polynomisk tid.

$$P = \{ L \subseteq \{0,1\}^* : \text{det eksisterer en algoritme } A \text{ som avgjør } L \text{ i polynomisk tid}\}$$

$$P = \{ L : L \text{ er akseptert av en polynomisk-tid algoritme}\}$$

01110111011011101011 ...



Bevis: Hvis A aksepterer L i  $O(n^k)$  tid for en konstant k, vet vi at det finnes en konstant c slik at A aksepterer L etter maks  $cn^k$  steg. Vi oppretter en algoritme A', slik at for enhver input x simulerer A'  $cn^k$  steg i A, og dersom A har akseptert x gir A' output 1(akseptert), og hvis den ikke har akseptert x gir A' output 0(avvist).

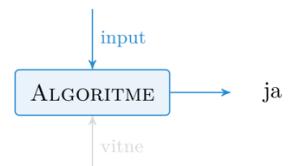
### NP

Et sett av konkrete beslutningsproblemer som kan verifiseres i polynomisk tid.  $NP =$  Nondeterministic polynomial. For en verifikasjonsalgoritme A og konstant c

$$L = \{ x \in \{0,1\}^* : \text{det eksisterer et sertifikat } y \text{ der } |y| = O(|x|^c) \text{ slik at } A(x,y)=1\}$$

Hvis vi kan løse et problem i polynomisk tid, så kan vi verifisere det i polynomisk tid, nettopp ved å løse det. Derfor er  $P \subseteq NP$ .

01110111011011101011 ...



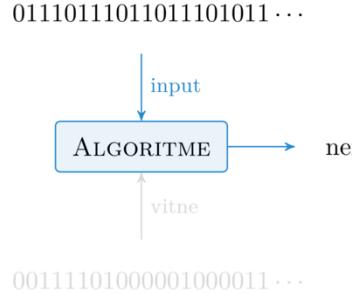
00111101000001000011 ...

For å tilhøre NP må problemet

- 1) Være et beslutningsproblem
- 2) Ha et endelig antall løsninger, der hver løsning har en polynomisk lengde

- 3) Gitt en polynomisk-lengde løsning, så burde vi kunne avgjøre om svaret var ja eller nei.

## co-NP



Et sett av språk  $L$  slik at  $L \in \text{NP}$ . Vi vet ikke om  $\text{co-NP} = \text{NP}$ , men vi vet at  $P \subseteq \text{NP} \cap \text{co-NP}$ .

Dersom man klarer å falsifisere løsningen i polynomisk tid, er problemet en del av co-NP-klassen. Kan falsifiseres i polynomisk tid

## NP-hardhet

Problemer kan ikke løses i polynomisk tid. Det sies å være en klasse av problemer som er minst like vanskelige som det vanskeligste problemet i NP-klassen. Sagt på en annen måte: problemer som lar seg redusere fra NP-problemer i polynomisk tid, men som ikke nødvendigvis lar seg verifisere i polynomisk tid med en gitt løsning. Hvis vi kan redusere fra NP-hardt  $H$  til et annet problem  $P$ , så må  $P$  også være NP-hardt, hvis ikke hadde ikke  $H$  vært NP-hardt. Vi kan redusere alle NP-problemer til NP-harde problemer, de er vanskeligere.

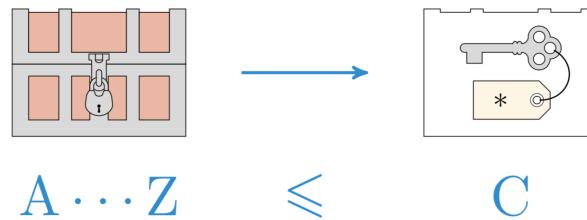
NP-harde problemer som lar seg verifisere i polynomisk tid kalles NP-komplette.

## NPC

De komplette «språkene» i NP, under polynomiske reduksjoner. For å være NPC, må det altså være NP-hardt og i NP.

Kompletthet: Et problem er komplett for en gitt klasse og en gitt type reduksjoner dersom det er maksimalt for redusibilitetsrelasjonen.

Maksimalt: Et element er maksimalt dersom alle andre er mindre eller lik. For reduksjoner:  $Q$  er maksimalt dersom alle problemer i klassen kan reduseres til  $Q$ .



NP-komplette er NP-harde problemer som også er i NP, dvs. vi kan redusere fra alle L i NP til problemet, og problemet er selv i NP. Hvis vi løser et NPC-problem, kan vi dermed løse alle NP-problemer ved å redusere til dette. Vanskelighetsgrad for å løse: P  $\rightarrow$  NP  $\rightarrow$  NPC  $\rightarrow$  NPH

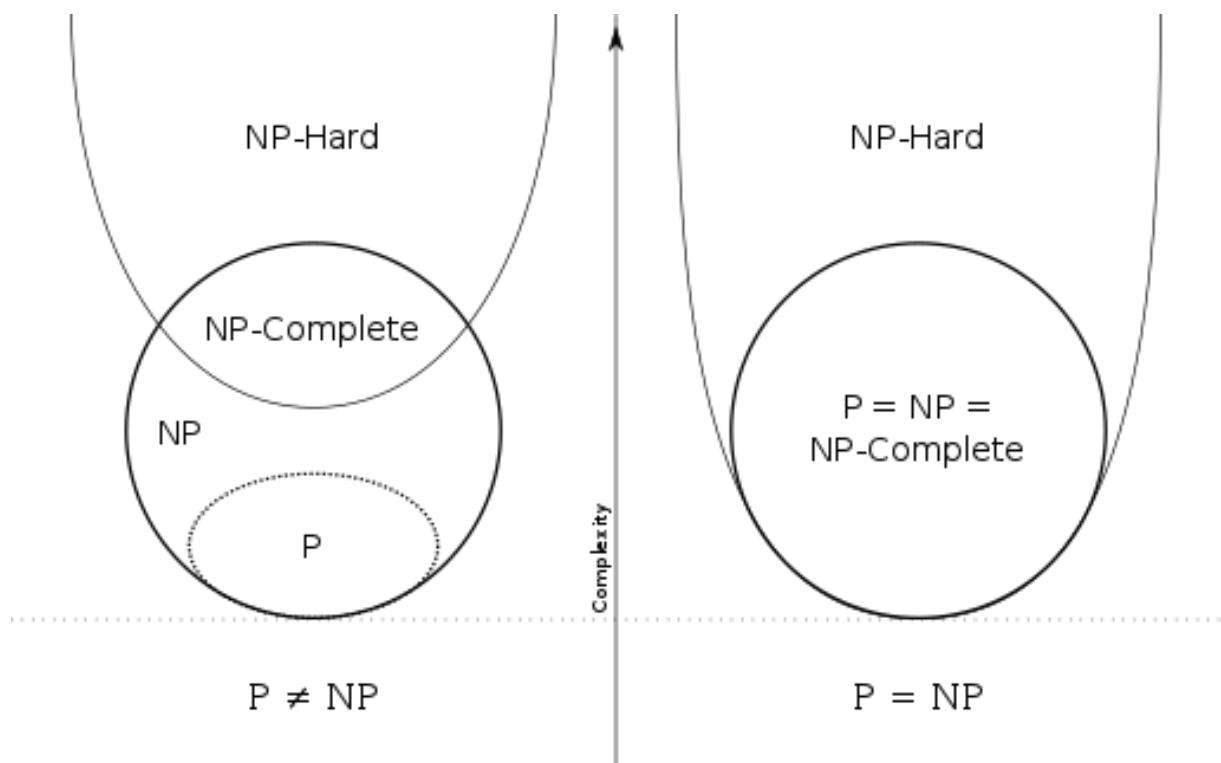
### Formell definisjon: NP-hardhet og NP-kompleksitet

Gitt et språk L  $\subseteq \{0,1\}^*$

- 1) L  $\in$  NP
- 2) L'  $\leq_p$  L for enhver L'  $\in$  NP

Dersom språket L oppfyller krav 2, men ikke nødvendigvis krav 1, sier vi at L er **NP-hardt**. Dersom det oppfyller begge kravene sier vi at L er **NP-komplett**.

