TDT4120 Algoritmer og Datastrukturer

Yathuwaran Raveendranathan

November, 2019

Innhold

1 Problemer og Algoritmer

Kap. 1. The role of algorithms in computing

Kap. 2. Getting started: Innledning, 2.1–2.2

Kap. 3. Growth of functions: Innledning og 3.1

1.1 Kjenne egenskapene til random-access machine-modellen (RAM)

Tar utgangspunkt i en *generic one-processor, random access memory model* som kjører programmene som aloritmene er implementert i. I modellen blir aritmetiske instruksjonene utført en etter en! Hver instruksjon tar konstant tid.

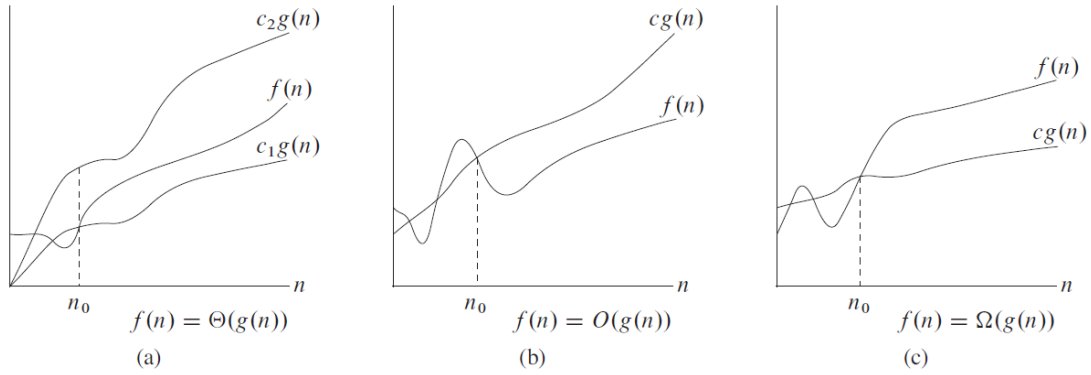
1.2 Kunne definere problem, instans og problemstørrelse

Problem: Relasjon mellom input og output

Instans: Én bestemt input

Problemstørrelse, n: Lagringsplass for en gitt instans.

1.3 Kunne definere asymptotisk notasjon, O, Ω, Θ, o og ω



Figur 1: Visualisering av notasjoner

Brukes til å definere kjøretid under forskjellige omstendigheter.
$\Omega(g(n))$ blir brukt når $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for alle $n \geq n_0$. Kjøretidsgrafen er da skvist mellom en øvre og nedre grense.
$O(g(n))$ blir brukt når $0 \leq f(n) \leq cg(n)$ for alle $n \geq n_0$. Vi har da at kjøretidsgrafen ligger under en øvre grense.
Ω blir brukt når $0 \leq cg(n) \leq f(n)$ for alle $n \geq n_0$. Vi har da at kjøretidsgrafen ligger over en nedre grense.

$o(g(n))$ blir brukt når vi ikke har asymptotisk tett øvre grense. $o(g(n)) : n_0 \geq 0$ slik at

$0 \leq f(n) < cg(n)$ for alle $n \geq n_0$.
På samme måte blir $\omega(g(n))$ brukt når nedre grense ikke er asymptotisk tett. $o(g(n))$: $n_0 > 0$ slik at $0 \leq cg(n) < f(n)$ for alle $n \geq n_o$

1.4 Kunne definere best-case, average-case og worst-case

Worst-Case kjøretid av en algoritme gir oss en øvre grense for en tilfeldig input. Det gir oss en garanti påat algoritmen aldri bruker lenger tid.
Average-Case er forventet kjøretid, gitt en sannsynlighetsfordeling.
Best-Case gir beste mulige kjøretid for en gitt størrelse.

1.5 Forstå løkkeinvarianter og induksjon

En løkkeinvariant er en egenskap ved en programvareløkke som er sann før og etter hver iterasjon. Det er en logisk forsikring, som noen ganger blir sjekket innenfor koden ved hjelp av et forsikringkall. Å kjenne til slike invarianter er viktig for forståelsen av løkkens effekt.
Induksjon går ut på å først bevise en gitt statefor det første naturlige tallet, for så å bevise at å finne statetil et vilkårlig naturlig tall implisitt gir neste naturlige talls state".

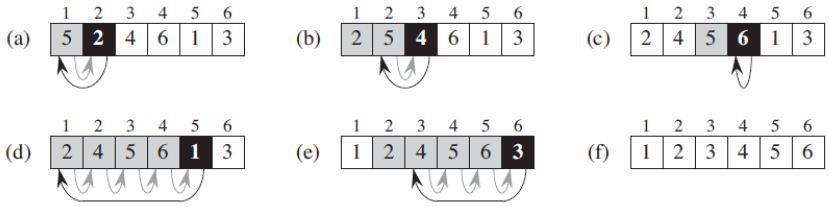
1.6 Forstå rekursiv dekomponering og induksjon over delproblemer

Et problem er forholdet mellom instanser og riktige svar, og algoritmen finner et riktig svar for hver instant. Vi kan se på en instans, og deler den opp i flere delinstanser og antar vi kan løse delinstansene. Vi samler så delsvarene til et endelig svar. Blir induktivt bevist. **Se problemløsningsguide**
Eksempel: Sorterer alle element untatt det siste rekursivt.

1.7 Forstå Insertion-Sort

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

Tar inn en usortert tabell og går gjennom fra 2.element. While loop for bytte plass mellom forrige og nåværende posisjon om forrige er større. Fortsetter helt til tabellen er sortert i stigende rekkefølge.



Figur 2: Insertion-Sort

2 Datastrukturer

Kap. 10. Elementary data structures: Innledning og 10.1–10.3

Kap. 11. Hash tables: s. 253–264

Kap. 17. Amortized analysis: Innledning og s. 463–465 (tom. «at most 3»)

NB: For statiske datasett kan worst-case for søk være $O(1)$

2.1 Forstå hvordan stakker og køer fungerer

- *Stack-Empty, Push, Pop, Enqueue, Dequeue*

2.1.1 Stakker

Hovedprinsippet er **LIFO - last in, first out"**. Elementene i en stakk stables på hverandre, og det siste elementet blir da alltid plassert på toppen. Og det eneste som er tilgjengelig for brukeren er det som ligger øverst på stakken. Feilmeldingen *stack overflow* kommer ofte hvis stakken har begrenset plass. Blir realisert med enten **array** eller **lenkede lister**
- Stack-Empty, Push, Pop.

2.1.2 Kø

Hovedprinsippet er **FIFO - first in, first out** . Elementer som blir lagt i en kø kommer ut i samme rekkefølge. Blir realisert med enten **array** eller **lenkede lister**
- Enqueue (add), Dequeue (remove)

2.2 Forstå hvordan lenkede lister fungerer

- *List-Search, List-Insert, List-Delete, List-Delete', List-Search', List-Insert'*

Består av noder som peker på neste og muligens forrige node. Det tar lineær tid å slå opp en gitt posisjon, mens det tar konstant tid å sette inn/ slette elementer.

```
List-Search(L,k)
    x = L.head
    while x != NIL and x.key != k
        x = x.next
    return x
```

```
List-Insert(L,x)
    x.next = L.head
    if L.head != NIL
        L.head.prev = x
    L.head = x
    x.prev = NIL
```

```
List-Delete(L,x)
    if x.prev != NIL
        x.prev.next = x.next
    else L.head = x.next
    if x.next  != NIL
        x.next.prev = x.prev
```

Vi bruker $'$ for å ignorere grensebetingelsene ved $head$ og $tail$.

2.3 Forstå hvordan pekere og objekter kan implementeres

Vi kan implementere objekter som feks lenkede lister i språk som ikke direkte støtter dette ved å ta i bruk en array enten av 2 dimensjoner, slik at kolonnen representerer key, $next$ og $prev$ mens radene innholder verdiene. Vi kan også bruke en 1-dimensjonal array til å representere et objekt ved å bruke $pekere$. $Pekere$ er bare minneadresses til første minnelokasjon i et objekt, og vi kan dermed adressere flere minnelokasjone ved å la det være en offset til $pekeren$.

2.4 Forstå hvordan direkte adressering og hashtabeller fungerer

Vanligvis brukes **direkte adressering** for å slå opp på element på plass nummer i i en array [konstant tid]. Det fungerer så lenge vi har råd til å ha en liste som har en posisjon for hver mulig verdi $\rightarrow$ git mange ubrukte plasser. Bruker **hashtabeller** for å unngå ubrukte plasser. I prinsippet er det relativt likt **Bucket-Sort**. I hver bøtte er ddet plass til et par posisjoner fra listen, og mange problemer er da spart ved å lage en liten **hashtabell**. Risikoen er at flere verdier kan havne i samme bøtte $\rightarrow$ kollisjoner. Bruker lenkede lister for å unngå kollisjon. **Hashtabellen** er egentlig en vanlig array, og man bruker da en hash-funksjon, h som tilordner enhver pos i listen til en pos i hashtabellen.

$$h(k) = km, \quad 0 < k < 1 \tag{1}$$

Divisjonsmetoden

$$h(k) = k\%m \tag{2}$$

hvor k er posisjonsnummer i listen, m er antallet plasser i hashtabellen (gjerne et primtall)

Multiplikasjonsmetoden

$$h(k) = [m((kA)\%1] \tag{3}$$

hvor A er et tall mellom 0 og 1. runder svaret nedover til slutt. Kjøretiden til hashing er som regel $O(1)$.

```
Hash-Insert(T,k)
    i = 0
    repeat
        j = h(k,i)
        if T[j] == NIL
            T[j] = k
            return j
        else i = i+1
    until i == m
    error "hash table overflow"
```

```
Hash-Search(T,k)
    i = 0
    repeat
        j = h(k,i)
        if T[j] == k
            return j
        i = i+1
    until T[j] == NIL or i == m
    return NIL
```

