markdown.md

# Velkommen til TDT4120 - Algoritmer og datastrukturer

Her er alle mine øvinger og noen av notatene mine fra faget.

## Forelesningsplan

Med notater fra forelesninger og eksamensperiode

### Forelesning 1 - Problemer og algoritmer

- [x] Kunne definere problem, instans og problemstørrelse
  - Problem:
    - Relasjon mellom input og output.
  - Instans av et problem:
    - En sub-klasse av problemet.
    - En bestemt input.
  - Problemstørrelse:
    - Lagringsplass som trengs for en instans.
- [x] Forstå løkkeinvarianter og naturlig induksjon
  - Løkkeinvarianter
    - Brukes til bevis for løkker.
    - Init: Før start.
    - Vedlikehold: Holder den før/etter en iterasjon.
    - Terminering: Løkken sier noe nyttig.
- [x] Forstå bokas pseudokode-konvensjoner
  - En måte å spesifisere algoritmer på, uavhengig av programmeringsspråk.
- [x] Kjenne egenskapene til random-access machine-modellen
- [x] Forstå Insertion-Sort
- [x] Kunne definere best-case, average-case og worst-case
  - Kjøretid: Funksjon av problemstørrelse -> f(problemstørrelse)
  - Best-case: Beste mulige kjøretid for en gitt størrelse.
  - Average-case: Forventet, gitt en sannsynlighetsfordeling.
  - Worst-case: Verste mulige. **Brukes mest**
- [x] Forstå ideen bak divide-or-conquer
  - Splitt og hersk: Del opp i mindre problemer, så få kontroll.
- [x] Forstå Merge-Sort
  - Kjøretid: Θ(n lg(n)) -> Både best, avg. og worst!
  - Tar å deler opp i 2 for hver bit. Så når hver bit er 2 i størrelse (eller 1)
    - Så sorterer man de små bitene. Så {7,2} => {2,7}. Trengs bare en gang.
    - Merger de 2 og to, så det går fra f.eks. 8 til 4 biter, så 4 til 2.
      - De sorteres når de merges.
- [x] Kunne definere asymptotisk notasjon, O, Ω, Θ, o og ω.
  - Huskeregel:

```
ω > 0(f(n)) (Small Omega)
Ω ≧ 0(f(n)) (Big Omega)
0 = 0(f(n)) (Big Theta)
O ≦ 0(f(n)) (Big O)
o < 0(f(n)) (Small o)
```

  - Kompleksitetsregler:

```
1 < ln(n) < n < nᵏ < kⁿ < n! < nⁿ | k = en konstant
```

## Forelesning 2 - Datastrukturer

- [x] Forstå hvordan stakker og køer fungerer
  - Stakker:
    - Som elementer i en boks. Øverste først inn, og elementer kan bare legges øverst.
    - Push: Legge til element på toppen.
    - Pop: Fjerne øverste elementet.
  - Køer:
    - Putter elementer inn og henter ut elementer i andre enden.
    - Enqueue: Legge til element i starten.
    - Dequeue: Hente ut element fra slutten.
- [x] Forstå hvordan lenkede lister fungerer
  - Elementer som har elementer i seg. (Peker til neste element)
  - `x = (1,(2,(3,...))) => x[1][1][0] = 3`
- [x] Forstå hvordan pekere og objekter kan implementeres
  - Omtrent som lenkede lister. Peker mot en addresse i minnet som om det var et annet objekt i den addressen.
- [x] Forstå hvordan rotfaste trær kan implementeres
  - Binære (binær = 2):
    - Tre med maks 2 armer per gren.
    - Kan ha 2 pekere til hver sin grennode.
  - Generelt om trær:
    - En toppnode med masse barn som har barn osv.
    - En node kan ikke nå et barnebarn.
- [x] Forstå hvordan direkte adressering og hashtabeller fungerer
  - Direkte adressering:
    - Bruke nøkkel direkte som indeks. Triviell form for hashing.
  - Hashtabeller:
    - Genererer en indeks fra nøkkelverdien.
    - Enveisfunksjon! Går ikke tilbake.
    - Lengde kan være en funksjon. Da er:

```
1 = 1
2 = 1
10 = 2
...
n = len(n)
```

- [x] Forstå konfliktløsing ved kjeding (chaining)
  - Kan brukes i hashing. Bare en bedre system for å bruke det.
  - Som bucket sort. Plasserer alle i samme "bøtte" om de får samme genererte indeks.
  - Kolliderer de, som at lengden (forrige eksempel) er lik, plasseres de i samme kjede.
    - Dette blir en lenket liste for hver av indeksene (lengdene).
- [x] Kjenne til grunnleggende hashfunksjoner
  - Lengde, ASCII-verdier <- Sortering.
  - RSA, DSA <- Kryptografi.
- [x] Vite at man for statiske datasett kan ha worst-case O(1) for søk
  - Bare å lage en hashfunksjon selv og se.
  - Enklere for en datamaskin å hente ut en hashverdi fra et sted den vet hvor er i minnet.
- [x] Kunne definere amortisert analyse
- [x] Forstå hvordan dynamiske tabeller fungerer
  - Brukes blant annet i python:
    - `objekt.attr = objekt[attr]`
  - Ikke effektivt, men fleksibelt.

## Forelesning 3 - Splitt og hersk

- [x] Forstå strukturell induksjon∗
  - `+ 2`

    - `+ 2 x 2`
    - `+ ...`
    - `+ 2 x ... x 2`
    - `= 2 x ... x 2 x 2 - 2`
    - `+ 2 x ... x 2 x 2`
    - **`= 2 x ... x 2 x 2 - 2`**
    - `sum^ln(n)_{i=1} 2^i = 2n - 1`
- [x] Forstå designmetoden divide-and-conquer (splitt og hersk)
    - `len(2 x 2 x 2 x ... x 2) = m`
    - `lg(2 x 2 x 2 x ... x 2) = m`
    - `lg(3 x 3 x 3 x ... x 3) = m`
    - `lg(k x k x k x ... x k) = m`
- [x] Kunne løse rekurrenser med substitusjon, rekursjonstrær og masterteoremet
    - `f(x) = 1 + f(x)`
    - `[f(x)] = [1 + [f(x)]]`
    - `[f(x)] = [1 + [1 + [f(x)]]]`
    - `[f(x)] = [1 + [1 + [1 + [...]]]]`
    - `f(x) = 1 + f(x - 1)`
    - `[f(x)] = [1 + f(x - 1)]`
    - `[f(x)] = [1 + f(x - 1)]`
- [x] Forstå hvordan variabelskifte fungerer
- [x] Forstå Quicksort og Randomized-Quicksort
    - Quicksort starter på siste element.
    - Randomized starter på et tilfeldig element.
    - Worst case er ferdig sortert. $O(n^2)$.
- [x] Forstå binærsøk
    i. Sortèr utvalget.
    ii. Dele i 2.
    iii. Velg den biten som inneholder søket.
    iv. Fortsett punkt 2 og 3 til elementet er funnet.