NPC = Klasse av NP-komplette språk.



Man vet at P er et subsett av NP, men man vet ikke om P = NP.

P	Polynomial time
NP	Nondeterministic polynomial time
NPC	Nondeterministic polynomial time complete
NPH	Nondeterministic polynomial time hard

## Sammenhengen mellom alle NP-problemer:

Om vi kan løse problemet, så kan vi også verifisere det med samme algoriteme, og bare ignorere sertifikatet:  $P \subseteq NP$  og  $P \subseteq co-NP$

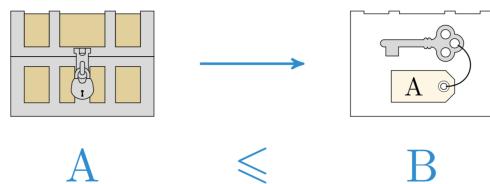
Vet ikke om:  $P = NP$   $co-NP$

$NPC \subseteq NP$

Hvis et problem  $A \in NPC$ , sa vil også  $A \in NP$ . Dette er fordi  $NP = P + NPC$ . Hvis noe ikke er i  $P$  eller  $NPC$ , er det heller ikke i  $NP$ , fordi  $P \cap NPC = \emptyset$ .

Du vet at problem  $A$  er i  $NP$  og problem  $B$  er i  $NPC$ . Du vil vise at  $A$  også er i  $NPC$ . Da reduserer du fra  $B$  til  $A$ . Tenk kister der  $B$  inneholder nøkkelen til  $A$ . Om man kan løse  $B$  kan man løse  $A$ .

Redusibilitet: Hvis  $A$  kan reduseres til  $B$  i polynomisk tid, skriver vi  $A \leq_p B$



Du står overfor de tre problemene  $A$ ,  $B$  og  $C$ . Alle tre befinner seg i mengden  $NP$ . Du vet at  $A$  er i mengden  $P$  og at  $B$  er i mengden  $NPC$ . Anta at du skal bruke polynomiske reduksjoner mellom disse problemene til å vise . . . :

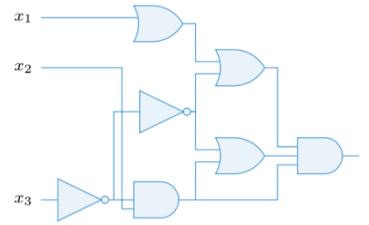
1. ... at  $C$  er i  $P$  må  $C \leq A$  ( $C$  reduseres til  $A$ )
2. ... at  $C$  er i  $NPC$  må  $B \leq C$  ( $B$  reduseres til  $C$ )
3. ... hvis  $B$  kan reduseres til  $A$  er  $P = NP$  (NB: ikke løst enda)
4. Alle disse reduksjonene skjer i polynomisk tid

## Redusibilitets-relasjon

For å forstå beviseteknikken som brukes for å bevise at et problem er  $NPC$ , er det et par begreper som må på plass. Ett av de er redusibilitets-relasjonen  $\leq_p$ . I denne sammenhengen brukes det for å si at et språk ( $L_1$ ) er polynomisk-tid redusertbart til et annet språk ( $L_2$ ):  $L_1 \leq_p L_2$ .

Boken (Cormen m.fl) trekker frem et eksempel der førstegradslikningen  $ax+b=0$  kan transformeres til  $0x^2+ax+b=0$ . Alle gyldige løsninger for annengradslikningen er også gyldige løsninger for førstegradslikningen. Ideen bak eksempelet er å vise at et problem,  $X$ , kan reduseres til et problem,  $Y$ , slik at inputverdier for  $X$  kan løses med  $Y$ . Kan du redusere  $X$  til  $Y$ , betyr det at å løse  $Y$  krever minst like mye som å løse  $X$ , dermed må  $Y$  være minst like vanskelig å løse som  $X$ . Det er verdt å merke seg at reduksjonen ikke er gjensidig, du kan dermed ikke bruke  $X$  til å løse  $Y$ .

## Noen kjente NPC problemer



Noen vanlige eksempler på problemer som er NPC:

### CIRCUIT-SAT

**Instans:** En kombinatorisk logisk krets med én utverdi, input = sertifikat

**Spørsmål:** Er det mulig å tilfredsstille kretsen? (Finnes det en kombinasjon innverdier som gir utverdi 1?)

### SAT (satisfiability)

$$\phi = ((x_1 \rightarrow x_2) \vee \\ \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

**Instans:** En logisk formel

**Spørsmål:** Kan formelen være sann? (bli 1)

Vi vet at SAT er i NP fordi vi kan gi den en løsning, som er verdier til de ulike bolske verdiene, sette de inn i funksjonen og få ut svaret, i polynomisk tid.

Vi må så vise at vi kan redusere fra et NPC problem til SAT. Vi kan redusere fra CIRCUIT-SAT til SAT. Vi kan ikke redusere ved å ta en direkte oversettelse av kretsen, for dette er ikke polynomisk. Vi må skrive en formel, som jeg ikke skjønte noe særlig av hvordan de kom frem til. Hvis formelen er tilfredsstilt, så har hver ledning i kretsen en veldefinert verdi, og outputen til kretsen er 1 (??).

### 3-CNF-SAT (Conjunctive normal form)

**Instans:** En logisk formel på 3-CNF-form, dvs. AND av tre og tre OR-er,

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge \\ (\neg x_1 \vee x_2 \vee x_3) \wedge \\ (x_1 \vee x_2 \vee x_3)$$

eks.  $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (x_3 \vee x_4 \vee x_1)$ .

**Spørsmål:** Kan formelen være sann? (bli 1)

Vi vet at 3-CNF-SAT er i NP av samme argument som for SAT.

Vi vil så vise at vi kan redusere fra SAT til 3-CNF-SAT. Gjør dette ved å vise at vi kan skrive om formelen i SAT til CNF, og skrive om dette igjen til 3-CNF.

### CLIQUE

**Instans:** Urettet graf og heltall k

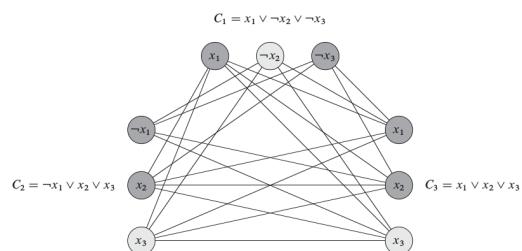
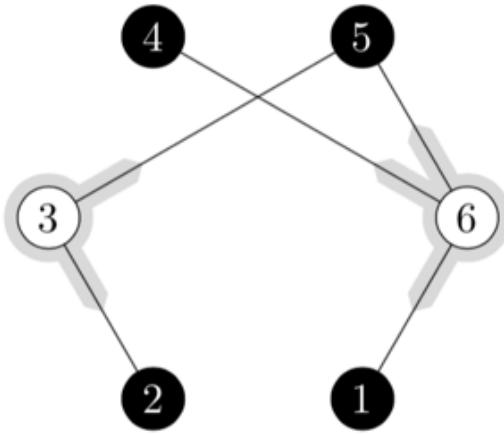
**Spørsmål:** Har grafen en komplett delgraf med k noder?

Vi vet at CLIQUE er i NP fordi hvis vi får en løsning kan vi fint sjekke om den har k noder.

Vi vil så vise at vi kan redusere fra 3-CNF-SAT til CLIQUE. La hver literal være en node i et tre, og la det gå kanter mellom alle noder som kan være 1 samtidig (altså ikke komplementer av hverandre), går heller ikke kanter mellom literaler i samme parantes, eks. for  $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (x_3 \vee x_4 \vee x_1)$  vil ikke noen av nodene i  $(x_1 \vee x_2 \vee x_3)$  ha en kant mellom seg. Vi har k slike trippel-ledd. For en løsning av 3-CNF-SAT, har vi at minst én node i hver av trippel-leddene er 1, ved å velge én av nodene fra hver av trippel-leddene som er 1 får vi en komplett delgraf med k noder.