2.5 Forstå konfliktløsing ved kjeding (chaining)

Chaining blir brukt for å unngå konflikt ved hashing. For å unngå at to verdier hasher til samme indeks har hver posisjon en liste (lenket liste). Mange kollisoner vil gi lineært lange lister, og søk vil ta lineærn tid. Vi kan også anta at vi får en jevn og tilfeldig fordeling...og konstant kjøretid.

```
Chained-Hash-Insert(T,x)
    insert x at the head of list T[h(x.key)]
```

```
Chained-Hash-Search(T,k)
    search for an element with key k in list T[h(k)]
```

```
Chained-Hash-Delete(T,x)
    delete x form the list T[h(x.key)]
```

2.6 Kunne definere amortisert analyse

Mens average-case er snitt over instanser, er amortisert arbeid et snitt over operasjoner

2.7 Forstå hvordan dynamiske tabeller fungerer

For å allokere nytt minne og kopiere over data når en hashtabell, stakk eller kø er full er tidkrevende, da det tar lineær tid. Vi tar derfor i og allokerer mye minne. Dobler gjerne størrelsen for hver gang. **Se 2.likning i** (4).

```
Table-Insert(T,x)
    if T.size == 0
        allocate T.table with 1 slot
        T.size = 1
    if T.num = T.size
        allocate new-table with 2*T.size slots
        insert all items in T.table into new-table
        free T.table
        T.table = new-table
        T.size = 2*T.size
    insert x into T.table
    T.num = T.num +1
```

2.8 To kritiske summer

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}, \quad \sum_{i=0}^{n-1} 2^i = 2^n - 1 \tag{4}$$

3 Splitt og hersk

Kap. 2. Getting started: 2.3

Kap. 4. Divide-and-conquer: Innledning, 4.1 og 4.3–4.5

Kap. 7. Quicksort

Oppgaver 2.3-5 og 4.5-3 med løsning (binærsøk)

Appendiks B og C i dette heftet

3.1 Forstå designmetoden divide-and-conquer (Splitt og Hersk)

Splitt og hersk går ut på å løse et problem rekursivt ved å bruke de tre stegene;

Splitt: problemene inn i subproblem, som en mindre instanser av samme problem.

Hersk: subproblemene ved å løse dem rekursivt. Om subproblemet er lite nok kan de løses rett frem.

Kombiner så løsningene av subproblemet til en løsning av det opprinnelige problemet. Når problemet er lite nok kommer vi altså til en *base-case* og vi sier at rekursjonen *bottoms out*.

3.2 Forstå maximum-subarray-problemet med løsninger

Maximum-subararray-problemet går at på at du skal finne en sekvens $A[i..j]$ (subarray) som gir den største summen av hvilken som helst kontinuerlig rekke av verdier, gitt en array, $A[low..high]$ med tilfeldige verdier. Subarrayet ligger alltid i enten $A[low...mid]$, $A[mid+1...high)$ eller slik at den krysser midten.

Dette kan løses rekursivt med en splitt og hersk metode. Vi deler da inn i to subarray. Hvor den ene finner max subarray i $A[low...mid]$ og den andre i $A[mid+1...high)$. Vi må også sjekke en tredje subarray som krysser midten, men det er ikke et subproblem av samme problem og vi må da også dele den inn i to subarrayer hvor den ene er til venstre for midten, og den andre en til høyre, for så å legge sammen summen. Vi sjekker til slutt hvilken av de tre subarrayene som gir størst sum, og returnerer den. Kjøretid for dette problemet blir $T(n) = \Theta(nlgn)$

3.3 Forstå Bisect og Bisect' (se appendiks C i Pensumheftet)

Binærsøk går ut på å finne en ønsket verdi, v, gitt en sortert tabell. Vi halverer først tabellen, og sjekker om v er større eller mindre enn det mmidterste tallet, og deretter fortsetter vi rekursivt med enten tabellen på høyre eller venstre side, helt til posisjon på ønsket verdi er funnet.

```
1 Bisect(A,p,r,v)
2     if p >= r
3         q = [(p+r)/2]
4         if v == A[q]
5             return q
6         else if v > A[q]
7             return Bisect(A,p,q-1,v)
8         else return Bisect(A,q+1,r,v)
9     else return NIL
```

, hvor A er en sortert liste, p er startindeks, r er sluttindeks, og v er ønsket verdi.
Den iterative metoden av problemet er gitt som

```
Bisect'(A,p,r,v)
    while(p <= r)
        q = [(p+r)/2]
        if v == A[q]
            return q
        else if v > A[q]
            r = q-1
        else p = q+1
    return NIL
```

3.4 Forstå Merge-Sort

```
Merge-sort(A,p,r)
    if p<r
        q = [(p+r)/2]
        Merge-Sort(A,q+1,r)
        Merge-Sort(A,p,q)
        Merge(A,p,q,r) \\ fletter sammen listene fra v.h og h.h
```

Deler listen i 2 og sorterer deretter de to nye listene ved rekursjon. Deler da opp på nytt og nytt helt til hele listen er sortert.

3.5 Forstå Quicksort og Randomized-Quicksort

```
Quicksort(A,p,r)
    if p < r
        q = Partition(A,q,r)
        Quicksort(A,p,q-1)
        Quicsort(A,q+1,r)
```

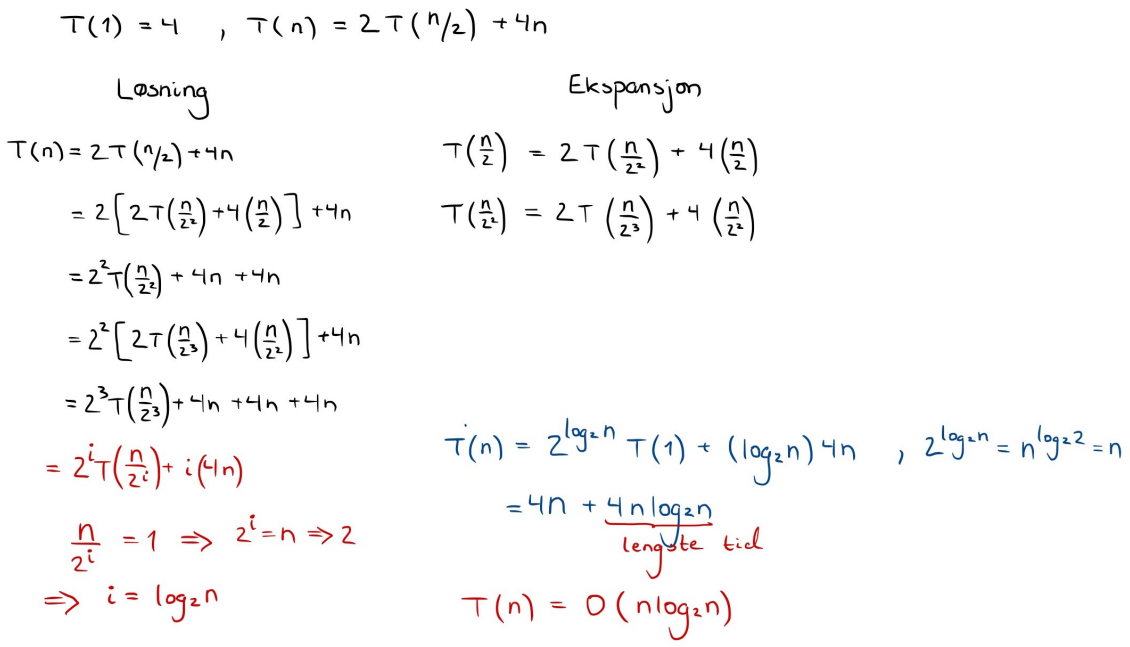
Partition velger ut et element i mengden vi skal sortere. Elementet kalles *pivotelementet*, a. Mengden blir så sortert til to nye mengder, hvor den ene er større enn a, og den andre mindre. Og quicksort blir kjørt igjen rekursivt på de to nye mengdene.

Randomized quicksort fungere på akkurat samme måte, med unntak av at *pivotelementet* tilfeldig for så å bli plassert bakerst. Deretter blir den vanlige **partition**-funksjonen kalt.

3.6 Kunne løse rekurrenser med substitusjon, rekursjonstrær og masterteoremet

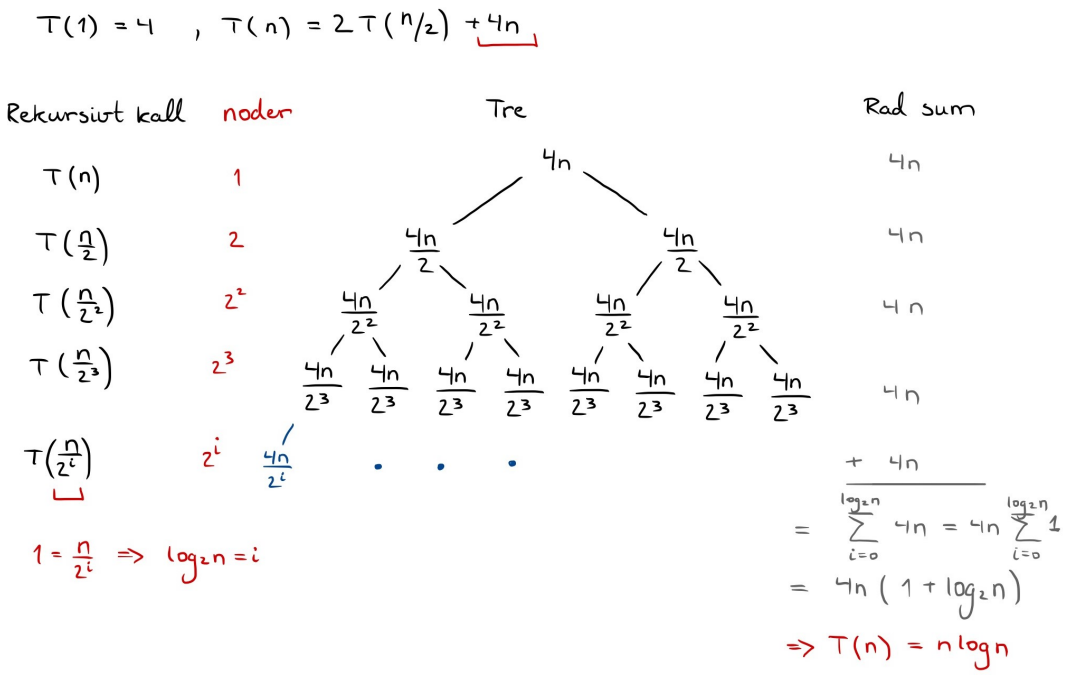
Blir brukt enten for å finne den asymptotiske kjøretiden Θ eller O-grensene for løsningen.

