# Algoritmer og datastrukturer
# TDT4120
# Pensumoversikt

Håkon Ødegård Løvdal

Sist endret: 23. oktober 2014

# 1 Intro

Dette dokumentet er skrevet for min egen del, i desperasjon for å tilegne meg pensum i TDT4120 høsten 2013. All informasjon er hentet fra Cormen og tilfeldige skumle steder på internett. All bruk og lesing skjer på eget ansvar. Du får ignorere eventuelle skrivefeil da det er raskt skrevet, uten korrekturlesning. Tex-fila er tilgjengelig på GitHub og OnlineWikien, så du er velkommen til å endre og modifisere som du selv vil!

# 2 Begreper

| | |
|---|---|
| Algoritme | En veldefinert prosedyre, som tar en verdi eller mengde verdier som input, og produserer en verdi, eller mengde med verdier som output. Det er en sekvens som transformerer input til output. |
| Problem | Et problem er en relasjon mellom input og output. |
| Probleminstans | En probleminstans er en bestemt input. |
| Iterasjon | En gjennomkjøring av en gjenntatt handling |
| Asymtpoisk notasjon | Notasjon hvor man kun tar vare på største ledd |
| In-Place | En algoritme er in-place når den operer på input dataen uten å måtte 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. |

# 3 Asymptotisk notasjon

| Navn | Notasjon | Kjøretid |
|---|---|---|
| Store-O | $O(n)$ | $\leq n$ |
| Theta | $\Theta(n)$ | $= n$ |
| Store-Omega | $\Omega(n)$ | $\geq n$ |

## 3.1 Kompleksitetsklasser

Grovinndeling i stigende rekkefølge:

1. Konstant: 1
2. Polylogaritmisk: $(\log_b n)^k$
3. Polynomisk: $n^k$
4. Eksponentiell: $2^n$
5. Faktoriell: $n!$
6. Alt som er enda verre, f.eks. $n^n$

### 3.2 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 $= \lg(n)$.

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

Dukker ofte opp som lureoppgave på eksamen!

## 4 Kjøretider på pensumalgoritmer

| **Problem** | **Algoritme** | **Kjøretid** | | |
|---|---|---|---|---|
| | | **BC** | **AC** | **WC** |
| Sortering | Insertion Sort | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| | Bubble Sort | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| | Merge Sort | $\Theta(n \lg(n))$ | $\Theta(n \lg(n))$ | $O(n \lg(n))$ |
| | Heap Sort | $O(n \lg(n))$ | $O(n \lg(n))$ | $O(n \lg(n))$ |
| | Quick Sort | $\Theta(n \lg(n))$ | $O(n \lg(n))$ | $\Theta(n^2)$ |
| | Counting Sort | - | $\Theta(n)$ | $\Theta(k+n)$ hvis $k = O(n)$ |
| | Radix Sort | - | $\Theta(d(n+k))$ | - |
| | Bucket Sort | - | $\Theta(n)$ | - |
| Grafer/Trær | Toplogisk sortering | - | $\Theta(\|V\| + \|E\|)$ | - |
| | DFS | - | $\Theta(\|V\| + \|E\|)$ | - |
| | BFS | - | $O(\|V\| + \|E\|)$ | - |
| | Prim | $O(E \lg(V)$ | $\Theta(E \lg(V))$ | - |
| | Kruskal | $O(E \lg(V)$ | $\Theta(E \lg(V))$ | - |
| | Binærsøk | $O(\lg n)$ | - | $O(\lg n)$ |
| Korteste vei | Bellman-Ford | - | $O(\|V\| \times \|E\|)$ | - |
| | Dijkstra | $O(\|E\| \lg(\|V\|)$ (bin-heap) | $O(\|V\|^2)$ (array) | - |
| | DAG-Shortest Path | - | $\Theta(\|V\| + \|E\|)$ | - |
| | Floyd-Warshall | - | $\Theta(\|V\|^3)$ | - |
| Flyt | Ford-Fulkerson | - | $O(E\|f^*\|)$ | $f^*$ = maks flyt i G |
| | Edmonds-Karp | - | $O(VE^2)$ | - |
| Grådig | Huffmann | $O(n \lg(n))$ | - | - |

## 5 Rekurensanalyse

### 5.1 Masterteoremet

Masterteoremet er en form for "kokebok"-metode for å løse rekurenser. Man kan som regel løse de fleste rekurrenser av typen:

$$T(n) = aT(\tfrac{n}{b}) + f(n)$$

hvor $a \geq 1$ og $b > 1$. $f(n)$ må også være asymptotisk positiv.