For to noder i grafen som ikke er fra samme trippel-ledd, og begge er 1, så kan de ikke være komplementer av hverandre, og der går derfor en kant mellom de.

Hvis vi har en clique. Ingen kanter i samme trippel-ledd har kanter mellom seg, så cliquen har kun én node per trippel-ledd. Vi kan si at alle nodene i cliquen er 1, siden det ikke går noen kanter mellom komplimenter av noder. Tar vi de verdiene tilbake til formelen til 3-CNF-SAT vil den gi 1 fordi hvert trippel-ledd har minst ett ledd som er 1, nemlig noden i cliquen. CLIQUE er altså NPC.



## VERTEX-COVER

**Instans:** En urettet graf og et heltall k

**Spørsmål:** Har grafen et nodedekke med k noder? Dvs. k noder som til sammen har minst en kant til alle andre noder (de k nodene trenger ikke ha en kant til seg av en av de andre k nodene).

VERTEX-COVER er i NP fordi for et nodedekke  $V'$ , sjekker vi først at den har k noder, og deretter at for hver kant  $(u,v)$  i den originale grafen, så er enten u eller v (eller begge) med i  $V'$ .

Vi må redusere fra CLIQUE.

Reduserer ved å invertere alle kantene,  $(u,v) \in E \Leftrightarrow (u,v) \notin E$

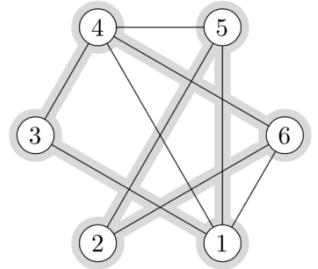
CLIQUE  $\Rightarrow$  VERTEX-COVER

Anta G har en klikk  $V'$ ,  $|V'| = k$ . La  $(u,v) \in E \Rightarrow (u,v) \notin E \Rightarrow u \text{ eller } v \notin V'$  (da hadde de vært med i klikken og  $(u,v) \in E \Leftrightarrow u \text{ eller } v \in V-V' \Rightarrow (u,v)$  er dekket av  $V-V'$ ).  $(u,v)$

var en tilfeldig kant, så alle kanter  $(u,v) \in E$  er dekket av  $V - V'$ , med  $|V| - k$  noder, som gir et  $|V| - k$  stort nodedekke.

**VERTEX-COVER => CLIQUE**

Anta  $G$  har et vertex cover  $V'$ ,  $|V'| = |V| - k$ . For alle  $u, v \in V$ , hvis  $(u, v) \in E$ , så må  $u$  eller  $v$  være med i  $V'$ . Så hvis verken  $u$  eller  $v$  er med i  $V'$ , så går det ikke noen kant mellom  $u$  og  $v$  i  $E$ , som betyr at det går en kant mellom  $(u, v)$  i  $E$ . For alle  $v \notin V'$ , går det en kant mellom  $v$  og de danner en klikk på  $|V| - |V'| = k$  noder.



### HAM-CYCLE

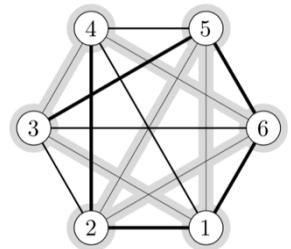
**Instans:** En urettet graf

**Spørsmål:** Finnes det en sykel som inneholder alle nodene?

HAM-CYCLE er i NP fordi hvis vi får en sykel, er det lett å sjekke at den er innom alle nodene, og at startnode = sluttoden i polynomisk tid.

Vi må redusere fra VERTEX-COVER til HAM-CYCLE.

Helt syre.



### TSP (traveling salesman problem)

**Instans:** En komplett graf med heltallsvekter og et heltall  $k$

**Spørsmål:** Finnes det en rundtur med kostnad  $\leq k$ ?

TSP er i NP fordi gitt en løsning, kan vi sjekke om den inneholder alle nodene, legge sammen kantvektene og sjekke om dette er mindre eller lik  $k$ .

Vi må redusere fra HAM-CYCLE til TSP.

Reduksjonen: Opprett en komplett graf  $G'$  med kantvekt 0 om  $(u, v)$  er i  $G$  og 1 ellers.

**HAM-CYCLE => TSP**

Antar at  $G$  har en HAM-CYCLE  $h$ , slik at hver kant  $i \in h$  er med i  $G$ . Siden hver kant er med i  $G$  er kostnaden til denne kanten i  $G'$  lik 0, og summen av kantene er 0,  $h$  er altså også en løsning på TSP, en sykel innom alle nodene med vektsum 0.

**TSP => HAM-CYCLE**

Antar  $G'$  har en sykel  $h'$  med maks kostnad 0. Siden alle kantene har kostnad 0 og 1, må alle kantene i  $h'$  være 0, følgelig er kantene i  $h'$  også kanter i  $E$ , så  $h'$  er også en Hamiltonsykel for  $G$ .

## SUBSET-SUM

1    2    4    7    9

**Instans:** Et sett  $S$  med positive heltall og et positivt heltall  $t$

**Spørsmål:** Finnes det et subsett av  $S$  der summen av elementene er lik  $t$ ?

Bevis er ikke pensum.

SUBSET-SUM er i NP fordi vi fint kan summere tallene i subsettet og sjekke om det er lik  $t$

Vi må redusere fra 3-CNF-SAT til SUBSET-SUM.

### 0-1-ryggsekkproblemet er NP-hardt

Kjøretiden er  $O(nW)$ , der  $n$  er antall gjenstander og  $W$  er maks vekt til ryggsekken. Tilsynelatende polynomisk, men det er ikke det. Når vi bruker  $n$  som inputstørrelse, bruker vi egentlig antall elementer som inputstørrelse. Når vi bruker  $W$  som inputstørrelse bruker vi selve verdien av tallet  $W$ , dvs. hvis vidobler  $W$  vil antall bits i  $W$  øke med én, mens kjøretiden vil doble seg. Hvis vi øker antall elementer i  $n$  med én, vil kjøretiden øke med én. Kjøretiden vil altså vokse eksponentielt når vi øker  $W$  med én (bit), mens hvis vi øker  $n$  med én blir kjøretiden  $O((n+1)W) = O(nW)$ , dvs ingen endring. Ryggsekkproblemet har altså pseudopolynomisk kjøretid, det ser polynomisk ut men er eksponentiell. Det er altså NP-hardt.

Dobler  $n \Rightarrow$  dobbelt så lang liste  $\Rightarrow O(2nW)$

Dobler  $W \Rightarrow$  dobler antall bits  $W$  er representert ved  $\Rightarrow = (n2^{\lg w} + \lg w = 2\lg w) = (nw^2)$ . Vokser eksponentielt i forhold til  $n$ .

### Lengste-enkle-vei-problemet er NP-hardt

Redusere fra HAM-CYCLE, en graf har en HAM-CYCLE hvis det finnes en lengste vei på  $n-1$  kanter. Det er ikke et beslutningsproblem, men et optimaliseringsproblem, så det er NP-hardt. Hvis vi har en HAM-CYCLE, kan vi splitte en node og få en lengste vei på  $n-1$  kanter. Hvis vi har en lengste vei på  $n-1$  kanter, kan vi slå sammen to noder og få en HAM-CYCLE.

## Bevise NP-kompletthet ved én reduksjon

Istedenvor å bevise at et språk  $L$  er NP-komplett ved å vise at vi kan redusere fra alle NP-språk til  $L$ , kan vi vise at  $L$  er NP-komplett ved å redusere fra ett annet NPC språk,  $L'$ , til  $L$ . Hvis  $L$  også er i NP, så er  $L$  NP-komplett.