Substitusjonsmetoden går ut på å gjette grensene for så å bruke matematisk induksjon til å bevise det vi har antatt. Se eksempel i figur 3.



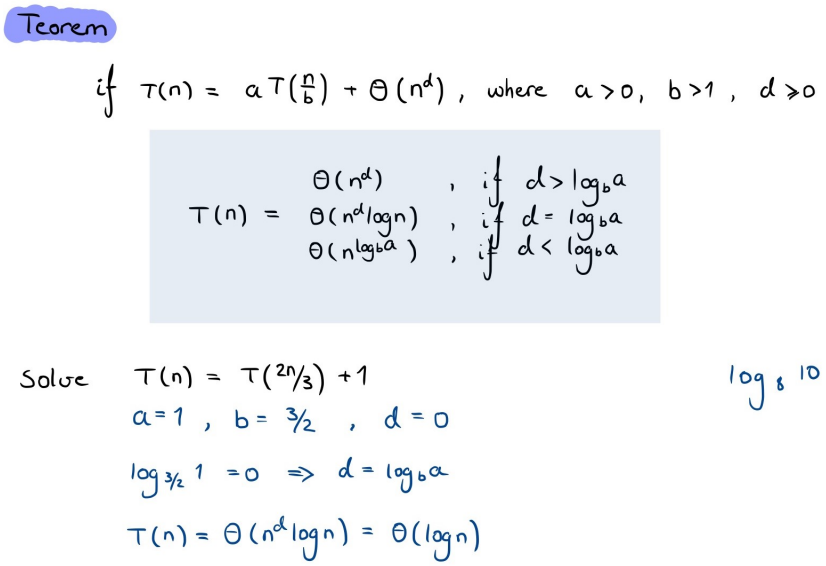
Figur 3: Eksempel på substitusjon

Rekursjonstrær konverterer en rekkurens til et tre hvor nodene representerer kostnaden som oppstår på de forskjellige nivåene av rekkurensen. Se eksempel i figur **??**.



Figur 4: Eksempel på bruk av rekursjonstre

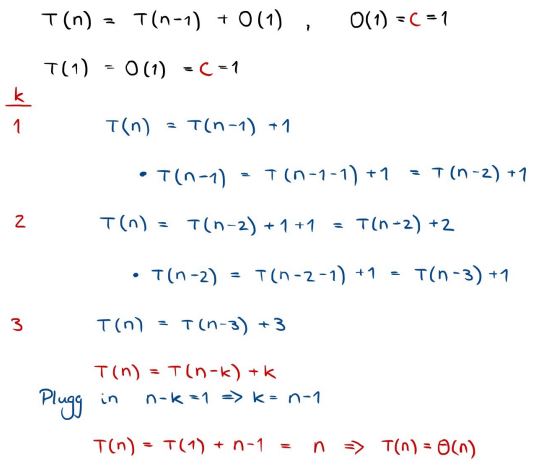
Masterteoremet gir en kjøretid på formen $T(n) = aT(n/b) + f(n)$, hvor $a \geq 1$, $b > 1$ og $f(n)$ er en gitt funksjon. Denne formen viser til et splitt og hersk problem hvor algoritmen lager seg a subproblem, hvert av problemene er da $\frac{1}{b}$ av problemet, og splitt og kombineringsstegene tar $f(n)$ tid. Se eksempel i figur **??**.



Figur 5: Eksempel på bruk av masterteorem

3.7 Kunne løse rekurrenser med iterasjonsmetoden (se appendiks B i Pensumheftet)

Iterasjonsmetoden er en mer matematisk fremgangsmåte, og er også veldig lik substitusjonsmetoden. Se eksempel i figur 5.



Figur 6: Eksempel på bruk av iterasjonsmetoden

3.8 Forstå hvordan variabelskifte fungerer

Variabelskifte vil si at man bruker matematisk manipulasjon for å gjøre en ukjent rekursjon om til noe du er kjent med. Et eksempel er;

$$T(n) = 2T(\sqrt{n}) + \log n \tag{5}$$

, vi definerer da $n = 2^{\log n}$ hvor $\log n = m$

$$T(2^m) = 2T(2^{\frac{m}{2}}) + m \tag{6}$$

,deretter kan vi definere $S(m) = T(2^m)$

$$S(m) = 2S(m/2) + m \tag{7}$$

Vi får da en rekursjon med kjøretid $O(m \log m)$

4 Rangering i lineær tid

Kap. 8. Sorting in linear time

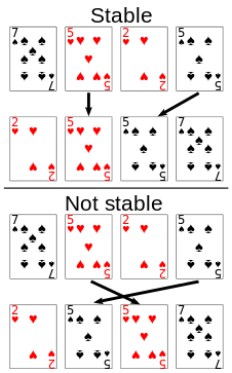
Kap. 9. Medians and order statistics

4.1 Forstå hvorfor sammenligningsbasert sortering har en worst-case på $\Omega(nlgn)$

Den lengste veien fra rota på et beslutningstre til hvilken som helst sine løvnoder representerer worst case for sammenligningsbaserte algoritmer, Det betyr da at den er lik høyden på beslutningstiden, som er lik $nlgn$. Kjøretid i worst case er derfor $\Omega(nlgn)$.

4.2 Vite hva en stabil sorteringsalgoritme er

En stabil sorteringsalgoritme betyr at likeelementer blir samlet i rett rekkefølge under sortering. Se figur 7.



Figur 7: Eksempel på stabil sorteringsalgoritme

4.3 Forstå Counting-Sort, og hvorfor den er stabil

```
Counting-Sort(A,B,k)
    let C[0..k] be a new array
    for i = 0 to k
        C[i] = 0
    for j = 1 to A.length
        C[A[j]] = C[A[j]] +1
        // C now contains the numbers of elements equal to i
    for i = 1 to k
        C[i] = C[i] + C[i-1]
        // C[i] now contains the numbers of elements <= i
    for j = A.length downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

Først blir en matrise C ititialisert med like mange = som det er forskjellige verdier i A, deretter blir antall element med samme verdi telt opp og plassert i C. Deretter blir en B matrise laget for å sortere. Det blir alle elementene lagt inn i riktig rekkefølge baseert på A og C. Og denne algoritmen er stabil siden alle element som er like blir plasser i rikitg rekkefølge.

4.4 Forstå Radix-Sort, og hvorfor den trenger en stabil subrutine

```
Radix-Sort(A,d)
    for i = 1 to d
        \\use stable sorting algorithm to sort. [countingsort, mergesort]
```

Radix-sort blir kun brukt for å sortere tall. Vi sorterer da fra **LSB** $\rightarrow$ **MSB** ved hjelp av en stabil sorteringsalgoritme for å holde på den riktige rekkefølgen på elementene.

4.5 Forstå Bucket-Sort

```
Bucket-Sort(A)
    let B[0..n-1] be a new array
    n = A.length
    for i = 0 to n-1
        make B[i] an empty list
    for i = 1 to n
        insert A[i] into list B[nA[i]]
    for i = 0 to n-1
        sort list B[i] with insertion sort
    concetanate the lists B[0],B[1],...,B[n-1] together in order
```

Bucketsort starter med å lage en array B med like mange elementer som er i A. Deretter blir hvert element lagt i en bøtte i sin respektive posisjon i B gitt av $nA[i]$. Alle bøttene blir sortert vha insertion sort, og deretter kombineres alle bøttene. $A[i]\epsilon[0,1]$

4.6 Forstå Randomized-Select

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

Randomized-Select blir brukt for å finne det i'ende minste elementet i arrayet A. Bruker da **Randomized-Partition** og sjekker om tallet vi skal finne ligger på v.h, h.h eller om det er likt pivotelementet, og deretter kjøres det rekursivt.

4.7 Kjenne til Select

Select fungerer på samme måte som **Randomized-Select**, men her er ikke pivotelementet lenger tilfeldig valgt, men det blir regnet ut et godt pivotelement.

5 Rotfaste trestrukturer

Kap. 6. Heapsort

Kap. 10. Elementary data structures: 10.4

Kap. 12. Binary search trees: Innledning og 12.1–12.3

- Forventet høyde for et tilfeldig tre er $\Theta(lgn)$
- Det finnes søketre med garantert høyde på $\Theta(lgn)$

5.1 Forstå hvordan heaps fungerer, og hvordan de kan brukes som prioritetskøer

- Parent, Left, Right, Max-Heapify, Build-Max-Heap, Heapsort, Max-Heap-Insert, Heap-Extract-Max, Heap-Increase-Key, Heap-Maximum. Også tilsvarende for minheaps, f.eks., Build-Min-Heap og Heap-Extract-Min

En max-heap består av noder, med barnenoder. Barnenodene er skal alltid være mindre eller lik foreldrenoden, mens i min heap blir det omvendt. hvor venstre barnenode er gitt $Left(i) = 2i$, høyre barnenode er gitt $Right(i) = 2i + 1$, og foreldrenoden som $Parent(i) = \frac{i}{2}$. *Build max/min heap* lager en heap bassert på en usortert tabell. En heap kan brukes som en prioritetskø, hvor hver node presenterer et element med en viss prioritet, gitt av dens key. Vi kan da legge til å fjerne elementer fra køen, og si hvilken posisjon den skal ha i køen.

5.2 Forstå Heapsort

```
1 Heapsort(A)
2     Build-Max-Heap(A)
3     for i = A.legth to 2
4         exchange A[1] with A[i]
5         A.heap-size = A.heap-size - 1
6         Max-Heapify(A)
```

Her blir det laget en max-heap av en mengde A, og deretter blir det største elementen byttet ut med den minste og *Max-Heapify* blir kjørt før det samme skjer igjen, helt til alle elementene er sortert.

5.3 Forstå hvordan rotfaste trær kan implementeres

Rotfaste trær kan implementeres på flere måter. Vi kan implementere det slik at hver node peker til foreldre og sine barn, og det tar da $O(1)$ tid å finne både sine foreldre og barn. En annen måte en å peke til foreldre, barn, og sin nærmeste søsken, og det tar da $O(1)$ tid å finne foreldre, mens det tar $O(i)$ tid å finne sitt barn, i. Vi kan også implementere det i en array, hvor indeksen er barna og elementene representerer foreldrene. En siste måte er å lagre nodene som array, hvor første element er verdien til noden, og neste element er peker til barna.

5.4 Forstå hvordan binære søketrær fungerer

Et binært tre er et tre hvor hver node har maks to barn. Nodene peker til begge barna og forelderen. I et binært søketre er venstre barn mindre eller lik verdien til forelderen, mens høyre barn er større.

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

Printer ut hele det binære søketreet.