**Case 1:** $f(n) = O(n^{\log_b a - \varepsilon})$ , for en $\varepsilon > 0$
$\Rightarrow T(n) = \Theta(n^{\log_b a})$

**Case 2:** $f(n) = \Theta(n^{\log_b a})$
$\Rightarrow T(n) = \Theta(n^{\log_b a} \log(n))$

**Case 3:** $f(n) = \Omega(n^{\log_b a+\varepsilon})$, for en $\varepsilon > 0$ og $af(\frac{n}{b}) \leq cf(n)$, hvor $c < 1$
$\Rightarrow T(n) = \Theta(f(n))$

### 5.1.1 Bruk av masterteoremet

Gitt et problem i hver "case":

1. $T(n) = 4T(\frac{n}{2}) + n$
2. $T(n) = 4T(\frac{n}{2}) + n^2$
3. $T(n) = 4T(\frac{n}{2}) + n^3$

I disse problemene er $a = 4, b = 2$ og $f(n)$ henholdsvis $n, n^2, n^3$. I alle disse tilfellene vil vi sammenligne $f(n)$ med $n^{\log_b a} \Rightarrow n^{\log_2 4} \Rightarrow n^2$.

**Case 1:** $f(n) = n = O(n^{2-\varepsilon})$ for $\varepsilon = 0.5$. Altså $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

**Case 2:** $f(n) = n^2 = \Theta(n^2)$. Altså $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^2 \log n)$

**Case 3:** $f(n) = n^3 = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 0.5$ og $af(\frac{n}{b}) \leq cf(n)$, feks. $4(\frac{n}{2})^3 = \frac{n^3}{2} \leq cn^3$ for $c = \frac{1}{2}$. Altså $T(n) = \Theta(f(n)) = \Theta(n^3)$

## 5.2 Eksempel på rekursens fra desember 2007

| Rekursive kall | Størrelse, delproblem | Arbeid i hvert kall | Rekurrens | Kjøretid |
|---|---|---|---|---|
| Ett | Redusert med 1 | Konstant | $T(n) = T(n-1) + \Theta(1)$ | $\Theta(n)$ |
| Ett | Halvert | Konstant | $T(n) = T(\frac{n}{2}) + \Theta(1)$ | $\Theta(\lg n)$ |
| Ett | Redusert med 1 | Lineært | $T(n) = T(n-1) + \Theta(n)$ | $\Theta(n^2)$ |
| Ett | Halvert | Lineært | $T(n) = T(\frac{n}{2}) + \Theta(n)$ | $\Theta(n)$ |
| To | Redusert med 1 | Konstant | $T(n) = 2T(n-1) + \Theta(1)$ | $\Theta(2^n)$ |
| To | Halvert | Konstant | $T(n) = 2T(\frac{n}{2}) + \Theta(1)$ | $\Theta(n)$ |
| To | Redusert med 1 | Lineært | $T(n) = 2T(n-1) + \Theta(n)$ | $\Theta(2^n)$ |
| To | Halvert | Lineært | $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ | $\Theta(n \lg n)$ |

Merk at siste rekursen er løst med masterteoremet, mens de andre er løst uten.