- Notater:
    - Merge sort:

```
Merge(A, p, q, r)
1 Copier L til R
2 for k = p til r
3       if L[i] <= R[j]
4           A[k] = L[i]
5           i++
6       else A[k] = R[j]
7           j++
```

    - Litt grunnleggende:
        - `log(x^(log(y))) = log(y^(log(x)))`
        - `2^log₃(n) = n^log₃(2)`

## Forelesning 4 - Rangering i lineær tid

- [x] Forstå hvorfor sammenligningsbasert sortering har en worst-case på Ω(n lg n)
    - De sammenligner to og to elementer.
- [x] Vite hva en stabil sorteringsalgoritme er
    - Det er en algoritme som bevarer rekkefølgen basert på et sorteringskriterie.
    - Eks:
        - { (B, 2), (C, 2), (A, 1), (B, 1) }
        - => (sorterer på **første** nøkkel-verdi)
        - { (A, 1), (B, 2), (B, 1), (C, 2) }
- [x] Forstå Counting-Sort, og hvorfor den er stabil
    - Eksempel:

        - `list = [ 0a, 4a, 1a, 2a, 4b, 1b ]`
        - Lager først en liste fra max til min verdier som forekommer.
        - `min = 0, max = 4`
        - `countings = [0] * (max - min)`
        - Så teller den forekomster i den nye listen. Indeksen er tallene.
        - `countings = [ 1, 2, 1, 0, 2 ]`
        - Denne listen med countings viser nå plasseringer relativt til hverandre.
        - Vi vil ha absolutt verdier, så vi ikke er avhengig av tidligere, så vi oppdaterer den slik:
        - `countings = [ 1, 3, 4, 4, 6 ]`
        - Lager en liste med lik lengde som `list` . Her skal den ferdige listen lages.
        - `new_list = [null] * (max - min)`
        - Så for hvert element i `list` , går vi enkelt i `countings` og finner posisjonen:
        - `1b ligger i countings[1]. countings[1] = 3 - ergo er 1b i posisjon 3`
        - `countings[1] -= 1`
        - `4b ligger i countings[4]. countings[4] = 6 - ergo er 1b i posisjon 3`
        - `countings[3] -= 1`
        - Etter å legge inn de 2 bakerste elementene er `countings` listen slik:
        - `countings = [ 1, 2, 4, 4, 5 ]`
        - Vi itererer gjennom hele og ender opp med:
        - `list = [ 0a, 4a, 1a, 2a, 4b, 1b ]`
        - `new_list = [ 0a, 1a, 1b, 2a, 4a, 4b ]`
        - `countings = [ 0, 1, 3, 4, 4 ]`
    - Eksempelet viser også at algoritmen er stabil.
        - Viktig å gå bakover når man skal plukke ut elementer til den nye listen, eller blir den ustabil.
- [x] Forstå Radix-Sort, og hvorfor den trenger en stabil subrutine
    - Den sorterer på f.eks. først enere, så tiere osv.
    - Den sorterer like mange ganger som det er nøkler å basere seg på.
        - Altså er det tall, som det i dette faget er, blir dette antall siffere på det høyeste tallet.
    - Om den ikke er stabil vil alle de tidligere sorteringene feile. Den er avhengig av tidligere sorteringer.
- [x] Forstå Bucket-Sort
    - Eksempel:
        - Du har 10 bøtter.
        - Du vet du skal sortere tall mellom 1 og 10.
        - Du kan så putte elementene i hver sin bøtte
        - Så bygge opp resultatet ved å ta ut tallene fra bøttene i riktig rekkeføge.
    - Vil ikke sortere grunntigere enn bøttene.
    - Krever en del RAM (Minne), pga antall bøtter kan bli ganske mange.
    - Er stabil. Den bygger en stack i hver bøtte. Så først inn - sist ut.
- [x] Forstå Randomized-Select
    - Velger heller en random pivot for å unngå worst-case O($n^2$).
    - En hybrid av Quicksort og binærsøk.
- [x] Forstå Select
    - Notater:

```
Rekursjon:

Time:
1 if n > 1
2     t = Time(n - 1)
3     return t + n
4 else return 1

Iterativ:

Rec-Ins-Sort:
1 if j > 1
2     Rec-Ins-Sort(A, j - 1)
3     key = A[j]
4     i = j - 1
```

```
5     while i > 0 and A[i] > key
6         A[i + 1] = A[i]
7         i--
8     A[i + 1] = key
```

## Forelesning 5 - Rotfaste trestrukturer

- [x] Forstå hvordan heaps fungerer, og hvordan de kan brukes som prioritetskøer
  - Trestruktur slik at man kan akksessere elementene i O(lg(n)) tid.
- [x] Forstå Heapsort
  - Steg
    - Build-max-heap O(n)
    - Extract-max O(log(n)) - fordi trestrukturen.
    - Max-heapify O(log(n))
    - Max-heap-insert O(log(n))
    - Heap-increase-key O(log(n))
    - Heap-maximum Θ(1) - Velger bare den første noden (rotnoden, som også er først i stacken).
- [x] Forstå hvordan binære søketrær fungerer
  - Fungerer ofte rekursivt mtp. hvordan trærene kan enkelt deles opp i to.

```
Successor(x)
1 if x.right != NIL
2     return Min(x.right)
3 y = x.p
4 while y != NIL and x == y.right
5     x = y
6     y = y.p
7     return y
```

  - Traversering

```
Inorder-Walk(x)
1 if x != NIL
2      Inorder-Walk(x.left)
3      print x.key
4      Inorder-Walk(x.right)
```

- [x] Forstå flere ulike operasjoner på binære søketrær, ut over bare søk
  - Insetting

```
Tree-Insert(T, z)
1 y = NIL
2 x = T.root
3 while x != NIL
4     y = x
5     if z.key < x.key
6         x = x.left
7     else x = x.right
8 z.p = y
9 if y == NIL
10    T.root = z
11 else if z.key < y.key
12    y.left = z
13 else y.right = z
```

  - Transplant - brukes ved sletting

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

  - Sletting

```
Tree-Delete(T,z)
1 if z.left == NIL
2     Transp(T, z, z.right)
3 elseif z.right == NIL
4     Transp(T, z, z.left)
5 else y = Minimum(z.right)
6     if y.p != z
7         Transp(T, y, y.right)
8         y.right = z.right
9         y.right.p = y
10    Transp(T, z, y)
11    y.left = z.left
12    y.left.p = y
```

- [x] Vite at forventet høyde for et tilfeldig binært søketre er Θ(lg n)
- [x] Vite at det finnes søketrær med garantert høyde på Θ(lg n)
- Kjøretider for binære søketre:

| Algoritme | Kjøretid |
|---|---|
| Inorder-Tree-Walk | Θ(n) |
| Tree-Search | O(h) |
| Tree-Minimum | O(h) |
| Tree-Successor | O(h) |
| Tree-Insert | O(h) |
| Tree-Delete | O(h) |

## Forelesning 6 - Dynamisk programmering

- [x] Forstå ideen om en delproblemrelasjon eller delproblemgraf
  - DP er nytting om man har overlappende delproblemer.
  - Korrekt om man har optimal substruktur.
- [x] Forstå induksjon over velfunderte relasjoner ∗
- [x] Forstå designmetoden dynamisk programmering
  - Hva er **DP**?
    a. **Karakterisere strukturen** av en optimal løsning.
    b. **Rekursivt definere verdi** av den optimale løsningen.
    c. **Kalkuler verdien** av en optimal løsning.
    d. **Konstruer** en optimal løsning fra beregnet informasjon.
  - Oppskrift, **Sniedivich**
    a. **Embed** your problem in a family of related problems.
    b. **Derive** a relationship between the solutions to these problems.
    c. **Solve** this relationship.
    d. **Recover** a solution to your problem from this relationship.
- [x] Forstå løsning ved memoisering (top-down)
  - Nyttig når vi har overlappende løsninger.
  - Korrekt når vi har optimal substruktur.
- [x] Forstå løsning ved iterasjon (bottom-up)
- [x] Forstå hvordan man rekonstruerer en løsning fra lagrede beslutninger
- [x] Forstå hva optimal delstruktur er
- [x] Forstå hva overlappende delproblemer er
- [x] Vite forskjellen på et segment og en underfølge (subsequence)
- [x] Forstå eksemplene stavkutting, matrisekjede-multiplikasjon og LCS
  - Stavkutting. Denne er eksponentiell:

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

```
q, t = -∞, - > 6.6 > -, -
```

- En annen metode med 2 prosedyrer som er kvadratisk O(n²):

```
Memoized-Cut-Rod(p, n)
1 let r[0 ..n] be a new array
2 for i = 0 to n
3  r[i] = -∞
4 return Aux(p, n, r)
```

- Selve prosedyren:

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

- **LCS**
  - Mye brukt i bioinformatikk.
  - Eksempel 1:

```
1 klapper takpapp
2 kapp akpapp
3 kapp kapp
```

  - Eksempel 2:

```
1234567 1234567`
klapper takpapp
```

  - Endelig eksempel:

```
. . . . . | s t o r m k a s t
. . . . 0 | 1 2 3 4 5 6 7 8 9
. -------------------------
. | 0 | 0 | 0 0 0 0 0 0 0 0 0
. -------------------------
a | 1 | 0 | 0 0 0 0 0 0 1 1 1
t | 2 | 0 | 0 1 1 1 1 1 1 1 2
o | 3 | 0 | 0 1 2 2 2 2 2 2 2
m | 4 | 0 | 0 1 2 2 3 3 3 3 3
m | 5 | 0 | 0 1 2 2 3 3 3 3 3
a | 6 | 0 | 0 1 2 2 3 3 4 4 4
k | 7 | 0 | 0 1 2 2 3 4 4 4 4
t | 8 | 0 | 0 1 2 2 3 4 4 4 5
```

  - Observasjon 1:

```
. . . . . | s t o r m k a s t
. . . . 0 | 1 2 3 4 5 6 7 8 9
. -------------------------
. | 0 | 0 | 0 0 0 0 0 0 0 0 0
. -------------------------
a | 1 | 0 | ⇑ ↑ ↑ ↑ ↑ ↑ ↑ ← ←
t | 2 | 0 | ↑ ⇖ ← ← ← ← ↑ ↑ ↖
o | 3 | 0 | ↑ ↑ ⇖ ⇐ ← ← ← ← ↑
m | 4 | 0 | ↑ ↑ ↑ ↑ ⇖ ⇐ ← ← ←
m | 5 | 0 | ↑ ↑ ↑ ↑ ↖ ⇑ ↑ ↑ ↑
a | 6 | 0 | ↑ ↑ ↑ ↑ ↑ ↑ ⇖ ⇐ ←
k | 7 | 0 | ↑ ↑ ↑ ↑ ↑ ↖ ↑ ⇑ ↑
t | 8 | 0 | ↑ ↖ ↑ ↑ ↑ ↑ ↑ ↑ ⇖
```

  - Observasjon 2:

```
. . . . . | s t o r m k a s t
. . . . 0 | 1 2 3 4 5 6 7 8 9
. -------------------------
. | 0 | 0 | 0 0 0 0 0 0 0 0 0
. -------------------------
a | 1 | 0 | ↑ ↑ ↑ ↑ ↑ ↑ ← ←
t | 2 | 0 | ↑ ↖ ← ← ← ← ↑ ↑ ↖
o | 3 | 0 | ↑ ↑ ↖ ← ← ← ← ← ↑
m | 4 | 0 | ↑ ↑ ↑ ↑ ↖ ← ← ← ←
m | 5 | 0 | ↑ ↑ ↑ ↑ ↖ ↑ ↑ ↑ ↑
a | 6 | 0 | ↑ ↑ ↑ ↑ ↑ ↑ ↖ ← ←
k | 7 | 0 | ↑ ↑ ↑ ↑ ↑ ↖ ↑ ↑ ↑
t | 8 | 0 | ↑ ↖ ↑ ↑ ↑ ↑ ↑ ↑ ↖
```

- [x] Forstå løsningen på 0-1-ryggsekkproblemet

- Eksempler på DP-problemer:
  - Stavkutting.
  - LCS (Longest Common Subsequence).
  - Ryggsekk.
  - Matrisekjede.

## Forelesning 7 - Grådige algoritmer

- [x] Forstå designmetoden grådighet
  - Løs det mest lovende delproblemet rekursivt.
  - Bygg løsningen på denne delløsningen.
- [x] Forstå grådighetsegenskapen (the greedy-choice property)
  - Grådighetsegenskapen:
    - Vi kan velge det som ser best ut, her og nå.
    - **Vi kan finne en global optimal løsning ved å ta lokalt optimale valg.**
  - Optimal substruktur:
    - En optimal løsning bygges av optimale delløsninger.
  - `Grådig valg + optimal delløsning => optimal løsning`
  - Trinn for å identifisere.
    a. Globalt opmalitetskriterium.
    b. Lokalt opmalitetskriterium.
    c. Kontruksjontrinn. Ny lokalt optimum i hvert trinn. Det skal lede til et globalt et.
- [x] Forstå eksemplene aktivitet-utvelgelse og det fraksjonelle ryggsekkproblemet
- [x] Forstå Huffman og Huffman-koder
  - `abbabcad` i bits
    - `a=000, b=001, c=01, d=1`
    - `000 001 001 000 001 01 001 1 (uten mellomrom)`
    - Prefix hindrer lesing fra å kunne kombineres med andre tegn.
- [x] Forstå bevismetoden bevis ved fortrinn (exchange arguments) ∗
  - Exchange arguments.
  - Betrakt en vilkårlig løsning og gradvis tranformer den til en gårdig enn, uten å miste kvaliteten.
    - Den grådige blir nå minst like god som alle andre.
  - Prøv selv med aktiviteter med varighet og dealine hvor du alltid velger den som slutter tidligst.
    - Eksempel i "Notater" (2 linjer under denne).
- [x] Forstå bevismetoden bevis ved forsprang (staying ahead) ∗

- Notater:
  - Aktivitetsutvalg (tabell), start på første slutt (her: 0-**2**):

```
0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
|←-----→| . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| . |←--------------→| . | . | . | . | . | . | . | . | . | . | . |
| . | . | . | . |←----------→| . | . | . | . | . | . | . | . | . |
|←------------------------------→| . | . | . | . | . | . | . | . |
| . | . | . | . | . |←--------------→| . | . | . | . | . | . | . |
| . | . | . | . | . | . | . |←------→| . | . | . | . | . | . |
| . | . | . | . | . | . | . | . |←------→| . | . | . | . | . |
```

```
| . | . | . | . | . | . | . | . | . | . | . |←----------→| . | . |
| . | . | . | . | . | . | . | . | . | . | . | . | . |←----------→|
```

- Aktivitetsutvalg (etter algoritme):

```
0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
|███████| . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| . |←------x------→| . | . | . | . | . | . | . | . | . | . | . | . |
| . | . | . | . |███████████| . | . | . | . | . | . | . | . | . | . |
|←--------------x--------------→| . | . | . | . | . | . | . | . |
| . | . | . | . | . |←-------x------→| . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . |███████| . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . |←--x--→| . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . | . | . | . |███████████| . | . |
| . | . | . | . | . | . | . | . | . | . | . | . | . | . |←----x----→|
```

- Grådig Algoritme:

```
Recursive-Activity-Selector(s, f, m, n)
1 m = k + 1
2 while m ≦ n and s[m] < f[k]
3    m = m + 1
4 if m ≦ n
5    S = Recursive-Activity-Selector(s, f, m, n)
6    return {aₘ} ∪ S
7 else return ∅
```