Bevis: Hvis vi har et NPC språk  $L'$ , så vet vi at for alle  $L''$  i NP, så kan vi redusere fra  $L''$  til  $L'$  i polynomisk tid. Hvis vi da kan redusere fra  $L'$  til  $L$ , så kan vi (transitivitet) redusere fra  $L''$  til  $L$ , som viser at  $L$  er NP-hardt. Hvis  $L$  kan verifiseres polynomisk, er da  $L$  NPC.

For å bevise at et språk er NP-komplett må vi altså

- 1) Vise at  $L \in \text{NP}$
- 2) Velge et NPC språk  $L'$
- 3) Beskrive en algoritme som mapper hver instans  $x \in \{0,1\}^*$  av  $L'$  til en instans  $f(x)$  av  $L$
- 4) Vise at denne funksjonen  $f$  tilfredsstiller  $x \in L' \Leftrightarrow f(x) \in L$  for alle  $x \in \{0,1\}^*$
- 5) Vise at denne algoritmen kjører i polynomisk tid

Det vil si

- 1) Vise at  $L$  er i NP
- 2) Velge et kjent språk  $L'$  som er i NPC
- 3) Vise at vi kan redusere fra  $L'$  til  $L$  i polynomisk tid

# Kjøretider til pensum-algoritmer

- $n$  er antallet elementer som skal jobbes med.
- $k$  er den høyeste verdien tallene kan ha.
- $d$  er maks antall siffer et tall kan ha.

Problem	Algoritme	Kjøretid			Stabil	In place
		BC	AC	WC		
Sortering	Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	Ja	Ja
	Bubble Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	Ja	Ja
	Merge Sort	$\Theta(n \lg(n))$	$\Theta(n \lg(n))$	$\Theta(n \lg(n))$	Ja	Nei
	Heap Sort	$O(n)$	$O(n \lg(n))$	$O(n \lg(n))$	Nei	Ja
	Quicksort	$\Theta(n \lg(n))$	$\Theta(n \lg(n))$	$\Theta(n^2)$	Nei	Ja
	Counting Sort	$\Theta(k + n)*$	$\Theta(k + n)*$	$\Theta(k + n)*$	Ja	Nei
	Radix Sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$	$\Theta(d(n + k))$	Ja*	Nei
	Bucket Sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2)$	Ja	Nei
Søk	Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	Nei (Ja, m/ lenket liste)	Ja
	Binærsøk	$O(n)$	$O(\lg(n))$	$O(\lg(n))$	-	-
	Select	$O(n)$	$O(n)$	$O(n)$	-	-
	Randomized select	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2)$	-	-

\* Radix sort krever en stabil subroutine for selv å være stabil.

\* Dersom  $k = O(n)$ , får vi kjøretid  $\Theta(n)$

	Kjøretid	Bruksområde	Fordeler
BFS	$O(V+E)$	Søk	
DFS	$\Theta(V+E)$	Søk	
MST-Kruskal	$O(E \lg V)$	MST	
MST-Prim	$O(E \lg V) / O(E+V \lg V)$	MST	
Bellman-Ford	$O(VE)$	SSSP	Kan ha negative kanterverkter, og vil rapportere negative sykler.
DAG	$O(V+E)$	SSSP	
Dijkistra	$O(E \lg V)$	SSSP	Kan bare ha positive kantverkter
Floyd-Warshall	$O(V^3)$	APSP	
Ford-Fulkerson	$O(E f^* )$	Max-Flow	$f^*$ = maksimal flyt. Implementert med BFS blir det $O(VE^2)$
Topologisk sort	$\Theta(V+E)$		
Huffman	$O(n \lg n)$	PBC	

MST=Minimal spend tre, SSSP = Korteste vei én til alle, APSP = Korteste vei alle til alle, PBC = Finn den korteste binære prefikskoden som representerer en tekst

# Pseudokode med kommentarer

## Merge sort

```
1 #Sorter A[p..r] ved å sortere halvdelene først
2 MERGESORT(A, p, r)
3 #Hvis A[p..r] har lengde minst 2..
4 if p < r
5     #Del på midten
6     q = [(p + r)/2]
7     #Sorter A[p..q] rekursivt
8     MERGESORT(A, p, q)
9     #Sorter A[q+1..r] rekursivt
10    MERGESORT(A, q + 1, r)
11    #Flett sorteret segmenter A[p..q] og A[q+1..r]
12    MERGE(A, p, q, r)
```

Forenklet versjon av Merge:

```
14 #Forenklet versjon
15 #Flett sorterte segmenter A[p..q] og A[q+1..r]
16 MERGE(A, p, q, r)
17 copy into L and R
18 #Vi setter inn i A[p..r]. For hver indeks k, bruk min{L[i], R[j]}
19 for k = p to r
20     #Hvis L[i] er minst
21     if L[i] ≤ R[j]
22         #Bruk L[i]
23         A[k] = L[i]
24         #Vi har brukte L[i], så flytter oss videre
25         i = i + 1
26     else A[k] = R[j]
27         j = j + 1
```

Utfyllende versjon av merge:

```
1 #Utfyllende versjon
2 #Flett sorterte segmenter A[p..q] og A[q+1..r]
3 MERGE(A, p, q, r)
4 ### Kopier inn i L and R ###
5 #Lengden til første segment, A[p..q]
6 n1 = q - p + 1
7 #Lengden til andre segment, A[p+1..r]
8 n2 = r - q
9 #Vi kopierer midlertidig innholdet i A[p..q] og A[q+1..r] til L og R
10 let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
11 #For hver indeks i i L
12 for i = 1 to n1
13     #Kopier tilsvarende element fra A[p..q]
14     L[i] = A[p + i - 1]
15 #For hver indeks i i R
16 for j = 1 to n2
17     #Kopier tilsvarende element fra A[q+1..r]
18     R[j] = A[q + j]
19 #Vaktpost (sentinel) så vi slipper å sjekke om vi er ved slutten
20 L[n1 + 1] = ∞
21 #Vaktpost (sentinel) så vi slipper å sjekke om vi er ved slutten
22 R[n2 + 1] = ∞
23 #Hvor er vi i L?
24 i = 1
25 #Hvor er vi i R?
26 j = 1
27 ### Slutt av kopiering ###
28 #Vi setter inn i A[p..r]. For hver indeks k, bruk min{L[i], R[j]}
29 for k = p to r
30     #Hvis L[i] er minst
31     if L[i] ≤ R[j]
32         #Bruk L[i]
33         A[k] = L[i]
34         #Vi har brukt L[i], så flytter oss videre
35         i = i + 1
36     else A[k] = R[j]
37         j = j + 1
```

## Quicksort

```
1 #Sorterer A[p..r] ved å sortere små og store verdier hver for seg
2 QUICKSORT(A, p, r)
3 #Hvis A[p..r] har lengde minst 2...
4 if p < r
5     #A[p...q-1] ≤ A[q] ≤ A[q+1...r]. Merk at A[q] er på rett plass
6     q = PARTITION(A, p, r)
7     #Sorter små verdier A[p...q-1] rekursivt
8     QUICKSORT(A, p, q - 1)
9     #Sorter store verdier A[q+1...r] rekursivt
10    QUICKSORT(A, q + 1, r)
11
12 #Del to segmenter: Ett med små og ett med store verdier
13 PARTITION(A, p, r)
14 #Splittelement (pivot): Vilkårlig grense for hva vi kaller smått
15 x = A[r]
16 #Siste indeks i segmentet med små verdier. Foreløpig tomt
17 i = p - 1
18 #Siste indeks i segmentet med store verdier
19 for j = p to r - 1
20     #A[j] er liten: Må flyttes til segmentet med små verdier
21     if A[j] ≤ x
22         #Utvid segmentet med små verdier. A[i] foreløpig stor
23         i = i + 1
24         #Både A[i] og A[j] er i feil segment, så de bytter plass
25         exchange A[i] with A[j]
26 #Splittelementet er nå på riktig (sortert) plass, mellom segmentene
27 exchange A[i + 1] with A[r]
28 #Returnerer hvor splitten havnet
29 return i + 1
```