```
Tree-Search(x,k)
    if x == NIL or k == x.key
        return x
    if k < x.key
        return Tree-Search(x.left,k)
    else return Tree-Search(x.right,k)
```

Går gjennom treet helt til vi finner verdien vi vil ha, og returnerer noden når vi har funnet rett.

```
Iterative-Tree-Search(x,k)
    while x != 0 and k != x.key
        if k < x.key
            x = x.left
        else x = x.right
    return x
```

Samme som **Tree-Search**, men uten rekursive kall.

Tree-Minimum går ned langs venstre side helt til den kommer til siste element, og returnerer den.

Tree-Maximum går gjennom langs høyre side helt til den kommer til siste element, og returnerer den.

Tree-Successor finner den noden med den minste verdien mindre enn x.

Tree-Predecessor finner den noden med den største verdien mindre enn x.

```
Tree-Insert(T,z)
    y = NIL
    x = T.root
    while x != NIL
        y = x
        if z.key < x.key
            x = x.left
        else x = x.right
    z.p = y
    if y == NIL
        T.root = z \\ Tree was empty
    else if z.key < y.key
        y.left = z
    else y.right = z
```

Går gjennom treet og finner rett posisjon å sette inn z.

Transplant gjår det mulig å flytte rundt på subtre.

Delete gjør det mulig å slette en node.

6 Dynamisk programmering

Kap. 15. Dynamic programming: Innledning og 15.1, 15.3–15.4

Oppgave 16.2-2 med løsning (0-1 knapsack)

Appendiks D i dette heftet

Delkapitler 15.2 og 15.5 kan være nyttige som støttelitteratur.

6.1 Forstå ideen om en delproblemgraf

En delproblem graf er brukt til å indikere avhengigheter mellom de forskjellige delproblemene. Hver node i grafen representerer et eksplisitt delproblem, mens kantene mellom nodene indikerer avhengigheter.

6.2 Forstå designmetoden dynamisk programmering

Dynmaisk programmering blir brukt når subproblem overlapper, altså når subproblem deler subsubproblem. Dynamisk programmering løser da subsubproblemene kun én gang og lagrer svaret i en tabell, slik at den kan henter frem senere. Fremgangsmåten for dynamisk programmering er gitt;

- Karakteriser strukturen til en optimal løsning
- Definer verdien til en optimal løsning rekursivt
- Kalkuler verdien til en optimal løsning, gjerne ved bottom-up
- Konstruer en optimal løsning fra den kalkulerte informasjonen

6.3 Forstå løsning ved memorisering (top-down)

Ved memorisering skrives prosedyren rekursivt, men modifisert slik at resultatet til hvert subproblem blir lagret. Først blir det sjekket om det finnes en løsning på det første subproblemet. Visst ikke blir det regnet ut på vanlig måte. Visst den derimot eksisterer blir den utregnede verdien returnert. Vi sier da at den rekursive prosedyren husker hvilke resultater som har blitt regnet ut.

6.4 Forstå løsning ved iterasjon (bottom-up)

Ved iterasjon avhenger fremgangsmåten gjerne av strørrelsen på subproblemet, slik at å løse et subploblem avhenger av å løse et mindre subproblem. Vi sorterer subproblemene etter størrelse, med minst først. Vi løser så subproblemene fra minst til størst. Og de større problemene blir da lettere siden de mindre problemene allerede er løst.

6.5 Forstå hvordan man rekonstruerer en løsning fra lagrede beslutninger

Ved å lagre beslutningene som ble tatt for å komme frem til en verdi for subproblemet kan vi rekonstruere en løsning ved å printe ut beslutningen helt til vi ikke har flere delproblem.

6.6 Forstå hva optimal delstruktur er

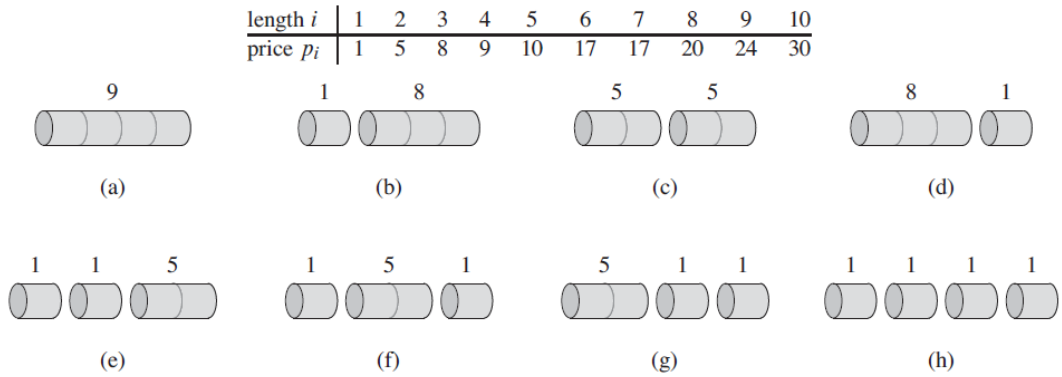
Et problem sies å ha en optimal delstruktur om en optimal løsning kan konstrueres for problemet basert på optimale løsninger for subproblemene.

6.7 Forstå hva overlappende delproblemer er

Overlappende delproblem er delproblem som deler subsubproblem.

6.8 Forstå eksemplene stavkutting og LCS

Stavkutting eksempelet går ut på at at man skal maksimere prisen på stålstaver gitt en ønsket total lengde. Vi ser da på hvordan vi kan kutte opp staven slik at den totale prisen blir høyest mulig. Hvert kutt medfører et subproblem der vi må finne en optimal løsning for den nye størrelsen. Se eksempel i figur 8.



Figur 8: Stavkutting ved i = 4. Vi ser her at den optimale løsninger er gitt av (c)

LCS - 'Longest common subsequence' går ut på å ta inn to sekvenser av verdier, ene av lengde m, og den andre med lengde n. Vi lager deretter to matriser på MxN. Vi går så gjennom sekvensene med to for løkker og lagrer de forskjellige subsekvense, for så å se hvilken som er lengst.

6.9 Forstå løsningen på 0-1-ryggsekkproblemet (se appendiks D i Pensumheftet)

```
1 Knapsack(n,W)
2     if n == 0
```

```
        return 0
    x = Knapsack(n-1,W)
    if w_n > W
        return x
    else y = Knapsack(n-1,W-w_n) + v_n
        retutn max(x,y)
```

0-1 Ryggsekkproblemet gått ut på at man enten tar med seg en gjenstand eller lar den ligge. Vi antar at vi har en sekk som kan holde en vekt, W. Og vi har n mulige gjenstander å ta med. Om vi velger å plukke opp en gjenstand, kjører vi rekursivt med ny vekt $W - w_n$, og legger til prisen v_n til slutt. Om vi lar gjenstanden ligge igjen kjører vi rekursivt med $n - 1$ gjenstander, og samme vekt. Til slutt sjekker vi har vi sjekket alle muligheter, og returnerer den kombinasjonen som gir mest verdi.

```
Knapsack'(n,W)
    let K[0..n,0..W] be a new table
    for j = 0 to W
        K[0,j] = 0
    for i = 1 to n
        for j = 0 to W
            x = K[i-1,j]
            if j < w_i
                K[i,j] = x
            else y = K[i-1, j-w_i] + v_i
                K[i,j] = max(x,y)
```

Knapsack' gjør det samme som **Knapsack** bare iterativt istedenfor rekursivt.

7 Grådige algoritmer

Kap. 16. Greedy algorithms: Innledning og 16.1–16.3

- Har optimal delstruktur, men ikke nødvendigvis overlappende delproblem

7.1 Forstå designmetoden grådighet

Fremgangsmåten for å designe en grådig algoritme er som følger;

1. Si at optimeringsproblemet er et problem som står igjen med et subproblem om vi tar det grådige valget.
2. Bevis at der alltid er en optimal løsning til det originale problemet som tar det grådige valget, slik at det grådige valget er trygt å ta.
3. Demonstrer optimal delstruktur ved å vise at, ved å ta det grådige valget, er det som er igjen et subproblem med den egenskapen at visst vi kombinerer en optimal

løsning til subproblemet med det grådige valget vi har tatt, så får vi en optimal løsning til problemet.

7.2 Forstå grådighetsegenskapen (the greedy-choice property)

Grådighetsegenskapen går at på at vi nå tar valgene mye lettere enn vi gjorde ved dynamisk programmering. Nå tar vi det valget som skiller seg klart ut. Og vi kan da ta det valget hver gang uten å vurdere de andre. **Grådighetsegenskapen** betyr med andre ord at et valg som er lokalt optimalt er en del av den globalt optimale løsningen.

7.3 Forstå eksemplene aktivitet-utvelgelse og det fraksjonelle ryggsekk-problemet

Aktivitetsutvelgelsen går ut på at vi har et sett, $S = a_1, a_2, ..., a_n$ med n forskjellige aktiviteter med starttid s_n og slutttid f_n. Ved å bruke en grådig algoritme vil vi velge den aktiviteten som gjør at ressursen er ledig lengst mulig når aktiviteten er ferdig. Og det gjøres ved å velge aktiviteten som er ferdig tidligst. Så kjører vi rekursivt med det nye intervallet av ledig tid.

Det fraksjonelle ryggsekkproblemet er det samme som **0-1 ryggsekproblemet**, bare at her kan vi ta med deler av de forskjellige gjenstandene. Den grådige løsningen er da å ta med mest av det som har størst verdi. Optimal delstruktur blir fylt ved at visst vi legger noe i sekken, må forstatt resten av sekken fylles optimalt.

7.4 Forstå Huffman og Huffman-koder