- Iterativ Algoritme:

```
Greedy-Activity-Selector(s, f)
1 n = s.length
2 A = {a₁}
3 for m = 2 to n
4    if s[m] ≧ f[k]
5        A = A ∪ {aₘ}
6        k = m
7 return A
```

## Forelesning 8 - Traversering av grafer

- [x] Forstå hvordan grafer kan implementeres
  - Defineres som G = (V, E) | V = |noder|, E = |kanter|
  - Nabolister, som beskriver grafen
    - A -> B*
    - B -> D -> E*
    - C -> A -> E*
  - Nabomatrise

```
 . | A B C D E
 -------------
A | 0 1 0 0 0
B | 0 1 0 1 1
C | 1 1 0 0 1
D | 0 0 0 0 0
E | 0 0 0 0 0
```

  - Tegning:

. . . . . +---+ . . . . . +---+ . . . . +-----+ | A | <-------+ | C | . . . . | . . . +---+ . . . . . +---+ . . . . v . . . . . . . . . . . . . . . . . . . . . + . . . . . . . . . . . . +---+ . . . | . . . . . . . . . . . . . | B | . . . | . . . . . . . . . . . . . +---+ . . . | . . . . . .+---+. . . . . . . . . +------------>.|.E.|. . . . + . . . . . . . . . . . . .+---+. . . . | . . . . . . . . . . . . . . . . . . | . . . .+---+. . . . . .^. . . . . +------>.|.D.|.+----------+. . . . . .+--- . . . . . .

- Graftraversering
  - British Museum Algorithm
    - Gå en tilfeldig vei. Vil å evig fordi den ikke husker.
- [x] Forstå BFS, også for å finne korteste vei uten vekter

- Breadth-first search:
  - Går til hvert naboelement. Fra de igjen - velg naboene som ikke er valgt.
- Breadth-first search, algoritme:

```
BFS(s, Adj)
1 level = { s: ø }
2 parent = { s: None }
3 i = 0
4 frontier = [s] ← level i - 1
5 while frontier:
6     next = [] ← level i
7     for u in frontier:
8         for v in Adj[u]:
9             if v not in level: # Unngå duplikater
10                level[v] = i
11                parent[v] = u
12                next.append(v)
13     frontier = next
14     i = i + 1
```

- [x] Forstå DFS og **parentesteoremet** (parantes)
  - Depth-first search.
    - Går helt inn til en kant før den traverserer en annen vei.
  - DFS. Algoritme:

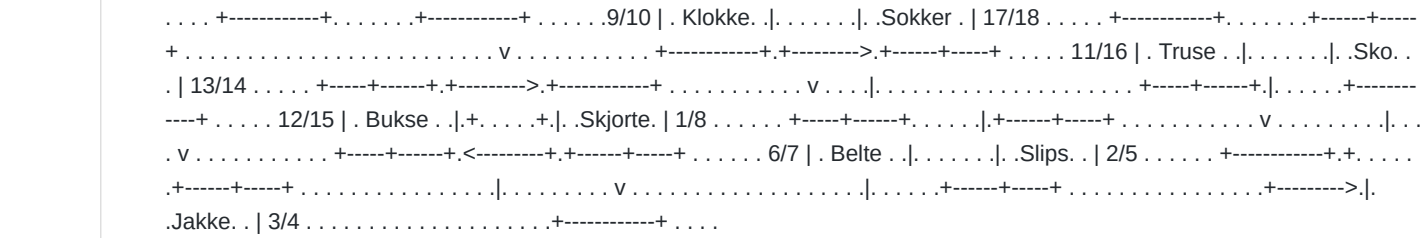
```
DFS(G)
1 for each vertex u ∈ G.V
2    u.color = white
3    u.π = NIL
4 time = 0 # global
5 for each vertex u ∈ G.V
6    if u.color == white
7        DFS-Visit(G, u)
```

  - DFS-Visit funksjon:

```
DFS-Visit(G, u)
1 time = time + 1
2 u.d = time
3 u.color = gray
4 for each v ∈ V.Adj[u]
5    if v.color == white
6        v.π = u
7        DFS-Visit(G, v)
8 u.color = black
9 time = time + 1
10 u.f = time
```

- [x] Forstå hvordan DFS klassifiserer kanter
  - 3 statuser:
    - Før oppdagelse.
    - Er i stacken. Har naboer som er i stack eller ikke oppdaget.
    - Tatt ut av stack.
  - 4 cases:
    - **Three edges:** Første oppdagelse. → **Ref. Forelesning 9.**
    - **Back edges:** En sykel.
    - **Forward edges:** Ikke en three edge. Hvis u er forgjenger av v.
    - **Cross edges:** Alle andre kanter.
- [x] Forstå Topological-Sort
  - Def.: Hvilken rekkefølge som er lov.
  - Eks: Grunnmur.
    - Vegger.
      - Tak.
        - Interiør.
      - Vinduer.
        - Interiør.

        - Vann.
            - Interiør.
    - Gir nodene en rekkefølge.
    - Foreldre før barn.
    - Evt.: Alle kommer etter avhengigheter.
    - Krevevr DAG (dvs. velfundert)! (Directed Acyclic Graph)
    - Ex:




- [x] Forstå hvordan DFS kan implementeres med en stakk ∗
    - Stack <=> Rekursjon. Implementeres med en callstack.
- [x] Forstå hva traverseringstrær (som bredde-først- og dybde-først-trær) er
- [x] Forstå traversering med vilkårlig prioritetskø ∗

### Forelesning 9 - Minimale spenntrær

- [x] Forstå skog-implementasjonen av disjunkte mengder
- [x] Vite hva spenntrær og minimale spenntrær er
    - Backlog
        - Kantklassifisering, DFS
        - Parentesteoremet
        - Topologisk sortering
    - Minimale:
        - Disjunkte mengder
        - Generisk MST (Minimal Spanning Tree)
        - Kruskals algoritme
        - Prims algoritme
    - Spenntrær:
        - Sammenhengene spennskog.
            - Spennskog: Dekkende skog eller asyklisk spenngraf.
    - Disjunkte mengder:
        - Union by rank-keuristikk
            - Rang er øvre grense for node høyde.
        - m operasjoner: O(m*a(n))
        - Sette varablene

```
Make-Set(x)
1 x.p = x
2 x.rank = 0
```

        - Sette varablene

```
Union(x, y)
1 Link(Find-Set(x), Find-Set(y))
```

        - Sette varablene

```
Link(x, y)
1 if x.rank > y.rank
2     y.p = x
3 else x.p = y
4     if x.rank == y.rank
5         y.rank = y.rank + 1
```

```
Find-Set(x)
1 if x != x.p
2      x.p = Find-Set(x.p)
3 return x.p
```

- [x] Forstå Generic-MST
  - Knytter sammen nodene i en graf på billigs mulig måte.
  - Kan ha negative kanter, men krever at "MST" blir asyklisk.
- [z] Forstå hvorfor lette kanter er trygge kanter
  - Kantklassifiseringer:
    - Tre-kanter
      - Konter i dybde-først-"skogen".
    - Bakoverkanter
      - Kanter til en forgjenger i DF-"skogen".
    - Foroverkanter
      - Kanter utenfor DF-skogen to en etterkommer i DF-skogen.
    - Krysskanter
      - Alle andre kanter.
  - Hvordan definere klassene
  - Møter en hvit node
    - Tre-kant
  - Møter en grå node
    - Bakoverkant
  - Møter en svart node:
    - Forover- eller krysskant
- [x] Forstå MST-Kruskal
  - En kant med minimal vekt blant de gjenværende er trygg så lenge den ikke danner sykler.
  - Algoritme:

```
MST-Kruskal(G, w)
1 A = ø
2 for each vertex v ∈ G.V
3    Make-Set(v)
4 sort G.E by w
5 for each edge (u, v) ∈ G.E
6    if Find-Set(u) != Find-Set(v)
7      A = A U {(u, v)}
8      Union(u, v)
9 return A
```