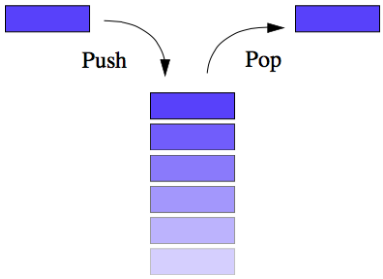
$$\text{tid} = \#\text{ subproblemer} \times \text{tid per subproblem}$$

# 6 Datastrukturer

| | Lenket liste | Array | Dynamisk Array |
|---|---|---|---|
| Indeksering | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |
| Ins/del front | $\Theta(1)$ | - | $\Theta(n)$ |
| Ins/del bakerst | $\Theta(n)$ | - | $\Theta(1)$ |
| Ins/del midten | Søk + $\Theta(1)$ | - | $\Theta(n)$ |
| Ubrukt plass | $\Theta(n)$ | 0 | $\Theta(n)$ |

## 6.1 Stack

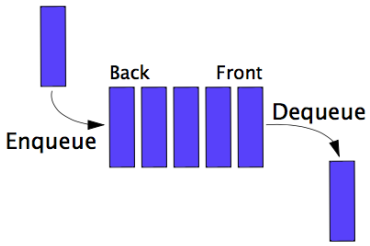
Stack er en *LIFO*, "Last In, First Out"-datastruktur. Stack implementeres oftest med et array, eller en lenket liste. En stack har egenskapene *INSERT, PUSH* og *POP (DELETE)*.




Figur 1: Enkel illustrasjon av en stack

## 6.2 Kø/Queue

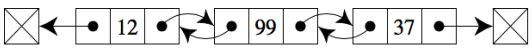
Kø er en *FIFO*, "Fist In, First Out"-datastruktur. En kø implementeres som oftest som et array (gjerne et dynamisk array, likt Java sin *ArrayList*. En kø har egenskapene *ENQUEUE* og *DEQUEUE*.




Figur 2: Enkel illustrasjon av en kø

## 6.3 Lenket-liste

En lenket liste er en datastruktur hvor objektene er ordnet lineært. Hvert objekt har helst to peker-variabler, *next* og *prev*. Har den begge disse peker-variablene er det en dobbelt lenket liste. Ved kun en peker (*next*) er det en enkelt-lenket liste.




Figur 3: Enkel illustrasjon av en dobbelt lenket liste

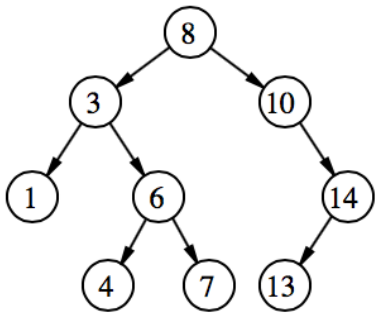
## 6.4 Binært søketre

Et binært søketre er et tre som tilfredstiller *binært-søketre-egenskapen.*

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

### 6.4.1 Binært-søketre-egenskapen

La $x$ være en gitt node i søketreet. Hvis $y$ er en node i det venstre subtreet til $x$ må $y$ sin verdi være mindre eller lik ($\leq$) $x$ sin verdi.
Tilsvarende for høyre subtre. Her må $y$ sin verdi være større enn eller lik ($\geq$) $x$ sin verdi.




Figur 4: Et binært søketre

### 6.4.2 Traversering av binæretrær

Vi benytter oss av tre måter å traversere et binært tre på, disse er:

**Preorder** Her printer man ut nodens verdi før dens barn, venstre og deretter høyre.

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

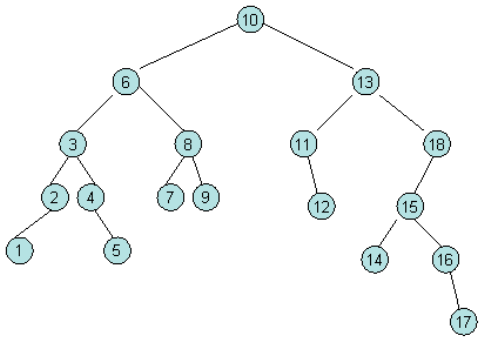
**Postorder** Her printer man nodens verdi etter man har printet venstre og høyre barn

Et eksempel fra figur 5:

**Preorder** 10, 6, 3, 2, 1, 4, 5, 8, 7, 9, 13, 11, 12, 18, 15, 14, 16, 17

**Inorder** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18

**Postorder** 1, 2, 5, 4, 3, 7, 9, 8, 6, 12, 11, 14, 17, 16, 15, 18, 13, 10




Figur 5: Enkelt binærtre for traverseringseksempel

## 6.5 Hasing

Hashing er en effektiv måte å lagre $(key, value)$-pairs. Istedenfor at indeksen er nøkkelen, er heller indeksen basert på nøkkelen med en hashfunksjon.