## Heap sort

```
1 # Sorter A "in place", ved ombyttinger
2 HEAPSORT(A)
3 # Gjør hele A til en haug (i lineær tid)
4 BUILD_MAX_HEAP(A)
5 # Invariant: Alt utenfor haugen er større, og sortert
6 for i = A.length downto 2
7     # Flytt største element sist i haugen
8     exchange A[1] with A[i]
9     # Vedlikehold av invariant: Alt utenfor er fortsatt større, sortert
10    A.size = A.size-1
11    # Reparer den nye rota, så vi fortsatt har en haug
12    MAX_HEAPIFY(A, 1)
```

```
1 # Vedlikehold av heaps
2 # La A[i] "synke" ned til rett plass. Induktivt: deltrær er heaps
3 MAX_HEAPIFY(A, i)
4 l = Left(i)
5 r = Right(i)
6 # Mindre enn venstre barn? Det må fikses!
7 if ≤ A.size and A[l] > A[i]
8     # m er noden med størst verdi blant forelder (i) og barn (l, r)
9     m = l
10 # m er noden med størst verdi blant forelder (i) og barn (l, r)
11 else
12     m = i
13 # m er noden med størst verdi blant forelder (i) og barn (l, r)
14 if ≤ A.size and A[r] > A[m]
15     # m er noden med størst verdi blant forelder (i) og barn (l, r)
16     m=r
17 # Er foreldrenoden mindre enn minst ett av barna?
18 if m ≠ i
19     # Bytt plass med største barn
20     exchange A[i] with A[m]
21     # Har nå kanskje ødelagt et deltre. Fiks det rekursivt
22     Max-Heapify(A, m )
```

```
1 # Bygge heap
2 # Fiks alle deltrær. Induksjon på tre størrelser
3 BUILD_MAX_HEAP(A)
4 # Hele A skal bli en haug
5 A.size = A.length
6 # Grunntilfelle: Trær av størrelse 1 er alt korrekte
7 for i = [A.length/2] down to 1
8     # Induktivt premiss: Deltrær er korrekte.
9     # Induktivt trinn: Fiks rota (i)
10    MAX_HEAPIFY(A, i)
```

## Insertion sort

```
1  INSERTION_SORT(A)
2  # Induksjonshypotese: listen A er sortert
3  for j = 2 to A.length
4      key = A[j]
5
6      #Induksjonstrinn: Sett inn A[j] på rett plass
7      #Iterer nedover fra forrige element
8      i = j - 1
9      #Sjekker om A[i] > A[j]
10     while i > 0 and A[i] > key
11         #Gjør plass et hakk lenger ned
12         A[i + 1] = A[i]
13         i = i - 1
14     #Større element er nå til høyre
15     A[i + 1] = key
```

## Counting sort

```
1  # A, B, k: input, output, maksverdi (A[i] \in {0, ..., k})
2  COUNTING_SORT(A, B, k)
3  # Antall forekomster av hver mulig verdi
4  let C[0..k] be a new array
5  # For hver mulig verdi i...
6  for i = 0 to k
7      # Ingen forekomster funnet av i ennå
8      C[i] = 0
9  # For hver posisjon i input-tabellen...
10 for j = 1 to A.length
11     # Vi har nå sett enda en forekomst av A[j]
12     C[A[j]] = C[A[j]] + 1
13 # For hver av tellingene unntatt 0...
14 for i = 1 to k
15     # Gjør C kumulativ: C[i] = Hvor skal den siste i-en være?
16     C[i] = C[i] + C[i - 1]
17 # Vil ikke bytte like verdier; C angir siste forekomster
18 for j = A.length down to 1
19     # C[i] = Hvor skal den siste i-en være? (i = A[j])
20     B[C[A[j]]] = A[j]
21     # Den neste i-en skal være ett hakk til venstre
22     C[A[j]] = C[A[j]] - 1
```

## Radix Sort

```
1 # Må være stabil og ikke bytte like verdier (bruk f.eks counting sort)
2 RADIX_SORT(A, d)
3 for i = 1 to d
4     sort A by digit d
5
6
7 XIDAR_NON_SORT(A, d)
8 for i = d down to 1
9     sort A on digit d
```

## Bucket sort

```
1 # Verdier i A: Uniformt fordelte, [0, 1)
2 BUCKET_SORT(A)
3 # Antall bøtter; ett element per bøtte
4 n = A.length
5 # Hodepekere til bøttene (lister)
6 create B[0..n-1]
7 # Iterere gjennom alle bøttene
8 for i = 1 to n
9     # (Evt. bruk dynamiske tabeller)
10    make B[i] an empty list
11 for i = 1 to n
12    # "Ryddig hashing"; Kan regne ut rett bøtte direkte
13    add A[i] to B[nA[i]]
14 for i = 0 to n-1
15    # Bruk f.eks INSERTION_SORT. Forventet: B[i].length = O(1)
16    sort list B[i]
17 # Ferdig sorterte, i rett rekkefølge
18 concatenate B[0] . . . B[n-1]
```

## Binærsoek

```
1 # A er sortert. Om mulig, finn en indeks i så A[i] == v
2 BINARY_SEARCH(A, p, r, v)
3 # Har vi i det hele tatt et segment å lete i?
4 if p ≤ r
5     # Del på midten
6     q = [(p + r)/2]
7     # Er dette verdien vi leter etter?
8     if v == A[q]
9         # Isåfall er vi ferdige
10        return q
11    # Ellers: må vi ligge til venstre
12    elseif v < A[q]
13        # Let videre til venstre
14        return BINARY_SEARCH(A, p, q - 1, v)
15    # Ellers må v ligge til høyre, så vi leter videre der
16    else
17        return BINARY_SEARCH(A, q + 1, r, v)
18 # Vi fant ikke v, så vi returnerer NIL
19 return nil
```

## Randomized select

```
1 # Finn det i-ende minste elementet i A[p .. r]
2 RANDOMIZED_SELECT(A, p, r, i)
3 # Bare ett element: Det må være det vi leter etter!
4 if p == r
5     return A[p]
6 # Nå er A[q] på rett plass, akkurat som i QUICKSORT!
7 q = Randomized-Partition(A, p, r)
8 # A[q] er det k-ende minste elementet i A[p .. r]
9 k = q - p + 1
10 # Er A[q] det i-ende minste elementet i A[p .. r]?
11 if i == k
12     # Da er det det vi leter etter
13     return A[q]
14 # Leter vi etter et mindre element?
15 elseif i < k
16     # Let bland de smp: Finn det k-ende minste i A[p..q-1]
17     return RANDOMIZED_SELECT(A, p, q - 1, i)
18 # Ellers: Finn det (i-k)-ende minste i A[q+1..r]
19 else
20     return RANDOMIZED_SELECT(A, q + 1, r, i - k)
```

## Select

Samme struktur som randomized-select: Se kommentarer der.

```
1 # Bytter ut randomized
2 # Partisjoneringen er kjernen: Finn en god pivot!
3 # Se RANDOMIZED_SELECT for kommentarer
4 SELECT(A, p, r, i)
5 if p == r
6     return A[p]
7 q = Good-Partition(A, p, r)
8 k = q - p + 1
9 if i == k
10    return A[q]
11 elseif i < k
12    return SELECT(A, p, q - 1, i)
13 else
14    return SELECT(A, q + 1, r, i - k)
```

```
1 # Tilhørende SELECT
2 # Velger pivot nøye. Beskrevet uten kode i boka
3 Good-Partition(A, p, r)
4 # n = A[p..r].length
5 n = r-p+1
6 # Deler A[p..r] i grupper på fem
7 m = [n/5]
8 # Vil inneholde medianen for hver av femmergruppen
9 create B[1 .. m]
10 # For hver femmergruppe..
11 for i = 0 to m-1
12     # Gruppen starter med A[q]
13     q = p + 5i
14     # For å finne medianen i gruppen. bruk f.eks INSERTION_SORT
15     sort A[q .. q + 4]
16     # Sett medianen inn i B
17     B[i] = A[q + 3]
18 # Finn medianen av medianene ... med SELECT!
19 x = SELECT(B, 1, m, [m/2])
20 Bruk medianen av medianene som pivot
21 return Partition-Around(A, p, r, x)
```