  - Kjøretid:

```
Operasjon Antall   Kjøretid
**Make-Set**    V        O(1)
**Sortering**   1        O(E lg E)
**Find-Set**    O(E)     O(lg V )
**Union**       O(E)     O(lg V )
**Totalt**: O(E lgV)
|E| < |V|² => lg|E| < 2 lg|V| => lgE = O(lgV)
```

- [x] Forstå MST-Prim
  - Bygger et tre gradvis; en lett kant over snittet rundt treet er alltid trygg.
  - Hva det er:
    - Kan implementeres vha. traversering
    - Der BFS bruker FIFO og DFS bruker LIFO, så bruker Prim en min-prioritets-kø
    - Prioriteten er vekten på den letteste kanten mellom noden til treet
    - For enkelhets skyld: Legg alle noder inn fra starten, med uendelig dårlig prioritet
  - Algoritme:

```
MST-Prim(G, w, r)
1 for each u ∈ G.V
2     u.key = ∞
3     u.pi = NIL
```

```
4 r.key = 0
5 Q = G.V
6 while Q != ø
7     u = Extract-Min(Q)
8     for each v ∈ G.Adj[u]
9         if v ∈ Q and w(u, v) < v.key
10            v.pi = u
11            v.key = w(u, v)
```

- Kjøretid:

```
Operasjon        Antall   Kjøretid
**Build-Min-Heap**    1       O(V )
**Extract-Min**       V       O(lg V )
**Decrease-Key**      E       O(lg V )
**Totalt**: O(E lgV)
**Dette gjelder om vi bruker en binærhaug**
**Kan forbedres til O(E + V lgV) med Fibonacci-haug**
```

- I det følgende: Farging som for BFS
- Kanter mellom svarte noder er endelige
- Beste kanter for grå noder også uthevet
- Boka uthever bare kantene i spenntreet

## Forelesning 10 - Korteste vei fra én til alle

- [x] Forstå ulike varianter av korteste-vei- eller korteste-sti-problemet
  - En enkel sti er en sti uten sykler
  - En kortest vei vil aldri inneholde en positiv sykel
  - Om vi ikke kan nå noen negative sykler så er «korteste sti» det samme som «korteste enkle sti»
  - Om en sti til v har en negativ sykel, så finnes det alltid en kortere sti – ingen er kortest!
  - Det vil likevel finnes en kortest enkel sti til v, men vi kjenner ingen generelle algoritmer for å finne den
- [x] Forstå strukturen til korteste-vei-problemet
- [x] Forstå at negative sykler gir mening for korteste enkle vei (simple path) ∗
- [x] Forstå at korteste enkle vei er ekvivalent med lengste enkle vei ∗
- [x] Forstå hvordan man kan representere et korteste-vei-tre
- [x] Forstå kant-slakking (edge relaxation) og Relax
- [x] Forstå ulike egenskaper ved korteste veier og slakking
- [x] Forstå Bellman-Ford
  - Gå gjennom alle kanter en gang.
  - Vi vet da at vi må ha vært innom en av nodene en gang.
  - Hvis vi har slakket alle alle kantene k ganger.
  - V - 1 antall ganger.
  - Algoritme:

```
Bellman-Ford(G, w, s)
1 Initialize-Single-Source(G, s)
2 for i = 1 to |G.V| - 1
3    for each edge (u, v) ∈ G.E
4        Relax(u, v, w)
5 for each edge (u, v) ∈ G.E
6    if v.d > u.d + w(u, v)
7        return false
8 return true
```

- [x] Forstå Dag-Shortest-Path
  - Algoritme:

```
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 2 G.Adj[u]
5         Relax(u, v, w)
```

- Kjøretid:

```
Operasjon            Antall    Kjøretid
Topologisk sortering   1       0(V + E)
Initialisering         1       0(V)
Relax                  E       0(1)
Totalt: 0(V + E)
```

- [x] Forstå kobling mellom Dag-Shortest-Path og dynamisk programmering∗
- [x] Forstå Dijkstra
  - Forklaring:
    - Om vi har sykler, kan vi ikke få til topologisk sortering
    - Alternativ: Besøke nodene i stigende avstandsrekkefølge
    - Alle korteste stier får da fortsatt sine kanter slakket i riktig rekkefølge
    - Men vi kjenner jo ikke avstandsrekkefølgen!

- Notater
  - **Sti-slakkings-egenskapen:**
    - Om p er en kortest 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 slakkinger forekommer, selv om de kommer
    - innimellom.
  - Slakkingsalgoritme, init:

```
Initialize-Single-Source(G, s)
1 for each vertex v 2 G.V
2    v.d = 1
3    v.π = nil
4 s.d = 0
```

  - Slakkingsalgoritme, init:

```
Relax(u, v, w)
1 if v.d > u.d + w(u, v)
2    v.d = u.d + w(u, v)
3    v.π = u
```

## Forelesning 11 - Korteste vei fra alle til alle

- [x] Forstå forgjengerstrukturen for alle-til-alle-varianten av korteste vei-problemet
  - **Fungerer oftest ikke med negative sykler.**
  - BFS:
    - Kjøretid = $O(V^2 + VE)$
  - Dijkstra:
    - Kjøretid:
      - Min-prioritets kø: $O(V^2 + VE) = O(V^3)$
      - Binær "min-heap": $O(VE \lg V)$
      - Fibonacci heap: $O(V^2 \lg V + VE)$
  - DAG:
    - Kjøretid = $\Theta(V^2 + VE)$
    - Rettet asyklisk graf. **Kan ha negative sykler.**
  - Bellman Ford:
    - Kjøretid = $\Theta(V^2E)$
- [x] Forstå Floyd-Warshall
  - Kjøretid: $\Theta(n^3)$
  - Den oppdager negative sykler.
  - Eksempel (med rettet graf):

```
W =
D^(0) =
[0 i 6 4]
[2 0 3 i]
```

```
[i-2 0 i]
[i-2 4 0]

D^(1) =
[0 i 6 4]
[2 0 3 6]
[i-2 0 i]
[i-2 4 0]

π^(0) =
[- - 1 1]
[2 - 2 -]
[- 3 - -]
[- 4 4 -]

π^(4) =
[- 4 2 1]
[2 - 2 1]
[2 2 - 1]
[2 4 2 -]
```

- [x] Forstå Transitive-Closure
  - Du har en graf som ...
- Noteter
  - **Intermediate vertex:**
    - xvvvvx - Alle nodene utenom start og slutt i en sti.

## Forelesning 12 - Maksimal flyt

- [x] Kunne definere flytnettverk, flyt og maks-flyt-problemet
  - Flyt:
    - Som en strøm. Kirchoffs lover må fylles fullt ut.