| Operasjon | WC | BC |
|---|---|---|
| Søke | $O(1)$ | $O(n)$ |
| Sett inn | $O(1)$ | $O(n)$ |
| Slette | $O(1)$ | $O(n)$ |
| Plass | $O(n)$ | $O(n)$ |

*Kommentar: Usortert array*

### 6.5.1 Direct-address tables

Fungerer best/optimalt når mengen mulige nøkler ($n$) er forholdsvis lav. Ved å lage en tabell med $n$ felter, kan hver indeks representere en nøkkel.

### 6.5.2 Hash-tables

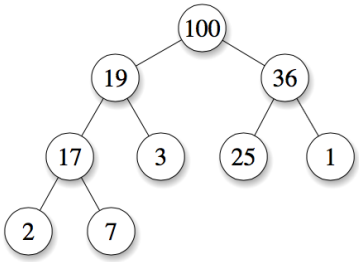
En mer effektiv måte å lage en slik tabell på. Man har en mindre tabell enn antall elementer. Hvert felt har derimot en egen nøkkel. Disse nøkklene blir generert av hashfunksjonen, $h(k)$. På grunn av at tabellen er mindre enn antall elementer kan kollisjoner oppstå (flere nøkler, gir samme hashverdi). Vi har flere metoder for å løse kollisjoner. Her brukes feks. ***chaining***.

## 6.6 Heap

En heap er en spesiell trestruktur, som tilfredstiller heap-egenskapen. Det finnes ingen regler for hvordan søsken er ordnet i heapen. En heap er ofte implementert med et array. Det er viktig å legge merge til at første indeks skal være tom. Det vil si at i et 0-indeksert array vil plass 0 være tom, mens første node står på indeks 1.

| Operasjon | Tidskompleksitet |
|---|---|
| Finn min/max | $\Theta(1)$ |
| Slett min/max | $\Theta(\log n)$ |
| Sett inn | $\Theta(\log n)$ |
| Omstrukturere heap | $\Theta(\log n)$ |
| Merge | $\Theta(n)$ |

*Kommentar: Min/max avhenger om det er min/max heap*




Figur 6: En liten max-heap

### 6.6.1 Heap-egenskapen

I en heap må hele treet være fullstendig fylt ut, bortsett fra muligens det laveste nivået, som fylles ut fra venstre.

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

**Max-Heap:**

For hver node $i$, bortsett fra rot-noden, må verdien til barnenode være mindre eller lik foreldrenoden. $A[PARENT(i)] \geq A[i]$

**Min-Heap:**

For hver node $i$, bortsett fra rot-noden, må verdien til en barnenode være større eller lik foreldre noden. $A[PARENT(i)] \leq A[i]$

# 7 Sorteringsalgoritmer

## 7.1 Insertion Sort

Insertion Sort er en enkel sorteringsalgoritme. Den tar de to første elementene og plasserer i forhold til hverandre. Deretter plasserer den det neste i forhold til de to forrige, osv. Veldig effektiv å bruke på små mengder. Kan feks. brukes i slutten på en Quick Sort algoritme.

```
for j = 0 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i − 1
    end while
    A[i] + 1 = key
end for
```

## 7.2 Bubble Sort

Bubble Sort testet to og to naboelementer. Dersom den første er større bytter de plass. Effektiv på små datamengder.

```
for i = 0 to A.length − 1 do
    for j = A.length downto i + 1 do
        if A[j] < A[j − 1] then
            exchange A[j] with A[j − 1]
        end if
    end for
end for
```