## Huffman

```
1 # Finn kode for tegn i C som minimerer forventet tekstlengde
2 HUFFMAN(C)
3 n = |C|
4 # Q er en prioritetskø, basser på frekvensene: Sjeldne tegn først!
5 Q = C
6 # Vi har n løvnoder; trenger n-1 interne
7 for i= 1 to n ≠ 1
8     allocate a new node z
9     x = EXTRACT_MIN(Q)
10    y = EXTRACT_MIN(Q)
11    # Slå sammen de to minste; gi dem felles foreldrenode
12    z.left, z.right = x, y
13    # Vi later som om deltreteet er et tegn
14    z.freq = x.freq + y.freq
15    # Sett dette "tegnet" tilbake i køen
16    Insert(Q, z)
17 # Tit slutt har vi bare en node i Q: Rota til treet
18 return EXTRACT_MIN(Q)
```

Lik Heap-extract max, men brukt på min-heap:

```
1 # Finn og fjern det største elementet i maks-haugen A
2 EXTRACT_MIN(A)
3 if A.size < 1
4     error "heap underflow"
5 # Det minste elementet er alltid først
6 min = A[1]
7 # Kan ikke bare forskyve A[1..n] til A[1..n-1].
8 A[1] = A[A.size]
9 A.size = A.size-1
10 # Bare rota kan være feil, nå. Fiks den.
11 MAX_HEAPIFY(A,1)
12 return min;
```

## Bredde først søk (BFS)

```
1 # Traverser noder oppdaget fra s, så noder oppdaget fra disse, etc
2 BFS(G, s)
3 ##### Først initialisering... #####
4 for each vertex u ∈ G.V-{s}
5     # Hvit = uoppdaget
6     u.color = white
7     # Beste gjetning på avstand fra s
8     u.d = ∞
9     # Hvilken node har vi oppdaget u fra?
10    u.π = NIL
11    # Grå = oppdaget men ikke besøkt
12    s.color = gray
13    # Avstand fra s til s
14    s.d = ∅
15    # Startnoden har aldri noen forgjenger
16    s.π = NIL
17    # "Huskelisten": En FIFO-kø. Oppdaget tidlig --> besøkt tidlig
18    Q = ∅
19    Før vi starter: "Skriv opp" startnoden på huskelisten
20    Enqueue(Q, s)
21
22 ##### Behandler nettverket #####
23 # Så lenge vi har oppdagede, ubesøkte noder...
24 while Q ≠ ∅
25     # ...velg den av dem vi oppdaget først
26     u = Dequeue(Q)
27     # For hver av nablene til den besøkte noden...
28     for each v ∈ G.Adj[u]
29         # Er noden uoppdaget
30         if v.color == white
31             # Ikke nå lenger!
32             v.color = gray
33             # s → v = s → u ⇒ v
34             v.d = u.d + 1
35             v.π = u
36             # Vi må huske å besøke den nyoppdagede noden etter hvert
37             Enqueue(Q, v)
38         # Svart = besøkt (og ferdigbehandlet)
39         u.color = black
```

## Dybde først søk (DFS)

```
1 # Besøk oppdagede noder umiddelbart
2 DFS(G)
3 # Først initialisering...
4 for each vertex u ∈ G.V
5     # Hvit = uoppdaget
6     u.color = white
7     # Hvilken node har vi oppdaget u fra?
8     u.π = NIL
9 # Vi må holde styr på når vi oppdaget og ferdigbehandlet hver node
10 # Merk: time er global
11 time = 0
12
13 ##### Behandler nettverket #####
14 # I denne implementasjonen: Traverser fra alle noder
15 for each vertex u ∈ G.V
16     # Ignorer noder vi alt er ferdige med
17     if u.color == white
18         Dette er den faktiske traverseringen
19         DFS_VISIT(G, u)
20
21
22 # Dybde-først-traversering fra u
23 DFS_VISIT(G, u)
24 # Når oppdaget vi u (discover-time)?
25 time = time+1
26 u.d = time
27 # u er oppdaget
28 u.color = gray
29 # For hver nab...o
30 for each v ∈ G.Adj[u]
31     # Er den uoppdaget?
32     if v.color == white
33         # Vi oppdager den nå, fra u
34         v.π = u
35         # Besøk u umiddelbart; kallstakken blir huskeliste!
36         DFS_VISIT(G, v)
37 # Vi er ferdige med u
38 u.color = black
39 # Når ble vi verdige me u (finish-time)?
40 time = time+1
41 u.f = time
```

## Kruskal

```
1 # Finn minimalt spennetre ved å velge minste lovlige kant
2 MST_KRUSKAL(G, w)
3 # Kantene som skal utgjøre et minimalt spennetre til slutt
4 A = ∅
5 # Initialisering for mengdeoperasjonene våre
6 for each vertex v ∈ G.V
7     # Vi starter med |V| individuelle partielle spenntrær
8     MAKE_SET(v)
9 # Preprosessering: Vil hele tiden velge minste kant
10 sort G.E by w
11 # altså i stigende rekkefølge, etter vekt
12 for each edge (u, v) ∈ G.E
13     # Mellom to ulike komponenter av den partielle løsningen?
14     if FIND_SET(u) ≠ FIND_SET(v)
15         # Da dannes ingen sykler, og vi kan velge kanten
16         A = A ∪ {(u, v)}
17         # Vi kobler sammen to partielle spenntrær til ett
18         Union(u, v)
19 return A

21 # Initialisering: Noden representerer mengden {x}
22 MAKE_SET(x)
23 # Mengden representeres av et tre; dette er foreldrenoden til x
24 x.p = x
25 # En slags "høyde"; største avstand til en løvnode
26 x.rank = 0

28 # Kombiner to mengder/trær
29 UNION(x, y)
30     # Finn røttene/"representantene" og koble dem sammen
31     LINK(FIND_SET(x), FIND_SET(y))
```

```

1 # To røtter; en henges under den andre
2 LINK(x, y)
3 # Utjevning: Den høyeste rang blir rot
4 if x.rank > y.rank
5     # Den nærmest "bunnen" blir barn av den andre
6     y.p = x
7 else
8     x.p = y
9     # Barn strengt mindre rang? Alt i orden. Ellers...
10    if x.rank == y.rank
11        # Vedlikehold betydning av rang: Avstanden til bunnen
12        y.rank = y.rank + 1

14 # Finn "representanten"/rota til mengden/treet som inneholder x
15 FIND_SET(x)
16 # Rota har seg selv som representant. Ikke der ennå...
17 if x != x.p
18     # ...så vi leter videre (rekursivt)
19     x.p = FIND_SET(x.p)
20 # Nå har x fått rota som forelder; snarvei til neste gang!
21 return x.p

```

## Prim

```
1 # Start i r. gjenta: Inkluder noden nærmest treeet
2 MST_PRIM(G, w, r)
3 # initialisering
4 for each u ∈ G.V
5     # Vekt til letteste kant i treeet
6     u.key = ∞
7     # Noden den letteste kanten kommer fra; utgjør MST til slutt
8     u.π = nil
9 # r er startnoden vår
10 r.key = 0
11 # Vi kunne ha traversert... men trenger ikke
12 Q = G.V
13 # Som ved traversering: Står det noe på huskelista vår?
14 while Q ≠ ∅
15     # Noden nærmest treeet
16     u = EXTRACT_MIN(Q)
17     # "Oppdag" - eller i hver fall oppdater - naboen
18     for each v ∈ G.Adj[u]
19         # Er v fortsatt ubrukt? Fant vi en bedre kant fra treeet til v?
20         if v ∈ Q and w(u,v) < v.key
21             # Da bruker vi denne nye kanten i stedet...
22             v.π = u
23             # ...og oppdaterer prioriteten til v i Q
24             v.key = w(u, v)
```