      $f(u, v) \leq c(u, v)$ for alle $u, v \in V$
      $\sum_{v \in V} f(u, v) = 0$ for alle $u, v \in V - \{s, t\}$

  - Kapasitet:
    - $c(u, v)$
  - Residualkapasitet:
    - $c_a(u, v) = c(c, v) - a(u, v)$
  - Flytforøkende sti (augmented path):
    - Sti fra s til t der residualkapasiteten til alle kantene er større enn 0.
  - Maks flyt:
    - Maksimer flyt fra s til t!
- [x] Kunne håndtere antiparallelle kanter og flere kilder og sluk
- [x] Kunne definere residualnettverket til et nettverk med en gitt flyt
  - Residualnettverket:
    - Fremoverkant ved ledig kapasitet.
    - Bakoverlent ved flyt.
- [x] Forstå hvordan man kan oppheve (cancel) flyt
  - Vi kan sende flyt baklengs langs kanter der det alt går flyt.
  - Vi opphever da flyten, så den omdirigeres til et annet sted.
  - Det er dette bakoverkantene i residualnettverket representerer.
- [x] Forstå hva en forøkende sti (augmenting path) er
  - Flytforøkende sti er når det er mer flyt igjen i nettverket.
  - **Hetland def: En sti i residualnettverket der vi kan sende mer flyt.**
- [x] Forstå hva snitt, snitt-kapasitet og minimalt snitt er
  - Deler opp nettverket. Gjerne for å se hvor flaskehalsene ligger.
  - Summen av alle kanter som går fra S til T er kapasiteten til nettverket hittil.
  - Netto flyt i nettverket i et snitt er S til T flyt minus T til S flyt.
- [x] Forstå maks-flyt/min-snitt-teoremet
- [x] Forstå Ford-Fulkerson
  - Finn økende stier så lenge det går.

- Kan ikke ha negative sykler.
- Deretter er flyten maksimal.
- Generell metode, ikke en algoritme.
- Om vi bruker BFS: «Edmonds-Karp» - som ikke kan ha negative kanter.
- Normalt implementasjon:
  - Finn økende sti først
  - Finn så flaskehalsen i stien
  - Oppdater flyt langs stien med denne verdien
- Algoritmer:
  - Metode:

```
Ford-Fulkerson-Method(G, s, t)
1 initialize flow f to 0
2 while there is an augm. path p in Gf
3    augment flow f along p
4 return f
```

  - Dypere:

```
Ford-Fulkerson(G, s, t)
1 for each edge (u, v) 2 G.E
2    (u, v).f = 0
3 while there is a path p from s to t in Gf
4    cf (p) = min {cf (u, v):(u, v) is in p}
5    for each edge (u, v) in p
6       if (u, v) 2 E
7           (u, v).f = (u, v).f + cf (p)
8       else (v, u).f = (v, u).f " cf (p)
```

  - Kjøretid:

| **Operasjon** | **Antall** | **Kjøretid** |
|---|---|---|
| Finn økende sti | O(\|f*\|) | O(E) |
| **Totalt: O(E\|f\|)** | | |

- [x] Vite at Ford-Fulkerson med BFS kalles Edmonds-Karp-algoritmen
  - Alternativ: «Flett inn» BFS
    - Finn flaskehalser underveis!
    - Hold styr på hvor mye flyt vi får frem til hver node
    - Traverser bare noder vi ikke har nådd frem til ennå
  - Denne «implementasjonen» står ikke i boka
  - Edmnfs-Karp algoritme:

```
Edmonds-Karp(G, s, t)
1 for each edge (u, v) Є G.E
2     (u, v).f = 0
3 repeat
4     for each vertex u Є G.V
5         u.f' = 0 # residual flow reaching u
6         u.π = NIL
7     s.f' = 1
8     Q = Ø
9     Enqueue(Q, s)
10    while t.f' == 0 and Q != Ø
11        u = Dequeue(Q)
12        for all edges (u, v),(v, u) Є G.E
13            if (u, v) Є G.E
14                c_f(u, v) = c(u, v) % (u, v).f
15            else c_f(u, v)=(v, u).f
16            if c_f(u, v) > 0 and v.f' == 0
17                v.f' = min(u.f', c_f(u, v))
18                v.π = u
19    Enqueue(Q, v)
20    u, v = t.π, t # at this point, t.f' = c_f(p)
21    while u != NIL
22        if (u, v) Є G.E
23            (u, v).f = (u, v).f + t.f'
24        else (v, u).f = (v, u).f - t.f'
```

```
25        u, v = u.π, u
26 until t.f' == 0
```

- Kjøretid:

| Operasjon | Antall | Kjøretid |
|---|---|---|
| Finn økende sti | O(VE) | O(E) |
| **Totalt: O(VE²)** | | |

- [x] Forstå hvordan maks-flyt kan finne en maksimum bipartitt matching
  - Matching: Delmengde M ⊆ E for en urettet graf *G = (V, E)*
    - Ingen av kantene i M deler noder.
    - Bipartitt matching: M matcher partisjonene.
- [x] Forstå heltallsteoremet
  - For heltallskapasiteter gir Ford-Fulkerson heltallsflyt.
  - Anvendelse i bipartitt matching:
    - Nyreeksempel.
    - Kanter for kapasitet 1.
    - Donoer er kilder og resipienter er sluk.

## Forelesning 13 - NP-kompletthet

- [x] Forstå sammenhengen mellom optimerings- og beslutnings-problemer
  - Selv om L er språket som aksepteres av A, så trenger ikke A avgjøre L, siden den kan la være a svare for nei-instanser (ved å aldri terminere)
  - Den avviser x dersom A(x) = 0
  - Den avgjør et språk L dersom:
    - x ∈ L → A(x) = 1
    - x ∉ L → A(x) = 1
- [x] Forstå koding (encoding) av en instans
- [x] Forstå hvorfor løsningen vår på 0-1-ryggsekkproblemet ikke er polynomisk
  - DP-løsning har kjøretid *T(n, T) = Θ(nW)*
  - Encoding:
    - For enkelthetskyld, la oss si vi bruker Θ(n) bits på objektene. En rimelig encoding vil bruke Θ(m) bits på kapasiteten, *der m = lgW*
  - Polynomisk? **Nei!** Den må kunne skrives som $T(n, m) = \Theta(n2^m)$
- [x] Forstå forskjellen på konkrete og abstrakte problemer
- [x] Forstå representasjonen av beslutningsproblemer som formelle språk
- [x] Forstå definisjonen av klassen P
  - Dette er språkene som kan avgøres i polynomisk tid.
  - Det er disse problemene vi kan løse i praksis. (Cobham's tese.)
- [x] Forstå definisjonen av klassene NP og co-NP
  - Nondeterministic Polynomial Time
  - NP: Språkene som kan **verifiseres** i polynomisk tid.
    - HAM-CYCLE:
      - Språket for Hamilton-sykel-problemet.
      - HAM-CYCLE ∈ **NP**
      - Lett å verifisere i polynomisk tid. Ikke alltid like lett å falsifisere.
  - co-NP: Språkene som kan *falsifiseres* i polynomisk tid.
    - L ∈ **co-NP** ↔ L(overline) ∈ **NP**
    - Tautologi!
- [x] Forstå redusibilitets-relasjonen $\leq_p$
  - Pensum ser på many-one-reduksjoner (Karp-reduksjoner).
  - Hvis A kan reduseres til B, skriver vi: $A \leq_p B$
  - $\leq_p$ er en preordning.
  - Hardhetsbevis:
    - Vise at B er vanskelig => Reduser fra et vanskelig prblem A => etabler $A \leq_p B$.
- [x] Forstå definisjonen av NP-hardhet og NP-kompletthet

    - Kompletthet:
        - Et problem er komplett for en gitt klasse og en gitt type reduksjoner dersom det er maksimalt for redusibilitetsrelasjonen.
        - Dette er altså de vanskeligste i klassen **NP**.
    - Maksimalitet:
        - Et element er maksimalt dersom alle andre mindre eller lik.
        - For reduksjoner: Q er maksimalt dersom alle problemer i klassen kan reduserer til Q.
    - NPC:
        - De komplette språkene i NP, under polynomiske reduksjoner.
    - NP-hardhet:
        - Et problem Q er NP-hardt dersom alle problemer i NP kan reduseres til det.
        - Alle i NP-hard er altså way over alle i NP - og er ikke i NP.
        - NP-komplett (NPC) er samme problemer, men som finnes i NP.