## 7.3 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.

```
function MERGE-SORT(A, p, r)
    if p < r then
        q = ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
    end if
end function

function MERGE(A, p, q, r)
    n1 = q − pr + 1
    n2 = r − q
    let L[1 . . . n1 + 1] and R[1 . . . n2 + 1 be new arrays
    for i = 1 to n1 do
        L[i] = A[p + i − 1]
    end for
    for j = 1 to n2 do
        R[j] = A[q + j]
    end for
    L[n1] + 1] = ∞
    r[n2] + 1] = ∞
```

```
    i = 1
    j = 1
    for k = p to r do
        if L[i] ≤ R[j] then
            A[k] = L[i]
            i = i + 1
        else
            if A[k] = R[j] then
                j = j + 1
            end if
        end if
    end for
end function
```

## 7.4 Heap Sort

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

```
BUILDMAXHEAP(A)
for i = A.length downto 2 do
    exchange A[1] with A[i]
    A.heapSize = A.heapSize − 1
    MAX-HEAPIFY(A, 1)
end for
```

## 7.5 Quick Sort

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 en pivot. Deretter kaller den seg selv rekursivt på de to partisjonene. Deretter fletter man sammen de to partisjonene.

```
function QUICKSORT(A, p, r)
    if p < r then
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A.q + 1, r)
    end if
end function

function PARTITION(A, p, r)
    x = A[r]
    i = p − 1
    for j = p to r − 1 do
        if A[j] ≤ x then
            i = i + 1
```

```
            exchange A[i] with A[j]
        end if
    end for
    exchange A[i + 1] with A[j]
    return i + 1
end function
```

## 7.6 Counting Sort

Counting Sort tar et heltall $N$ mellom 0 og $k$. Lager en liste med vardier fra 0 til $k$ og setter inn tallene på sin plass i lista. Fungerer best når verdiene på tallene som sorteres ligger tett etterhverandre og $k$ ikke er for høy.

```
function COUNTINGSORT(A, B, k)
    let C[0 . . . k] be a new array
    for i = 0 to k do
        C[i] = 0
    end for
    for j = 1 to A.length do
        C[A[j]] = C[A[j]] + 1
    end for
    for i = 0 to k do
        C[i] = C[i] + C[i − 1]
    end for
    for j = A.length downto 1 do
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] − 1
    end for
end function
```

## 7.7 Radix Sort

Radix Sort sorterer tall i et gitt tallsystem (her titallsystemet) etter minst signifikante siffer.

```
function RADIXSORT(A, d)
    for i = 1 to d do
        Bruk Stable Sort til å sortere array A
    end for
end function
```

## 7.8 Bucket Sort

Veldig lik Counting Sort, men bruker såkalte "bøtter" man putter tallene i. For eksempel $[5, 6)$ betur at verdier større eller lik 5, men mindre enn 4 skal i bøtta.

```
function BUCKETSORT(A)
    n = A.length
    la B[0 . . . n − 1] være et nytt array
```

```
    for i = 0 to n − 1 do
        Gjør B[i] til en tom liste
    end for
    for i = 1 to n do
        Sett inn A[i] i lista B[⌊nA[i]⌋]
    end for
    for i = 0 to n − 1 do
        Sorter lista B[i] med InsertionSort(B[i])
    end for
    Konkatiner listene B[0], B[1], ..., B[n − 1] i rekkefølge
end function
```

# 8 Grafer/Trær

## 8.1 Dybde først-søk

DFS implementeres med en stack. DFS utforsker grafen i dybden. Den fyller stacken med noder den støter på. Kan man ikke går 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.

```
function DFS(G, v)                                  ▷ v er startnode
    initaliser en tom stack, S
    for each vertex u in G do
        set visited[u] → false
    end for
    S.push(v)
    while S.notEmpty() do
        u = S.pop()
        for all w adjacent to u do
            if not visited[w] then
                visited[w] → true
                S.push(w)
            end if
        end for
    end while
end function
```

## 8.2 Bredde først-søk

BFS implementeres med en kø. BFS utforsker grafen i bredden. Man starter på foreldrenoden og legger inn alle dens barn i køen. Når 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. Når køen er tom, sjekker man ikke videre om det er ubesøkte noder.

```
function BFS(G, v)                                  ▷ v er startnode
    lag en kø Q
    legg v inn i Q
    while Q.notEmpty() do
        v = Q.dequeue()
```

```
        for each edge e adjacent to v do
            if e not marked then
                mark w
                Q.enqueue(e)
            end if
        end for
    end while
end function
```