```
1 Huffman(C)
2     Q = C
3     for i = 1 to n-1
4         allocate a new node z
5         z.left = x = Extract-Min(Q)
6         z.right = y = Extract-Max(Q)
7         z.freq = x.freq + y.freq
8         Insert(Q,z)
9     return Extraxt-Min(Q)
```

Huffman(C) tar inn ett sett, C med bokstaver. Vi vil så lage en kode hvor de bokstavene med høy frekvens har korte koder, og de med lav frekvens får lengre koder. Vi begynner med å legge bokstavene i en min-kø, Q. Deretter tar vi ut de to med minst frekvens og legger de til som barna til z og summerer opp frekvensen, og legger de bakerst i køen.

Huffman-koder går ut på å komprimere data. Vi har da en variabel lengde på kode-bitene. De hyppigst brukte bokstavene får korte koder og vice versa.

8 Traversering av grafer

Kap. 22. Elementary graph algorithms: Innledning og 22.1–22.4

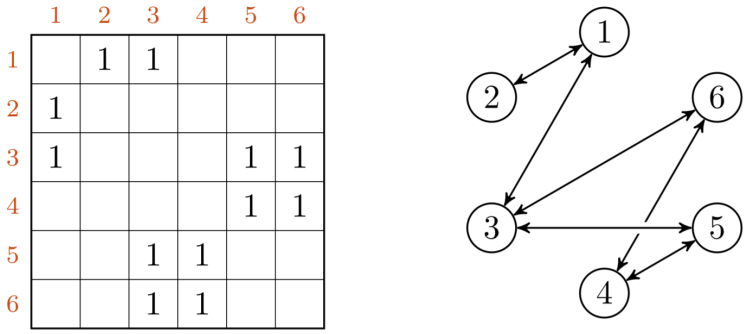
Appendiks E i Pensumheftet

- **DAG** (Directed Acyclic Graph): En rettet graf som ikke inneholder noen sykler $\rightarrow$ kan sorteres topologisk.

8.1 Forstå hvordan grafer kan implementeres

Grafer kan enten implementeres som enten nabolister eller nabomatriser.

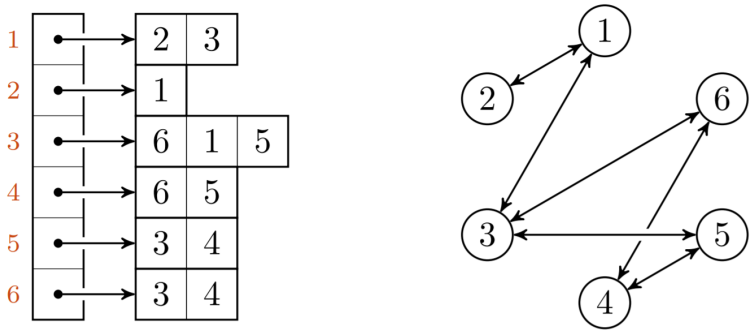
Nabomatriser er en matrise som gjengir avhengiheten mellom forksjellige noder i en graf, og egener seg godt til oppslag, men ikke så godt til traversering. Se figur 9.



Figur 9: Nabomatrise

I en urettet graf vil matrisen $A[u, v]$ være symmetrisk.

Nabolister er en liste (eller tabell) med ut-naboer for hver node. Det er altså en liste med en liste for hver node.



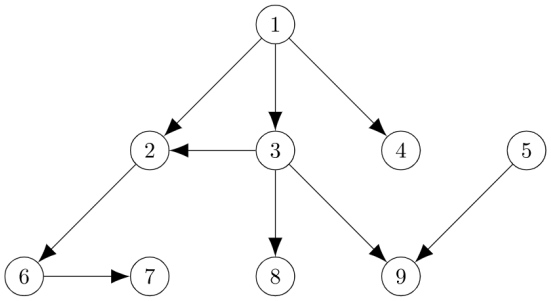
Figur 10: Naboliste

Nabolister er godt egnet for traversering, men ikke så egnet for oppslag. En urettet graf

består av kanter som går begge veier. Vi har en sykel i en rettet graf dersom vi kan starte fra en node og komme tilbake til den gjennom andre noder.

8.2 Forstå BFS, også for å finne *korteste vei uten vekter*

BFS går ut på at vi utforsker grafen i bredden, altså at vi utforsker alle kantene ut fra den noden vi står på før vi beveger oss videre. BFS representeres ved hjelp av en **kø** (FIFO). BFS egener seg godt for å finne ut hvor mange kanter som er nødvendig for å komme seg fra startnoden til de andre nodene. Ved å bruke en **kø** sikrer vi at vi alltid for den koreste veien til nodene.



Figur 11: Eksempel

Figur 11 gir følgende output;

1. Oppdaget: 1(0); Ferdig: -
2. Oppdaget: 2(1),3(1),4(1); Ferdig; 1(0)
3. Oppdaget: 6(3), 8(3), 9(3); Ferdig: 1(0), 2(1), 3(1), 4(1)
4. Oppdaget: 7(4); Ferdig: 1(0), 2(1), 3(1), 4(1), 6(3), 8(3), 9(3)
5. Ferdig: 1(0), 2(1), 3(1), 4(1), 6(3), 8(3), 9(3), 7(4)

Vi får alltid ut en liste sortert etter avstand.

8.3 Forstå DFS og parentesteoremet

DFS går ut på å utforske en graf i dybden. Det betyr at neste kant du skal utforske, går fra den noden du sist oppdaget. Dersom noden ikke har noen kanter som går til noder du ikke har oppdaget før, går du tilbake til forrige node og gjør det samme der. DFS prinsippet representeres som en **stakk** (LIFO). Noden som ligger øverst i stakken er alltid den som skal utforskes, og om du møter på nye noder, legger de øverst i stakken. Når du møter flere noder som går ut fra en node kan man i prinsippet velge hvilen som helst først. Til slutt når alle nodene du når ut til er sjekket, legges de nodene som ikke er koblet direkte til. **Parantesteoremet** vil si at om du har en node v med en etterkommer u, vil oppdagelsestiden være kortest for v, men tiden før den er ferdig vil være kortest for

u.
Ved å kjøre **DFS** på grafen i figur 11 får vi;

1. Oppdaget: 1; Ferdig: -
2. Oppdaget: 1,2; Ferdig: -
3. Oppdaget: 1,2,6; Ferdig: -
4. Oppdaget: 1,2,6,7; Ferdig -
5. Oppdaget: 1,3,8; Ferdig: 7,6,2
6. Oppdaget: 1,3,9; Ferdig: 7,6,2,8
7. Oppdaget: 1,4; Ferdig: 7,6,2,8,9,3
8. Oppdaget: - ; Ferdig: 7,6,2,8,9,3,4,1,5

8.4 Forstå hvordan DFS klassifiserer kanter

Tre-kanter er kanter i DFS.
Fremoverkanter er kanter til en forgjenger i DFS.
Bakoverkant er kanter utenfor DFS til er etterkommer i DFS.
Kryss-kanter er alle andre kanter.

8.5 Forstå Topological-Sort

For å forstå topologisk sortering kan vi si at nodene representerer oppgaver som skal gjøres. Noen oppgaver må gjøres før andre. Topologisk sortering går ut på å finne en mulig rekkefølge oppgavene kan gjøres, og tar utgangspunkt i en **DAG**. Algoritmen for topologisk sortering er ganske enkel. Vi tar utgangspunkt i **DFS** og reverserer så lista når vi er ferdig.

Algoritme	WC	BC
Dybde-først-søk	$\Theta(V + E)$	$\Theta(V + E)$
Bredde-først-søk	$\Theta(V + E)$	$\Theta(V)$
_____-først-søk	$\Omega(V + E)$	

Figur 12: Kjøretid

9 Minimale spenntre

Kap. 21. Data structures for disjoint sets: Innledning, 21.1 og 21.3

Kap. 23. Minimum spanning trees

- Erke-eksempel på grådighet; Velg én og én kant, alltid den billigste lovlige.

9.1 Forstå skog-implementasjonen av disjunkte mengder

- Connected-Components, Same-Component, Make-Set, Union, Link, Find-Set

Mengder representeres som trær ved hjelp av foreldrepekere $v.p$. Rota av treet representerer mengden, og **Find-Set(v)** git rotnoden. **Make-set(v)** lager et nytt sett der det eneste elementet er v. Siden settene er disjunkte krever vi at v ikke er i noen andre sett. **Union(v,u)** kobler sammen de to dynamiske settene som inneholder v og u, og vi antar at settene er disjunkte før operasjonen. *Rank* er øvre grense for høyden til en node.

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

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

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

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

9.2 Vite hva spenntrær og minimale spenntrær er

Et **spenntre** er en subgraf av en graf G som inneholder alle nodene til G og er et tre. Et tre med V noder har $V - 1$ kanter.

Et **Minimalt spenntre** (MST), av en vektet graf G er et spenntre av G som er slik at summer av kostnadene til kantene som inngår i treet er minimal (Ikke mulig å lage andre spenntre med mindre kantkostnad). Finnes flere MST av samme graf.

9.3 Forstå Generic-MST

Delgrafer er grafer som består av delmengder av nodene og kantene til den opprinnelige grafen.
Spenngraf eller dekkende delgraf er en delgraf med det samme nodesettet som originalgrafen.
Spennskoger eller dekkende skog er en asyklisk spenngraf.
Spenntrær er en sammenhengende spennskog.