- [x] Forstå den konvensjonelle hypotesen om forholdet mellom P, NP og NPC
- [x] Forstå hvorfor CIRCUIT-SAT er NP-komplett

- Notater:
    - Problem:
        - Generelt. Ex: Korstete vei.
    - Problemistans:
        - Instans av korsteste veien i Google Maps.
    - Hva er et beslutningsproblem:
        - Ja/Nei-svar.
        - Ex:
            - Spenntrær: Finnes det et spenntr i en gitt gra G som har vekt mindre enn eller like et gitt heltall K?
            - Korsteste vei: Finnes det en korsteste vei mellom to gitte node i en graf G som har vekt/avstand mindre enn eller like et gitt heltall K?
            - Travellting Salesman: Gitt en vektet graf G og et heltall K. Finnes det en sukel som besøker alle noder en gang og har total vekt mindre enn ...
    - Problemklasser:
        - Beslutningsproblemer som kan løses i polynomisk tid sier vi at tilhører P.
        - Polynomisk tid: $O(n2^m)$
        - Beslutningsproblemer hvor en gikk løsning kan verifiseres i polynomisk tid sier vi at tilhører **NP**:
        - Verifisere: A sjekke om en gitt lønsning på et problem er en gyldig løsning av problemet.
        - Det store spm: Er elle problemene i NP også i P?
            - NP eksempel:
                - Vertex Cover
        - Uformelt sier vi at et gitt problem A er i klassen NPC hvis vi ved å løse A også kan løse alle andre problem i NP, samtidig som A selv er i NP.
        - Gitt et problem A i NP. Hvordan kan vi vise at A er i NPC?
            - Se fordet at vi har et annet problem B som vi **vet** er i NPC. Hvis vi klarer å vise at A er like vanskelig eller vanskeligere enn B, så vet vi at A også er i NPC.
            - Formelt betyr dette at om vi finner en polynomisk tid reduksjon fra B til A så har vi vist at A er i NPC
    - Problemet er i NP
    - Alle andre problem i NP kan reduseres til disse problemene i polynomisk tid.
    - Fra Clique til Independent Set.
        - Clique: Gitt en graf G og et heltall K. Finnes det en delmengde med noder i G av størrelse K hvor alle nodene er naboer?
        - Independent Set: Gitt en graf G og et heltall K. Finnes det en delmende med noder i G av størrelse K hvor ingen av nodene er naboer?
        - Vi vet: Clique er i NPC
        - Vi ønsker å finne ut: Er Independent Set i NPC?
            - Er Independent Set i NP?
            - Kan vi redusere Clique til Independent Set i polynomisk tid?

### Forelesning 14 - NP-komplette problemer

- [x] Forstå hvordan NP-kompletthet kan bevises ved én reduksjon
- [x] Kjenne de NP-komplette problemene CIRCUIT-SAT, SAT, 3-CNF-SAT, CLIQUE, VERTEX-COVER, HAM-CYCLE, TSP og SUBSET-SUM
  - *SUBSET SUM*
  - CIRCUIT-SAT
    - **Instans:** En krets med logiske porter og én utverdi
    - **Spørsmål:** Kan utverdien bli 1?
    - Vi har et vilkårlig språk/problem L ∈ NP
    - Vi vil redusere dette til CIRCUIT-SAT
    - Det eneste vi vet er at x ∈ L kan verifiseres i polynomisk tid
    - Vi simulerer trinnene i verifikasjonsalgoritmen A med kretser!
    - **Spørsmålet blir:** Kan A (for et eller annet sertifikat) svare/få en output på 1?
    - Eksempel:

```
x ∈ {0, 1}\* => L er i NP, så ...
Det finnes en pol. alg. A, som er slik at x ∈ L nøyaktig når minst én y ∈ {0, 1}\* gir A(x, y) =
.
\* ---> \* (A ---> B => Reduksjon <=> A? Da kan du jo bare B.)
.
...
.
Er x med i språket L?  Kan utverdien bli 1?
```

  - SAT
    - **Instans:** En logisk formel
    - **Spørsmål:** Kan formelen være sann?
    - Direkte oversettelse av logisk krets?
    - Kan gi eksponentielt stor formel!
    - Eksempel:

```
x1---------x5---|
x2-------| . . .|--x8--|
. . . . .|-x6>o-| . . .|
. . . . .| . . .|--x9--|-x10-...
x3--x4>o-|--x7--|------|
.
\* ----> \*
.
Ø = x10 ^ (x4 ⇔ ¬x3)
. . . . ^ (x5 ⇔ (x1 v x2))
. . . . ^ (x6 ⇔ ¬x4)
. . . . ^ (x7 ⇔ (x1 ^ x2 ^ x4))
. . . . ^ (x8 ⇔ (x5 v x6))
. . . . ^ (x9 ⇔ (x6 v x7))
. . . . ^ (x10 ⇔ (x7 ^ x8 ^ x9))
.
Kan utverdien bli 1?  Kan Ø være sann?
```

  - 3-CNF-SAT
    - **Instans:** En logisk formel på 3-CNF-form
    - F.eks.: Ø = (x1 v ¬x2 v x4) ^ ⋯ ^ (¬x7 v x8 v x9)
    - **Spørsmål:** Kan formelen være sann?
    - Vi kan bruke ca. samme reduksjon, på syntakstreet til Ø!
    - Vi får da en formel Ø' av pol. størrelse
    - Ø' er en konjunksjon av termer, hver med maks 3 literaler
    - Dvs.: de to argumentene, samt resultatet av operatoren
    - Hver term gjøres om til CNF vha. en sannhetstabell
    - (x v y) gjøres om til (x v y v z) ^ (x v y v ¬z)
    - Tilsv. blir (x) til fire nye termer
    - Eksempel:

```
Ø' = y1 ^ (y1 ⇔ (y2 ^ ¬x2))
. . . . ^ (y2 ⇔ (y3 v y4))
```

```
. . . . ^ (y3 ⇔ (x1 ! x2))
. . . . ^ (y4 ⇔ ¬y5)
. . . . ^ (y5 ⇔ (y6 v x4))
. . . . ^ (y6 ⇔ (¬x1 ⇔ x3))
Ø'' = CNF, vha. sannhetstabeller
Ø''' = 3-CNF, vha. dummy-variable
.
Kan Ø være sann?  Kan Ø''' være sann?
```

- CLIQUE
  - **Instans:** En urettet graf G og et heltall k
  - **Spørsmål:** Har G en en komplett delgraf med k noder?
  - Vi vil redusere fra 3-CNF-SAT
  - Lag én node i G for hver literal i formelen
  - Ingen kanter mellom noder fra samme term
  - Ellers: Kanter mellom literaler som kan være sanne samtidig
  - La k være antall termer
  - Eksempel:

```
Ø = ( x1 v¬x2 v¬x3 ) ^
. . (¬x1 v x2 v x3 ) ^
. . ( x1 v x2 v x3 )
.
Tilsvarer: x1, x2, x3 = -, 0, 1
.
Kan Ø være sann?  Finnes en k-klikk?
```

- VERTEX-COVER
  - **Instans:** En urettet graf G og et heltall k
  - **Spørsmål:** Har G en et nodedekke med k noder? Dvs., k noder som tilsammen ligger inntil alle kantene
  - En klikk er en komplett delgraf
  - Tilsvarer en uavhengig mengde (kantfri delgraf) i komplementet !G = (V, !E)
  - Nodene utenfor en uavhengig mengde utgjør et nodedekke
  - Hvis G har en k-klikk . . .
  - . . . så har !G = (V, !E) en uavh. mengde med k noder ...
  - . . . og dermed også et (|V| - k)-nodedekke
  - Samme resonnement holder i motsatt retning
- HAM-CYCLE
  - Vi reduserer VERTEX-COVER $\leq_p$ HAM-CYCLE.
  - VERTEX-COVER dekker problemene HAM-CYCLE møter på da den skal bestemme den ene veien gjennom alle nodene.
- TSP
  - Vi kan redusere HAM-CYCLE $\leq_p$ TSP.
  - TSP er? korteste HAM-CYCLE.