## 8.3 Topologisk sortering

Topologisk sortering bruker en DAG til å finne en rekkefølge igjennom alle elementene i grafen. Det tillates ikke sykler. Man begynner i noden som ikke har noen kanter inn til seg.

```
  function TOPOLOGICALSORT(G)
      DFS(G) to compute finish times v.f for each vertex v.
      as each vertex is finished, insert it onto the end of the list
      return the list of verticies
  end function
```

## 8.4 Minimale spenntrær

### 8.4.1 Prims algoritmer

Prims algoritme velger en tilfeldig node, og legger alle kantene inn i en prioritetskø etter vekt. Velger den billigste kanten og legger den til i treet. Det vil si at den legger til den billigste kanten som er mulig å legge til fra treet den bygger.

```
function PRIM(G, w, r)
    for each u ∈ G.V do
        u.key = ∞
        u.π = NIL
    end for
    r.key = 0
    Q = G.V
    while Q ≠ ∅ do
        u = EXTRACT-MIN(Q)
        for each v ∈ G.Adj[u] do
            if v ∈ Q and w(u, v) < v.key then
                v.π = u
                v.key = w(u, v)
            end if
        end for
    end while
end function
```

### 8.4.2 Kruskals algoritme

Kruskals algoritme sorterer alle kanter etter kostnaden og velger den billigste tilgjengelige kanten , og legger til denne med noder i treet. Dette kun hvis

den ikke allerede er brukt og vil danne en sykel. Fortsetter til det ikke finnes kanter som kan legges til i treet.

**function** KRUSKAL(G, w)
  $A = \emptyset$
  **for** each vertex $v \in G.V$ **do**
    MAKE-SET($v$)
  **end for**
  sort edges of $G.E$ into nondecreasing order by weight $w$
  **for** each edge $(u, v) \in G.E$, taken in nondecreasing order by weight **do**
    **if** FIND-SET($u$) $\neq$ FIND-SET($v$) **then**
      $A = A \cup \{(u, v)\}$
      UNION(u,v)
    **end if**
  **end for**
  **return** $A$
**end function**

# 9 Korteste-vei-algoritmer

Det er flere gode algoritmer for å løse korteste-vei problemet.

**function** RELAX(u, v, w)
  **if** $v.d > u.d + w(u, v)$ **then**
    $v.d = u.d + w(u, v)$
    $v.\pi = u$
    Cormen bruker $\pi$ til å representere foreldrenode.
  **end if**

## 9.1 Bellman-Ford

Bellman-Ford er en korteste vei, en-til-alle algoritme. Den tillater negative kanter. Returnerer $false$ dersom det finnes en negativ sykel i grafen. Går igjennom alle kantene og bruker $RELAX$ på hver av dem $|V| - 1$ 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.

**function** BELLMANN-FORD($G, w, s$)
  INITIALIZE-SINGLE-SOURCE($G, s$)
  **for** $i = 1$ to $|G.V| - 1$ **do**
    **for** each edge $(u, v) \in G.E$ **do**
      RELAX($u, v, w$)
    **end for**
  **end for**
  **for** each edge $(u, v) \in G.E$ **do**
    **if** $v.d > u.d + w(u, v)$ **then**
      **return** $false$
    **end if**
  **end for**

```
    return true
end function
```

## 9.2 Dijkstra

Dijkstra er en korteste vei, en-til-alle algoritme. Tillater ikke negative kanter. Den velger noder en etter en fra hvor nærme de er startnoden. Dijkstra er en grådig algoritme.

```
function DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = ∅
    Q = G.V
    while Q ≠ ∅ do
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u] do
            RELAX(u, v, w)
        end for
    end while
end function
```

## 9.3 DAG-Shortest path

DAG-Shortest-Path er en korteste vei, en-til-alle algoritme. Tillater ikke negative kanter, og kan selvfølgelig ikke ha sykler, da det er en DAG. Gjør topologisk sortering av DAGen og besøker hver node *en* gang for å kjøre $RELAX$ på nodene foran.

```
function DAG-SHORTEST-PATH(G, w, s)
    TOPOLOGICAL-SORT(G)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for each vertex u, taken in topologically sorted order do
        for each vertex v ∈ G.Adj[u] do
            RELAX(u, v, w)
        end for
    end for
end function
```