## Bellman-Ford

```
1 # Slakk alle kantene til det bare må bli rett!
2 BELLMAN_FORD(G, w, s)
3 INITIALIZE_SINGLE_SOURCE(G, s)
4 # Ingen stier har flere enn |V|-1 kanter
5 for i = 1 to |G.V|-1
6     # Gå gjennom alle |V|-1 ganger
7     for each edge (u, v) ∈ G.E
8         # Alle stier må nå ha fått kantene slakket i rekkefølge!
9         RELAX(u, v, w)
10    # Vi kjører en iterasjon til...
11    for each edge (u, v) ∈ G.E
12        # Men bare sjekker om det fortsatt finnes snarveier
13        if v.d > u.d + w(u, v)
14            # Isåfall: Vi må ha kommet borti en negativ sykel
15            return false
16    # Ellers: Svaret vi fant må være rett!
17    return true
```

```
1 RELAX(u, v, w)
2 # Vil ha avstanden(d) for til-noden(v) = avstanden for fra-noden(u)
3 # + minste vekt mellom (u,v)
4 if v.d > u.d + w(u, v)
5     v.d = u.d + w(u, v)
6     # I dette tilfellet: Hvilken forgjenger v. $\pi$  ga oss minimum, v.d?
7     # Merk: DP
8     v. $\pi$  = u
```

```
11 INITIALIZE_SINGLE_SOURCE(G, s)
12 for each vertex v ∈ V
13     # initialiserer avstanden til nodene i DAG
14     v.d =  $\infty$ 
15     # Hvilken forgjenger v. $\pi$  ga oss minimum, v.d? merk: videre DP
16     v. $\pi$  = NIL
17 # Initialiserer avstanden til startnoden
18 s.d = 0
```

## Dijkstra

```
1 # Slakk ut-kantene fra den gjenværende med lavest estimat
2 DIJKSTRA(G, w, s)
3 INITIALIZE_SINGLE_SOURCE(G, s)
4 # Ferdige: Vi har slakket ut fra dem og de vil aldri endre seg igjen
5 S = ∅
6 # Prioritetskø som PRIM, men her prioritert etter v.d
7 Q = G.V
8 # Så lenge det er noen vi ikke har avstanden til...
9 while Q = ∅
10     # Ingen negative kanter ⇒ u.d vil aldri endres mer!
11     u = EXTRACT_MIN(Q)
12     S = S ∪ {u}
13     for each vertex v ∈ G.Adj[u]
14         # u.d er rett; slakk alle ut-kanter
15         RELAX(u, v, w)

1 RELAX(u, v, w)
2 # Vil ha avstanden(d) for til-noden(v) = avstanden for fra-noden(u)
3 # + minste vekt mellom (u,v)
4 if v.d > u.d + w(u, v)
5     v.d = u.d + w(u, v)
6     # I dette tilfellet: Hvilken forgjenger v.π ga oss minimum, v.d?
7     # Merk: DP
8     v.π = u

11 INITIALIZE_SINGLE_SOURCE(G, s)
12 for each vertex v ∈ V
13     # initialiserer avstanden til nodene i DAG
14     v.d = ∞
15     # Hvilken forgjenger v.π ga oss minimum, v.d? merk: videre DP
16     v.π = NIL
17 # Initialiserer avstanden til startnoden
18 s.d = 0
```

## DAG shortest path

```
1 # Grafen er delproblemgrafen. Bruk av dynamisk programmering
2 DAG_SHORTEST_PATH(G, w, s)
3 # Alle noder v er avhengig av havner før v
4 topologically sort the vertices of G
5 INITIALIZE_SINGLE_SOURCE(G, s)
6 # "Bottom-up"-løsning
7 for each vertex u ∈ V, taken in topologically sorted order
8     # For alle kantene (u,v) i E
9     for each vertex v ∈ G.Adj[u]
10        # Må alt ha slakket inn-kanter; slakk alle ut-kanter
11        RELAX(u, v, w)
```

```
1 RELAX(u, v, w)
2 # Vil ha avstanden(d) for til-noden(v) = avstanden for fra-noden(u)
3 # + minste vekt mellom (u,v)
4 if v.d > u.d + w(u, v)
5     v.d = u.d + w(u, v)
6     # I dette tilfellet: Hvilken forgjenger v. $\pi$  ga oss minimum, v.d?
7     # Merk: DP
8     v. $\pi$  = u
```

```
11 INITIALIZE_SINGLE_SOURCE(G, s)
12 for each vertex v ∈ V
13     # initialiserer avstanden til nodene i DAG
14     v.d =  $\infty$ 
15     # Hvilken forgjenger v. $\pi$  ga oss minimum, v.d? merk: videre DP
16     v. $\pi$  = NIL
17 # Initialiserer avstanden til startnoden
18 s.d = 0
```

## Floyd-Warshall

```
1 # For hver node i og j: Hva er korteste vei i → j?
2 FLOYD_WARSHALL(W)
3 n = W.rows
4 # Korteste vei i → j som ikke går via andre = w(i, j)
5 D(0) = W
6 # Vi får nå lov til å gå innom node k
7 for k = 1 to n
8     # Hva er korteste vei i → j som kun får gå innom {1, ..., k}?
9     let D(k) = (d(k)ij) be a new n × n matrix
10    # For hver mulig startnode...
11    for i = 1 to n
12        # For hver mulig slutt-node...
13        for j = 1 to n
14            # Er det raskere å gå innom k (og ellers fortsatt
15            # kun {1, ..., k-1}
16            d(k)ij = min(d(k-1)ij, d(k-1)ik + d(k-1)kj)
17    # Korteste vei i → j som får gå innom {1, ..., n}, dvs alle
18    return D(n)
```

```
1 # Forenklet: Bruker bare D heller enn D(k)
2 FLOYD_WARSHALL*(W)
3 n = W.rows
4 # Ikke via noen: dij = w(i, j); πij = i når i ≠ j og w(i, j) < ∞
5 initialize D and Π
6 # Vi får nå lov til å gå innom en node k
7 for k = 1 to n
8     # For hver mulig startnode...
9     for i = 1 to n
10         # For hver mulig slutt-node...
11         for j = 1 to n
12             # Er det raskere å gå innom k (og ellers fortsatt
13             # kun {1, ..., k-1}?
14             if dij > dik + dkj
15                 # Oppdater avstanden
16                 dij = dik + dkj
17                 # Husk valget: Forgjengeren til i om vi kommer fra j
18                 πij = πkj
19 return D, Π
```

## Johnson

```
1 # Alle til alle. Tillater negative vekter
2 # Forelesning - Forenkling: Ignorerer returverdi fra Bellman-Ford
3 # = Antar vi ikke har negative sykler
4 JOHNSON(G, w)
5 ### Sørger for ikke-negitive kanter ###
6 # Skal brukes til forarbeidet med BELLMAN-FORD
7 construct G*, with start node s
8 # Forarbeid: Gir oss info vi trenger til h
9 BELLMAN_FORD(G*, w, s)
10 # s er overflødig, derfor G og ikke G*
11 for each vertex v ∈ G.V
12     # Her er v.d = δ(s,v), beregnet av BELLMAN_FORD
13     h(v) = v.d
14 # s er overflødig, derfor G og ikke G*
15 for each edge (u,v) ∈ G.E
16     #  $h(v) \leq h(u) + w(u,v) \Rightarrow \hat{w}(u,v) \geq 0$ 
17      $\hat{w}(u,v) = w(u,v) + h(h) - h(v)$ 
18 ### Finner korteste vei ###
19 # Resultat: Alle til alle med DIJKSTRA
20 let D = (duv be a new n × n matrix
21 # For hver u, finn δ(u, .) for  $\hat{w}$  med DIJKSTRA
22 for each vertex u ∈ G.V
23     # Nå lovlig, fordi  $\hat{w}$  er ikke-negativ
24     DIJKSTRA(G,  $\hat{w}$ , u)
25     # For hver v, finn duv = δ(u,v) for w fra δ(u,v) for  $\hat{w}$ 
26     ### Retter opp første del ###
27     for each vertex v ∈ G.V
28         # Teleskopsum med -h(x) + h(x) for x mellom u og v
29         duv =  $\hat{\delta}(u, v) + h(v) - h(u)$ 
30 # Rett fordi w(p) ≤ w(q) ↔  $\hat{w}(p) \leq \hat{w}(q)$  for stier u → v
31 return D
```