```
Generic-MST(G,w)
    A = NILL
    while A does not form a spenning tree
        find an edge (u,v) that is safe for A
        A = A U{(u,v)}
    return A
```

Det er nå innført vekter på kantene, lengder/kostnader. OG vi vil knytte sammen nodene på billigst måte. Vi gir inn en urettet graf $G(V, E)$ og en vektfunksjon $w : E \to \mathbb{R}$. Og vi får ut en asyklisk delmengde. Deretter utvider vi en kantmengde gradvis. Hvis en kantmengde utgjør en del av et mindre spenntre sier vi den er **invariant**. Og en trygg kant er en kant som bevarer invarianten. **En lett kant** er en slags imaginær kant som kunne vært der, og det vil være nøyaktig en sti fra nodene som er i den lette kanten. For å velge rett snitt kan vi bruke to forskjellige algoritmer; enten **Kruskals** eller **Prims**

9.4 Forstå hvorfor lette kanter er trygge kanter

Lette kanter er trygge kanter fordi den respekterer løsningen vår og tar vare på invarianten.

9.5 Forstå MST-Kruskal

Sorterer alle kantene etter kostnad, og ser på kantene i stigende rekkefølge. Ta med en kant i spenntreet med mindre kanten ville ha lagd en sykel. Fortsett til ikke flere kanter kan legges til.

```
MST-Kruskal(G,w)
    A = NILL
    for each vertex v in G:V
        Make-Set(v)
    sort G.E bt w
    for edges (u,v) in G.E
        if Find-Set(u != Find-Set(v)
```

```
8                 A = A U{(u,v)}
9                 Union(u,v)
10        return A
```

Kjøretiden er; $O(ElgV)$.

9.6 Forstå MST-Prim

Bygger er tre gradvis; en lett kant over snittet rundt treet er alltid trygt. I hvert steg ser man på alle kanter som forbinder en node som er med i treet man hittil har bygd med en node som ikke er med, og velger den kanten med minst kostnad $\rightarrow$ prioritetskø. Når alle nodene er med, har vi funnet et minimalt spenntre.

```
1  MST-Prim(G,w,r)
2      for each u in G.V
3          u.key = inf
4          u.pi = NIL
5      r.key = 0
6      Q = G.V
7      While  Q != NILL
8          u = Extract-Min(Q)
9          for each v in G.Adj[u]
10             if v in Q and w(u,v) < v.key
11             v.pi = u
12             v.key = w(u,v)
```

Kjøretid er; $O(ElgV)$

10 Korteste vei fra én til alle

Kap. 24. Single-source shortest paths: Innledning og 24.1–24.3

10.1 Forstå ulike varianter av korteste-vei- eller korteste-sti-problemet

- Single-source, single-destination, single-pair, all-pairs

Single source går ut på at vi gitt en graf $G = (V, E)$ skal finne den korteste veien fra en gitt kilde vertex, s til hver vertex, v i grafen. Algoritmen for single source kan også brukes for å løse problemene nedenfor.

Single destination går ut på å finne korteste veien gitt destinasjons vertex, f fra hver vertex, v i grafen. Ved å snu på alle kantene i grafen, kan vi løse det med *single source*.

Single pair går ut på å finne korteste vei fra u til v, gitt vertxene u og v. Vi kan løse dette ved single source ved å ha u som start node.

All pair går ut på å finne korteste vei fra u til v for hvert par av vertexer u og v. Vi kan løse det med single source ved å kjøre med startnode u for alle parene.

10.2 Forstå strukturen til korteste-vei-problemet

Korteste vei algoritmer avhenger vanligvis av den egenskapen at den korteste veien mellom to vertexer inneholder andre korteste veier innad. Optimal substruktur er den viktigste indikatoren på at dynamisk programmering og den grådige metoden fungerer.

10.3 Forstå at negative sykler gir mening for korteste enkle vei (simple path)

Negative sykler gir mening for korteste enkle vei da de gir en kort vei, men vil ikke gi mening for korteste vei da det fins uendelig mange korte veier. 0-vekt sykler fjeres for hver gang helt til vi står igjen med en enkel vei.

10.4 Forstå at korteste enkle vei kan løses vha. lengste enkle vei og omvendt

Lengste enkle vei problemet er NP-hardt men kan løses ved korteste vei av $-G$.

10.5 Forstå hvordan man kan representere et korteste-vei-tre

Vi kan representere et korteste vei på nesten samme måte som BFS, hvor vi har med forgjengere og etterkommere, altså tar med vertexene imellom.

10.6 Forstå kant-slakking (edge relaxation) og Relax

Kanslakking går ut på først gi vertex v et estimat, for så å sjekke om det stemmer med $u.d + w(u, d)$. Visst det ikke stemmer så endrer vi til $v.d = u.d + w(u, v)$.

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

Vi bruker *Initialize-Single-Source* først for å initialisere $v.d$ og $v.pi$.

10.7 Forstå ulike egenskaper ved korteste veier og slakking

Triangle inequality betyr at for hvilken som helst kant (u, v) har vi at **korteste vei vekten**, $\delta(s, v) \leq \delta(s, u) + w(u, v)$

Upper bound property vil si at vi alltid har $v.d \geq \delta(s, v)$ for alle vertexer, viV, og at med en gang $v.d$ får verdien $\delta(s, v)$, vil den aldri endre seg.

No path property betyr at om det ikke er noen veier fra s til v, så vil vi alltid ha $v.d = \delta(s,v) = \inf$

Convergence property: Hvis $s \to u \to v$ er den korteste veien i G for en u, viV og hvis $u.d = \delta(s,u)$ ved hvilket som helst tidspunkt før kanslakkingen av (u,v), da vil $v.d = \delta(s,v)$ for alltid etter det.

Path relaxation property: Hvis $p =< v_0, v_1, ..., v_k >$ er en korteste vei fra $s = v_0$ til v_k, og vi kantslakker kantene til p i rekkefølgen; $(v_0,v_1),(v_1,v_2),...,(v_{k-1},v_k)$ så vil $v_k d = \delta(s,v_k)$. Denne egenskapen holder uavhengig av andre kantslakkinger som oppstår, selv om de blandes med slakkingen av p.

Predecessor subgraph property: Når $v.d = \delta(s,v)$ for alle viV, vil forgjengerer subgrafen være en korteste vei med rot i s.

10.8 Forstå Bellman-Ford

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

Bellman-Ford algoritmen løser korteste vei problemet også der vektene er negativ. Den returnerer altså om grafen G inneholder en negativ sykel eller ikke. Visst det ikke er en negativ sykel blir den koreste veien produsert. Kjøretiden er $O(VE)$.

10.9 Forstå DAG-Shortest-Path

```
1 DAG-Shorest-Paths(G,w,s)
2     topilogically sort the vertices of G
3     Initialize-Single-Source(G,s)
4     for each vertex u, taken in topologically sorted order
5         for each vertex v in G.Adj[u]
6             Relax(u,v,w)
```

DAG-Shortest-Path sorterer først alle vertexene topologisk, så går vi gjennom den topologisk sorterte mengden og slakker kantene $\to$ Korteste vei. Kjøretiden til algoritmen er $\Theta(V+E)$.

10.10 Forstå kobling mellom DAG-Shortest-Path og dynamisk programmering

DAG-Shortest-Path er et godt eksempel på dynamisk programmering, da avstanden fra s til innnaboer er delproblemer, også velger vi den som gir best resultat. Vi løser problemet ved **bottom-up** ved kantslakking av inn-kanter i topologisk sortert rekkefølge (*pulling)*.

10.11 Forstå Dijkstra

```
Dijkstra(G,w,s)
    S = NILL
    Q = G.V
    while Q != NILL
        u = Extract-Min(Q)
        S = S U{u}
        for each vertex v in G.Adj[u]
            Relax(u,v,w)
```

Dijkstra løser korteste vei problemet gitt en rettet graf $G(V, E)$ hvor alle vektene er ikke-negativ. Her har vi et sett S som tar vare på alle de korteste veiene som allerede er bestemt, og vi har kortere kjøretid enn **Bellman-Ford**. Q er en minimum prioritetskø. Kjøretiden er $O(ElgV)$ når alle vertexene kan nåes fra start vertexen.

11 Korteste vei fra alle til alle

Kap. 25. All-pairs shortest paths: Innledning, 25.2 og 25.3

11.1 Forstå forgjengerstrukturen for alle-til-alle-varianten av korteste vei-problemet

- Print-All-Pairs-Shortest-Path

For å løse korteste vei problemet fra alle til alle, må vi ikke bare finne de korteste veiene, men også en forgjengermatrise, $\Pi = (\pi_{ji})$ hvor $\pi = NIL$ enten om $i = j$ eller hvis det ikke er en vei fra i til j. Ellers er π_{ij} forgjengeren til j på en korteste vei fra i. Rota i matrisen en i.