## 9.4 Floyd-Warshall

Floyd-Warshall er en korteste vei, alle-til-alle algoritme. Den bruker DP. Lager en nabomatrise for alle noder og hvor det går 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.

```
function FLOYD-WARSHALL(W)
```

```
    D^(0) = W
    for k = 1 to n do
        let D^(k) be a n x n matrix
        for i = 1 to n do
            for j = 1 to n do
                D_ij^(k) = min(D_ij^(k-1), D_ik^(k-1) + D_kj^(k-1))
            end for
        end for
    end for
    return D^(n)
end function
```

# 10 Flyt-algoritmer

## 10.1 Ford-Fulkerson metoden

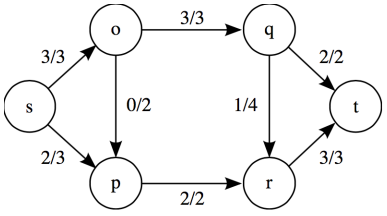
Ford-Fulkerson metoden finner maksimal flyt i et flytnettverk. Hver iterasjon forsøker å finne en flytforøkende sti, og setter på all den flyten som er mulig. Deretter leter den etter en ny flytforøkende sti, og gjentar prosessen. Når det ikke er flere flytforøkende stier har man oppnåd maksimal flyt. Den benytter seg av DFS for å finne flytforøkende sti.

## 10.2 Edmonds-Karp Algoritmen

Endret en bokstav på Ford-Fulkerson metoden. De benytter seg av BFS. Edmonds-Karp bruker Ford-Fulkerson og BFS til å finne flytforøkende stier.

## 10.3 Max Flow/Min Cut

Minimum-snitt (min-cut) pa et flytnettverk: det snittet som har lavest kapasitet av alle snitt. Det vil si *min-cut* angir en flaskehals i flytnettverket. Det vil si at det ikke kan sendes mer flyt gjennom nettverket enn det vi kan sende gjennom flaskehalsen. Man kan da ikke finne noen flytforøkende sti over flaskehalsen. Det vil da være maksimal-flyt, max-flow.



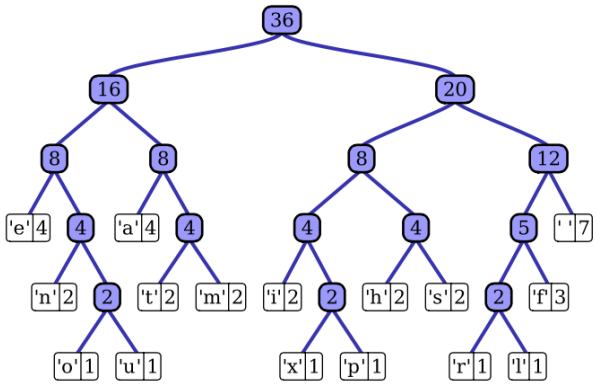

Figur 7: Et lite flytnettverk

# 11 Grådighet

En grådig algoritme er en algoritme som velger den beste lokale løsningen. Det vil si at den velger en lokalt optimalt løsning i håp om at den også skal være globalt optimal.

## 11.1 Huffmann

Huffman-koding er en komprimeringsalgoritme. Den bruker et binærtre som baserer seg på frekvensen til bokstaver til å finne en prefikskode med ulik bitlengde for alle bokstavene.




Figur 8: Et lite Huffmanntre

# 12 Dynamisk Programmering

DP $\approx$ rekursjon + memoisering $\Rightarrow$ $tid/subproblemer = \Theta(1)$ (teller ikke rekursjoner)

### 12.0.1 Memoiserings 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.

### 12.0.2 Bottom-Up DP-algoritme

Veldig lik memoiserings algoritmen, men lagrer verdiene i en hashmap/dictionary og går fra 1 til $n$. Et eksempel er en implementasjon av *Fibonacci* med $for - loop$ hvor man går fra 1 til $n$. For å spare plass kan man lagre kun de to siste elementene, da det er de som er interessante for å regne ut neste fibonacci-tall.