## Transitive-closure

```
1 # For hver node i og j: Finnes det en sti i → j?
2 TRANSITIVE_CLOSURE(G)
3 n = |G.V|
4 ##### Initialize T #####
5 # Finnes det en sti i → j som ikke går via andre noder?
6 let T(0) = t(0)ij be a new n × n matrix
7 # For hver mulig startnode...
8 for i = 1 to n
9     # For hver mulig sluttnode
10    for j = 1 to n
11        # Samme node eller sammenkoblet med kant?
12        if i == j or (i, j) ∈ G.E
13            # Da finnes det en sti i → j som ikke får via noen noder
14            t(0)ij = 1
15        # Ellers finnes ingen slik sti
16        else
17            t(0)ij = 0
18 ##### Slutt init: Grunntilfellet på plass ####
19 # Vi får nå lov til å gå gjennom node k
20 for k = 1 to n
21     # Finnes en sti i → j som kun får gå innom i {1, ..., k}?
22     let D(k) = (d(k)ij) be a new n × n matrix
23     # For hver mulig startnode...
24     for i = 1 to n
25         # For hver mulig sluttnode
26         for j = 1 to n
27             # Finnes i → j eller i → k → j, om vi kun får gå
28             # gjennom {1, ..., k-1}?
29             t(k)ij = t(k-1)ij ∨ (t(k-1)ik ∧ t(k-1)kj)
30 # Finnes det en sti i → j som får gå innom {1, ..., n}, dvs alle?
31 return T(n)
```

```
1 # Forenklet utgave: Bruker bare en tabell, T
2 # For hver node i og j: Finnes det en sti i → j?
3 TRANSITIVE_CLOSURE*(G)
4 n = |G.V|
5 # Samme node (I er identitetsmatrise) eller nabo, (A er nabomatrise)
6 initialize T
7 # Vi får nå lov til å gå innom node k
8 for k = 1 to n
9     # For hver mulig startnode...
10    for i = 1 to n
11        # For hver mulig sluttnode...
12        for j = 1 to n
13            # Enten allerede en sti i → j eller en ny sti i → k → j?
14            tij = tij ∨ (tik ∧ tkj)
15 return T(n)
```

## Ford-Fulkerson

```
1 # Finn maksinal flyt fra s til t i G
2 FORD_FULKERSON(G, s, t)
3 # Initialiser flyt f til 0
4 for each edge (u, v) ∈ G.E
5     (u, v).f = 0
6 # Ikke nødvendigvis en sti i G; kanter kan gå baklengs i p der
7 # Alle kanter i  $G_f$  har restkapasitet. En sti  $s \rightarrow t$  er dermed forsøkende
8 while there is a path p from s to t in  $G_f$ 
9     # Dette er "flaskehalsen" langs p: minste restkapasitet
10     $c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is in } p\}$ 
11    # Vi vil øke flyten langs p med denne flaskehals-restkapasiteten
12    for each edge (u, v) in p
13        # Foroverkant eller restkapasitet?
14        if (u, v) ∈ E
15            # Da kan flyten økes langs kanten
16            (u, v).f = (u, v).f +  $c_f(p)$ 
17        else
18            # Baklengs: Opphev flyt; omdirigeres implisitt til neste
19            # kant i p
20            (v, u).f = (v, u).f -  $c_f(p)$ 
```

```
1 # Forenklet versjon
2 # Finn maksimal flyt fra s til t i G
3 FORD_FULKERSON_METHOD(G, s, t)
4 # Lovlig, men neppe en optimal løsning. Forbedres gradevis
5 initialize flow f to 0
6 # Ikke nødvendigvis en sti i G; kanter kan gå baklengs i p der
7 while there is an augm. path p in  $G_f$ 
8     Flyten i hver kant kan økes, eller oppheves og omdirigeres
9     augment flow f along p
10 return f
```

## Edmonds-Karp

```
1 # Ford Fulkerson med BFS: Finner forøkende sti
2 EDMONDS_KARP(G, s, t)
3 ##### INIT #####
4 for each edge (u, v) ∈ G.E
5     (u, v).f = 0
6 # Gjenta til vi ikke får mer flyt frem til t
7 repeat #until t.a == 0
8     ##### INIT #####
9     # I hver iterasjon...
10    for each vertex u ∈ G.V
11        # Hvor mye mer flyt klarer vi å få frem til u?
12        u.a = 0 (reaching u in  $G_f$ )
13        # Dette er forgjengeren i BFS-treeet
14        u.π = NIL
15    # Alltid uendelig mye mer flyt hos s. Vi vil sende den videre
16    s.a = ∞
17    # FIFO-kø til bredde-først-søk i  $G_f$ 
18    Q = ∅
19    # Vi traverserer fra s, og leter etter t
20    ENQUEUE(Q, s)
21    ##### Finner forøkende sti #####
22    # Her begynner BFS. Avbrytes hvis vi finner t
23    while t.a == 0 and Q ≠ ∅
24        # Neste node vi skal besøke
25        u = DEQUEUE(Q)
26        # For alle potensielle naboer i  $G_f$ : Se etter positiv
27        # restkapasitet
28        for all edges (u,v), (v,u) ∈ G.E
29            # Fremoverkant?
30            if (u,v) ∈ G.E
31                # Restkapasitet = mulig økning av flyt langs kanten
32                 $c_f(u,v) = c(u,v) - (u,v).f$ 
33            # Bakoverkant: Restkapasitet = mulig
34            # oppheving/omdirigering
35            else
36                 $c_f(u,v) = (u,v).f$ 
37            # Restkapasitet ↔ kant i  $G_f$ . Ingen flytøkning ↔ ubesøkt
38            if  $c_f(u,v) > 0$  and v.a == 0
39                # Vi har med oss u.a ekstra flyt; får maks  $c_f(p)$ 
40                # gjennom (u, v)
41                v.a = min(u.a,  $c_f(u,v)$ )
42                # Hvor kom vi fra da vi oppdaget v?
```

```

43     v. $\pi$  = u
44     # Så vi husker å traversere v senere, og sende flyt
45     # videre fra den
46     ENQUEUE(Q, v)
47     # Har nå fått frem flyt til t eller gått tom for noder
48     # Vi har fått frem t.a ekstra flyt, kun begrenset av
49     # flaskehalsen  $c_f(p)$ 
50     u, v = t. $\pi$ , t (merk: t.a =  $c_f(p)$ )
51     # Tilbakesporing: Følge  $\pi$ -feltene tilbake fra t til s
52     while u  $\neq$  NIL
53         # Fremoverkant?
54         if (u, v)  $\in$  G.E
55             # Øk flyten ved t.a dvs  $c_f(p)$ 
56             (u, v).f = (u, v).f + t.a
57             # Bakoverkant: Opphev og omdiriger t.a, dvs  $c_f(p)$ 
58             else
59                 (u, v).f = (u, v).f - t.a
60             # Gå ett skritt bakover langs p
61             u, v = u. $\pi$ , u
62     # Ingen forøkende sti p funnet; flyten er maksimal
63 until t.a == 0

```