```
Print-All-Pairs-Shortest-Path(PI, i, j)
    if i == j
        print i
    elseif pi_ij == NIL
        print no path from i to j exists
    else Print-All-Pairs-Shortest-Path(PI,i,pi_ij)
```

11.2 Forstå Floyd-Warshall

```
Floyd-Warshall(W)
    n = W.rows
    D^(0) = W
    for k = 1 to n
        let D^(k) = (d_(ij)^(k)) be a new n x n matrix
        for i = 1 to n
            for j = 1 to n
                d_(ij)^(k) = min( d_(ij)^(k-1),  d_(ik)^(k-1) + d_(kj)^(k-1)
    return D^(n)
```

Tar inn vektene W. Vi initialiserer først avstandsmatrisen D, og ser på veien fra i til j som går gjennom $1, ...k-1$ Også en metode som bruker dynamisk programmering. Og kjører på $\Theta(V^3)$. Om vi ser at k er en del av veien kan vi si at alle vertexene i veien er med i settet $1, 2, ..., k-1$. Visst ikke dekomponerer vi veien p til $i \xrightarrow{p1} k \xrightarrow{p2} j$. Negative sykler er tillatt. Vi går altså gjennom for-loopen og ser hvilen av de to avstandene som er minst og legger til i matrisen D.

11.3 Forstå Transitive-Closure

```
Tranitive-Closure(G)
    n = |G.V|
    let T^0 = (t_ij) be a new n x n matrix
    for i = 1 to n
        for j = 1 to n
            if i == j or (i,j) in G.E
                t_ij = 1
            else t_ij = 0
    for k = 1 to n
        let T^(k) = t_ij^(k) be a new n x n matrix
        for i = 1 to n
            t_ij^k = T_ij^(k-1) OR (t_ik^(k-1) AND t_kj^(k-1))
    return T^n
```

Vi prøver her å finne ut om G inneholder en vei fra alle i til j. En måte å gjøre det på er å gi en vekt på 1 til alle kantene E, og kjøre **Floyd-Warshall**. Hvis det er en vei fra i til j får vi at $d_{ij} < n$, og $d_{ij} = \inf$ hvis ikke. En annen måte er å bruke OR og AND istedenfor min og $+$ i $Floyd - Warshall$.

11.4 Forstå Johnson

```
Johnson(G,w)
    compute G', where G'.V = G.V U{s},
```

```
        G'.E = G.E U{(s,v): v in G.V}, and
        w(s,v) = 0 for all v in G.V
    if Bellman-Ford(G',w,s) == False
        print the input graph contains a negative weight cycle
    else for each vertex v in G'.V
        set h(v) to the value of delta(s,v)
        computed by Bellman-Ford
        for each edge (u,v) in G'.E
            *w(u,v) = w(u,v) + h(u) - h(v)
        let D = d_uv be a new n x n matrix
        for each vertex u in G.V
            run Dijkstra(G,*w,u) to compute *delta(u,v) av alle v in G.V
            for each vertex v in G.V
                d_uv = *delta(u,v) + h(v) - h(u)
    return D
```

Kjøretiden for **Johnson** er $O(V^2 lgV + VE)$. Vi tar her inn en vektet, retter graf uten negative sykler. Outputten vi får er en nxn matrise D med avstander. Vi tilordner en verdi $h(u)../..h(v)$ til hver node, og vekten okes med differansen mellom dem, og positive og negative legg vil da oppheve hverandre med unntak av første og siste ledd. Til slutt rekonstruerer vi lengden ved å trekke fra den differansen, og legger til en ny node s for å sikre at vi når alle noder. Vi verken innfører eller fjerner negative sykler.

12 Maksimal flyt

Kap. 26. Maximum flow: Innledning og 26.1–26.3

12.1 Kunne definere flytnett, flyt og maks-flyt-problemet

Flytnett er en rettet graf gitt som, $G = (V, E)$. I flynettet er det ingen *self-loops*. Kapasiteter, $c(u, v) \geq 0$ Vi har en kilde, s, og et sluk, t som begge er mengder av V.

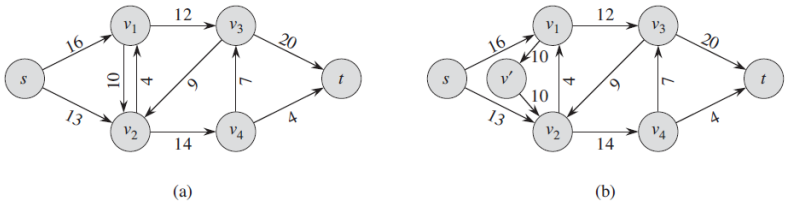
Flyt er en funksjon definert som $f : VxV \rightarrow \mathbb{R}$ som oppfyller $0 \leq f(u, v) \leq c(u, v)$.

Flytverdi er gitt som $|f| = \sum_v f(s, v) - \sum_v f(s, v)$

Maks-flyt-problemet: En analogi som kan brukes for å skjønne maks-flyt-problemet er å tenke på det som et system hvor vi har en kilde hvor det kommer ut en væske som skal fraktes til sluket. Kantene fungerer som rør, med en gitt gjennomstrømmingshastighet, og det samme har kilden og sluket. Vi ser på vertexene som skjøter mellom rørene. Vi skal da finne ut den maksimale strømningshastigheten væsken kan ha gjennom systemet.

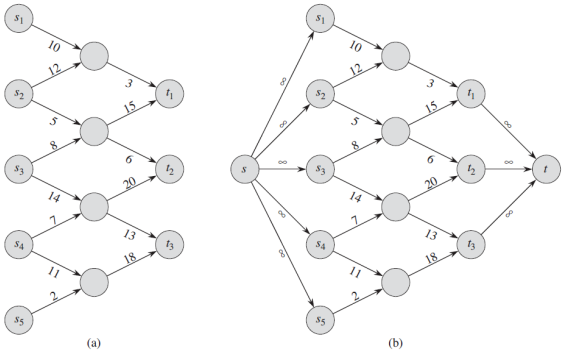
12.2 Kunne håndtere antiparallelle kanter og flere kilder og sluk

For å håndtere **antiparallelle** kanter splitter vi den ene med en node. Se figur **??** for eksempel.



Figur 13: Eksempel på håndtering av antiparalleller

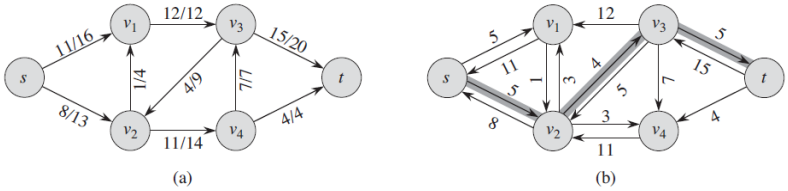
Flere kanter kilder og sluk kan håndterer ved å innføre supersluker og superkilder med ∞ kapasitet, c. Se figur 14 for eksempel.



Figur 14: Eksempel på håndtering flere kilder og sluk

12.3 Kunne definere restnettet til et flytnett med en gitt flyt

Restnett eller residualnettverk går ut på å legge til fremoverkanter ved ledig kapasitet i en kant, og bakoverkant ved fly. Se figur **??**



Figur 15: Eksempel på restnett

12.4 Forstå hvordan man kan oppheve (cancel) flyt

Oppheving av flyt er enkel matematikk. Si at vi sender 5 pakker fra u til v, og 3 pakker fra v til u. Vi kan da tilsvarende bare sende 3 pakker fra u til v. Dette er det som kalles *cancellation*.

12.5 Forstå hva en forøkende sti (augmenting path) er

En sti fra kilden til sluket i et restnett hvor flyten kan økes langs fremoverkanter, og flyten kan omdirigeres langs bakoverkanten. Altså en sti der den totale flyten kan økes.

12.6 Forstå hva snitt, snitt-kapasitet og minimalt snitt er

Et **snitt** (S, T) av en graf $G = (V, E])$ er en partisjon av V inn i S, og T = $V - S$. Gjøres gjerne for å se hvor flaskehalsen ligger.

Snittkapasitet av et snitt (S, T) er gitt som;

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v) \tag{8}$$

Et **Minimalt snitt** er et snitt hvor kapasiteten er minimum over alle snittene i restnettet.

12.7 Forstå maks-flyt/min-snitt-teoremet

Å finne det minimale snittet er det samme som å finne den maksimale flyten. Når vi har maksimal flyt vil vi få en flaskehals hvor kapasiteten er minst, og ingenting mer kan slippe gjennom. For å finne de minimale snittet når vi har funnet maksimal flyt gjøres ved å gjøre et bredde først-søk gjennom restnettet, fra (super)kilden. De nodene man når frem til er på kildens side.

12.8 Forstå Ford-Fulkerson-Method og Ford-Fulkerson

Ford-Fulkerson-Method går ut på å finne en flytforsøkende vei p og øke flyten f på hver kant til p med residualkapasiteten. Vi finner altså en flyforsøkende vei og setter på den flyten som veien tåler. Deretter finner man en ny flytforsøkende vei og gjør det samme. Når det ikke er flere flytforsøkende veier, har vi oppnådd maksimal flyt.

```
1 Ford-Fulkerson(G,s,t)
2     for each edge (u,v) in G.E
3         (u,v).f = 0
4     while there exists a path p from s to t
5     in the residual network G_f
6         c_f(p) = min{c_f(u,v):(u,v) is in p}
7         for each edge (u,v) in p
8             if (u,v) in p
```