### 12.0.3 Fem enkle steg:

1. definer subproblemene (#)
2. gjett (del av løsning) (#)
3. rekurens (*tid/subproblemer*)
4. *rekursivt + memoisering* $\vee$ *bottom-up*
   (a) sjekk om den er asyklisk $\vee$ toplogisk sortert
   (b) tid = # subproblemer $\times$ tid per subproblem
5. løs det orginale problemet

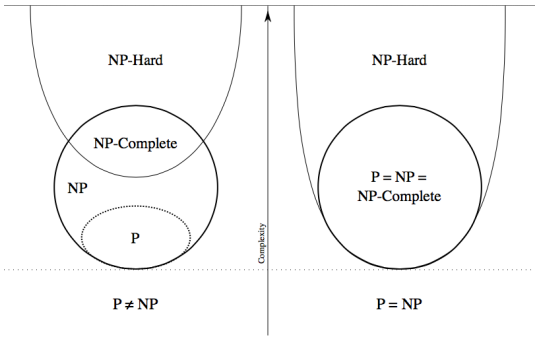
# 13 NP-Komplette problemer

NP $\Rightarrow$ Nondeterministic Polynomial
NPC $\Rightarrow$ NP-Complete
P $\Rightarrow$ Polynomial

Et problem er NPC hvis det er et "decisionproblem"og NP-Hard hvis det er et "opptimaliseringsproblem".

**Sammenhengen mellom alle NP-problemer:**
$P \subseteq NP$
$NPC \subseteq NP$

Hvis et problem $A \in NPC$, så 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 $P \cap NPC = \emptyset$.




Figur 9: Venndiagramet, med med $P \neq NP$ og $P = NP$

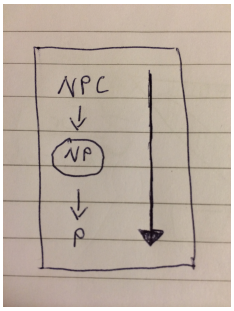
Du vet at problem A er i NP og problem B er i NPC. Du vil vise at A også er i NPC. Da reduserer du fra B til A.

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

To ikke helt legitime, men greie huskeregler:

*reduser fra litt til mer // fra litt ... $\leq$ ... til mer*
*reduser nedover: NPC $\rightarrow$ (NP) $\rightarrow$ P (forsiktig med denne)*




Figur 10: En tvilsom huskeregel, NB!

**Liste/strukturen til NPC-problemer redusert fra CIRCUT-SAT:**

- CIRCUT-SAT
- SAT
- 3-CNF-SAT
  - SUBSET-SUM
  - CLIQUE
    - VERTEX-COVER
    - HAM-CYCLE
    - TSP (Traveling Salesman Problem)

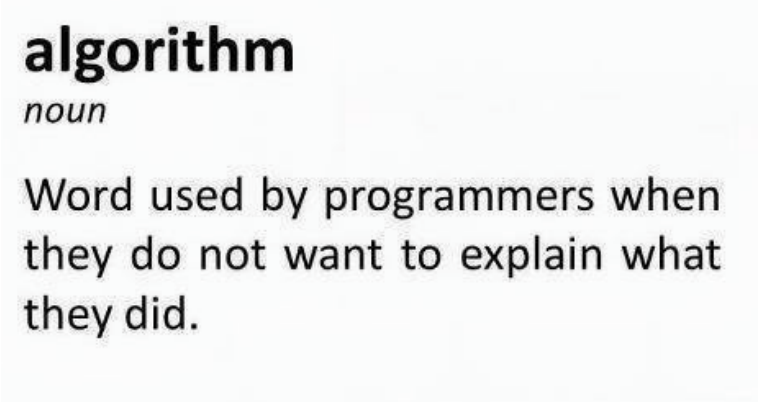
# 14 Pugg disse her for sikkerhetsskyld

Floyd-Warshall rekurrensen:

$$D_{ij}^{(k)} = min(\mathrm{D}_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)})$$

0-1 Knapsack-rekurrensen:

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




Figur 11: En avsluttende gave