- [x] Forstå NP-kompletthetsbevisene for disse problemene
- [x] Forstå at 0-1-ryggsekkproblemet er NP-hardt
- [x] Forstå at lengste enkle-vei-problemet er NP-hardt
- [x] Være i stand til å konstruere enkle NP-kompletthetsbevis

- Notater:
  - Q ∈ NPC ⇔
    - a. ∀ L ∈ NP ⇒ **NP-hard (NPH)**
    - b. Q ∈ NP

## Appendix A

- [x] Setningslogikk
  - And, or, not
- [x] Predikatslogikk
  - Forhold mellom de induviduelle x, y, x... og kvantifikatorene "for alle x" og "det eksisterer en x"

- [x] Inferensregler
    - Introduksjon og eliminasjon
    - Formelle systemer
        - Regler
            - | P | Q |
            - | 0 | 0 |
            - | 0 | 1 |
            - | 1 | 0 |
            - | 1 | 1 |
        - Implikasjon P ⇒ Q
        - Modus Ponens P ⇒ Q, P | Q
        - P ⇒ Q
        - P^- vs. ¬P
        - P ⇒ Q ≡ ¬Q ⇒ ¬P
        - P ⇒ Q, Q ⇒ P | P ⇔ Q
        - P ⇔ Q | P ⇒ Q, Q ⇒ P
        - P ⇔ Q ⇔ R ≡ P ⇒ Q ⇒ R ⇒ P
    - Negasjon, Reductio ad Absurdum
        - Q, ¬Q | ⊥
    - Andre bevis for at P eller Q er sant
        - P, Q | P ∧ Q
        - P ∧ Q | P, Q
    - Proof by Cases
        - P | P ∨ Q
        - P ∨ Q, P...R, Q...R | R
    - Annet bevis for eller
        - P ∨ Q ≡ ¬P ⇒ Q
    - Kvantifikatorer
        - Alle
            - P(a) | ∀xP(x)
            - ∀xP(x) | P(a)
        - Element
            - P(a) | ∃xP(x)
            - ∃xP(x) | P(y)
        - E!xP(x):
            - xP(x) ∀x∀y((P(x) ∧ P(y)) ⇒ x = y)

# Sorteringsalgoritmer i pensum

**Sammenligningsbasert:** Sammenligner to elementer for å se hvem som skal stå først i sekvensen. Her er algoritmene begrenset av Ω(nlgn) som nedre kjøretid.

**Split og hersk:** Deler opp sekvensen i mindre biter for å få kontroll over listen.

**In-place:** Bruker eksisterende struktur uten å lage en ny kopi.

**Stabil:** Like elementer blir "samlet" i samme rekkefølge som før sortering.

## Bubble sort

- [x] Sammenligningsbasert.
- [ ] Split og hersk.
- [x] In-place.
- [x] Stabil.

## Insertion sort

- [x] Sammenligningsbasert.
- [ ] Split og hersk.
- [x] In-place.
  - Bytter på to og to elementer.
- [x] Stabil.
  - Vil aldri flytte to like elementer forbi hverandre, uansett om man starter foran eller bak.

### Merge sort

- [x] Sammenligningsbasert.
- [x] Split og hersk.
- [ ] In-place.
  - Pensum diskuterer ikke hvordan man kan gjøre den in-place.
- [x] Stabil.
  - Bare hvis den alltid velger element fra venstre halvdel om elementene er like.

### Quicksort

- [x] Sammenligningsbasert.
- [x] Split og hersk.
- [x] In-place.
  - Den er rekursiv og "møblerer" om på elementene i returneringsfasen av algoritmen.
- [ ] Stabil.
  - Kan gjøres stabil, men mer effektiv uten.

### Selection sort

- [x] Sammenligningsbasert.
- [ ] Split og hersk.
- [x] In-place.
- [x] Stabil.

### Bucket sort

- [ ] Sammenligningsbasert.
- [ ] Split og hersk.
  - Den er ikke rekursiv og splittes bare opp til to nivåer.
- [x] In-place.
  - Må lage nye "bøtter" som blir en ny datastruktur i minnet.
- [x] Stabil.
  - Njæææ, både og, ikke opplagt. Øvingsfoiler sier ja. Kan basere oss på det siden boka også sier den bruker insertion-sort på hver bøtte - som også er stabil. En del YouTube-videoer sier også at den er stabil.

### Counting sort

- [ ] Sammenligningsbasert.
- [ ] Split og hersk.
- [ ] In-place.
  - Den lager en ny tabell med linker til de nye elementene som "injectes" på riktig plass i telle-tanke-systemet.
- [x] Stabil.
  - Ifølge boka, som også er pensum. Må være stabil for å brukes i radix-sort.

### Heapsort

- [ ] Sammenligningsbasert.
- [x] Split og hersk.
- [x] In-place.
  - Bruker eksisterende tre til å swappe elementer.
- [ ] Stabil.
  - Tar ikke hensyn til rekkefølge ettersom den baserer seg på en heap.

### Radix sort

Antar den bruker counting-sort eller merge-sort. Så den bruker counting-sort/merge-sort-algoritmen like mange ganger som siffer. Tar inn n elementer med d siffer innenfor et k intervall.

- [ ] Sammenligningsbasert.
- [ ] Split og hersk.
- [ ] In-place.
  - Bruker counting-sort.
- [x] Stabil.
  - Fordi counting-sort og merge-sort er stabil. Feiler om counting-sort eller merge-sort ikke er stabil.

## Kjøretider

| Algoritme | Best case | Average case | Worst case |
|---|---|---|---|
| **Insertion sort** | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| **Selection sort** | $O(n^2)$ | $\Theta(n^2)$ | $O(n^2)$ |
| **Merge sort** | $\Theta(n \lg(n))$ | $\Theta(n \lg(n))$ | $\Theta(n \lg(n))$ |
| **Heapsort** | $O(n \lg(n))$ | $O(n \lg(n))$ | $O(n \lg(n))$ |
| **Quicksort** | $\Theta(n \lg(n))$ | $\Theta(n \lg(n))$ | $\Theta(n^2)$ |
| **Bubble sort** | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| **Bucket sort** | $\Theta(n+k)$ | $\Theta(n+k)$ | $\Theta(n^2)$ |
| **Counting sort** | $\Theta(n+k)$ | $\Theta(n+k)$ | $\Theta(n+k)$ |
| **Radix sort** | $\Theta(d(n+k))$ | $\Theta(d(n+k))$ | $\Theta(d(n+k))$ |
| **Select** | $O(n)$ | $O(n)$ | $O(n)$ |
| **Randomized select** | $O(n)$ | $O(n)$ | $O(n^2)$ |