```
9                   (u,v).f = (u,v).f + c_f(p=
10              else (v.u).f = (v,u).f - c_f(p)
```

Kjøretiden for kapasiteter av heltall vil være $O(E|f*|)$, hvor $f*$ er den maksimale flyten.

12.9 Vite at Ford-Fulkerson med BFS kalles Edmonds-Karp-algoritmen

12.10 Forstå hvordan maks-flyt kan finne en maksimum bipartitt matching

Matching går ut på at vi vil ha ut en delmengde $M \subseteq E$ for en urettet fraf $G = (V, E)$. Bipartitt matching vil si at M matcher partisjonene. Vi gir altså inn en bipartitt urettet graf, $G = (v, E)$. Og vi vil få ut en matching med flest mulig kanter, altså der $|M|$ er maksimal. Hver kant og hver node inngår maks i ett par. Og kanter med flyt inngår i matchingen.

12.11 Forstå heltallsteoremet (integrality theorem)

For heltallskapasiteter gir Ford-Fulkerson heltallsflyt.

13 NP-kompletthet

Kap. 34. NP-completeness

Oppgave 34.1-4 med løsning (0-1 knapsack)

Appendix D i pensumheftet Polynomisk kjøretid vil si at for et problem med input av størrelse n, er verste kjøretid gitt som $O(n^k)$

13.1 Forstå sammenhengen mellom optimerings- og beslutnings-problemer

Optimeringsproblem er et problem hvor en lovlig løsning gjerne er knyttet til en verdi, og vi er ute etter den beste verdien.

Beslutningsproblem er problem hvor svaret ganske enkelt er ja eller nei.

Vi bruker beslutningsproblemer for å prøve å vise om et optimeringsproblem er vanskelig eller ikke, fordi at beslutningsproblemet er som regel letter å svare på. I NP-komplett sammenheng kan vi si at om vi kan vise at et beslutningsproblem er vanskelig, kan vi også bevise at optimeringsproblemet er vanskelig.

13.2 Forstå koding (encoding) av en instans

For at et dataprogram skal kunne løse et abstrakt problem må vi representere instansen på en måte som kan tolkes av programmet. Vi bruker altså en binærstreng for å representere instansen (polygon, graf, funksjon, par,..)

13.3 Forstå hvorfor løsningen vår på 0-1-ryggsekkproblemet ikke er polynomisk

Dette er enkelt å se hvis vi lar m være antall bits i W, så vi kan skrive kjøretiden som $T(n, m) = \Theta(n2m)$. Da er det forhåpentligvis tydelig at dette ikke er en polynomisk kjøretid. Kjøretider som er polynomiske hvis vi lar et tall fra input være med som parameter til kjøretiden (slik som $\Theta(nW)$, der W er et tall fra input, og ikke direkte en del av problemstørrelsen) kaller vi pseudopolynomiske.

13.4 Forstå forskjellen på konkrete og abstrakte problemer

Et**konkret problem** er et problem som tar inn et sett av bitstrenger som input.

Et **abstrakt problem** er en binær sammenheng mellom sett I av probleminstanser og et sett av løsninger på problemet S.

13.5 Forstå representasjonen av beslutningsproblemer som formelle språk

Vi har et alfabet, Σ som består av endelig mengde symboler, og språk, L som er alle sett av strenger som er laget av symbolene i Σ. Vi kan da bruke det formelle språket vi nettopp har definert til å representere beslutningsproblem. Det blir en sammenheng mellom problemene og algoritmene som blir brukt for å løse dem. F.eks kan vi si at en algoritme A aksepterer en streng $x \in 0, 1$, hvis gitt input x, produserer algoritmen output $A(x) = 1$.

Komplementet til L, $\overline{L}$ defineres som $\overline{L} = \Sigma * -L\ \Sigma*$ er det språket med alle strengene av Σ. Vi definerer også en tom streng som ϵ og et tomt språk som

13.6 Forstå definisjonen av klassene P, NP og co-NP

P-klassen er språkene som kan avgjøres i polynomisk tid. Og det er disse problemene vi kan løse i praksis. (Cobham's tese)

NP-klassen er språkene som kan verifiseres i polynomisk tid. Eksempel på dette er språket for *Hamilton-sykel-problemet*. Det er altså lett å verifisere i polynomisk tid, men ikke alltid like lett å falsifisere.

co-NP-klassen er språkene som kan falsifiseres i polynomisk tid. $L \in co - NP \Leftrightarrow \overline{L} \in NP$. Eks; Tautologi

13.7 Forstå redusibilitets-relasjonen $\leq_P$

Hvis vi kan redusere A til B, skriver vi det som $A \leq_P B$. Hardhetsbeiset går da ut på:

Vise at B er vanskelig $\rightarrow$ redusere fra et vanskelig problem $A \rightarrow$ etabler $A \leq_P B$.

13.8 Forstå definisjonen av NP-hardhet og NP-kompletthet

NP-hardhet er mengden av de problemene som alle problemer i NP kan reduseres til i polnomisk tid. Disse problemene trenger ikke nødvendigvis å ligge i NP.

NPC eller NP-kompletthet er kort forklart en samling av de vanskeligste problemene i **NP**. Hvis vi har er NPC problem, B som kan reduseres til et annet problem A på mindre tid enn det tar å løse B, vil B være minst like vanskelig som A. En algoritme for B kan brukes for A, men ikke omvendt.

13.9 Forstå den konvensjonelle hypotesen om forholdet mellom P, NP og NPC

For å bevise at et problem er med i **P**, må vi finne en algoritme som løser problemet i polynomisk tid.

For å bevise at et problem er med i **NP**, må vi finne en algoritme som tester om en gjettet løsning er korrekt på polynomisk tid.

For å bevise at et problem er med i **NPC** , må vi ise at det er med i **NP**, og at det er minst like vanskelig som et annet problem som er med i **NPC**.

Hvis det finnes en polynomisk løsning på et problem i **NPC**, finnes det også polynomiske løsninger på alle problemer i **NP**. Alle NP problemer kan omformes til hverandre, og hvis vi beviser ett av dem, har vi klart å bevise $P = NP$.

13.10 Forstå hvordan NP-kompletthet kan bevises ved én reduksjon

Hvis vi har er NPC problem, B som kan reduseres til et annet problem A på mindre tid enn det tar å løse B, vil B være minst like vanskelig som A. En algoritme for B kan brukes for A, men ikke omvendt.

13.11 Kjenne til NP-komplette problemer

CIRCUIT-SAT: Instansen er en krets med logiske porter og én utverdi. Spørsmålet er om utverdien kan bli 1.

SAT: Instansen er en logisk formel, og spørsmålet er om formelen kan være sann.

3-CNF-SAT: Instansen er en logisk formel på **3-CNF-form**, og spørsmålet er om formelen kan være sann.

CLIQUE: Instansen er en urettet graf G og et heltall k, og spørsmålet er om G har en komplett delgraf med k noder.

VERTEX-COVER: Instansen er en urettet graf og et heltall k, og spørsmålet er om G har et nodedekke med k noder. Dvs k noder som tilsammen ligger inntill alle kantene.

HAM-CYCLE: Instansen er en urettet graf G, og spørsmålet er om det finnes en sykel som inneholder alle nodene.

TSP: Instansen er en komplett graf med heltallsvekter og et heltall k, og spørsmålet er om det finnes en rundtur med kostnad $\leq k$.

SUBSET-SUM: Instansen er en mengde S av positive heltall t, og spørsmålet er om det finnes en delmengde $S' \subseteq S$ slik $\sum_{s \in S'} s = t$

13.12 Forstå at 0-1-ryggsekkproblemet er NP-hardt

Det binære ryggsekkproblemet (0-1-knapsack) er et såkalt **NP-hardt** problem, og det er ingen som har funnet noen polynomisk løsning på det; trolig vil ingen noen gang finne det heller.

13.13 Forstå at lengste enkle-vei-problemet er NP-hardt

Går basically ut på å finne en minimum **Hamiltonsykel**, og det er da **NP-hardt**.

13.14 Være i stand til å konstruere enkle NP-kompletthetsbevis

For å vise at et språk L er **NP-komplett** kan vi følge følgende oppskrift;

1. Vis at $L \in NP$
2. Velg et kjent NP-komplett språk L'
3. Beskriv en algoritme som betegner en funksjon

   $$f : \{0,1\}* \rightarrow \{0,1\}* \tag{9}$$

   som mapper instanser av L' til instanser av L
4. Vis at

   $$x \in L' \Leftrightarrow f(x) \in L, \tag{10}$$

   for alle $x \in \{0,1\}*$
5. Vis at algoritmen som betegner f har polynomisk kjøretid.

A Algoritmer og Kjøretid

Tabell 1: Algoritmer og Kjøretid

Algoritme	**Best-Case**	**Average-Case**	**Worst-Case**	**Referanse**
Insertion sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	sec 1.7
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	sec **??**
Merge sort	$\Theta(nlg(n))$	$\Theta(nlg(n))$	$\Theta(nlg(n))$	sec 3.4
Heapsort	$O(nlg(n))$	$O(nlg(n))$	$O(nlg(n))$	sec 5.2
Quicksort	$\Theta(nlg(n))$	$\Theta(nlg(n))$	$\Theta(n^2)$	sec 3.5
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	sec **??**
Bucket sort	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n^2)$	sec 4.5
Counting sort	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$	sec 4.3
Radix sort	$\Theta(d+(n+k))$	$\Theta(d+(n+k))$	$\Theta(d+(n+k))$	sec 4.4
Select	$O(n)$	$O(n)$	$O(n)$	sec 4.7
Randomized select	$O(n)$	$O(n)$	$O(n^2)$	sec 4.6

Sammenligningsbasert: Sammenligner to elementer for å se hvem som skal stå først i sekvensen. Her er algoritmene begrenset av (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

Tabell 2: Nyttig Informasjon

Algoritme	**Sammenligningsbasert**	**Split og Hersk**	**In-place**	**Stabil**
Insertion sort	X		X[1]	X[2]
Merge sort	X	X	[3]	X[4]
Heapsort		X	X[5]	[6]
Quicksort	X	X	X[7]	[8]
Bubble sort	X		X	X
Bucket sort		[9]	X[10]	X[11]
Counting sort			[12]	X[13]
Radix sort			[14]	X[15]

[1] Bytter på to og to elementer
[2] Vil aldri flytte to like elementer forbi hverandre, uansett om man starter foran eller bak
[3] Pensum dikuterer ikke hvordan det kan gjøres
[4] Kun hvis den velger elementer fra venstre halvdel om elementene er like
[5] Bruker eksisterende tre til å swappe elementer
[6] Tar ikke hensyn til rekkefølge ettersom den baserer seg på en heap
[7] Den er rekursiv og "møblerer"om på elementene i returneringsfasen av algoritmen
[8] Kan gjøres stabil, men mer effektiv uten
[9] Den er ikke rekursiv og splittes kun opp til to nivåer.
[10] Må lage nye 'bøtter' som blir en datastruktur i minnet
[11] Stabil siden den bruker Insertion-sort
[12] Lager ny tabell med lenker til de nye elementene som injeseres på rett plass.
[13] Må være stabil for å brukes i radix-sort.
[14] Bruker counting sort.
[15] Fordi counting-sort og merge-sort er stabil.