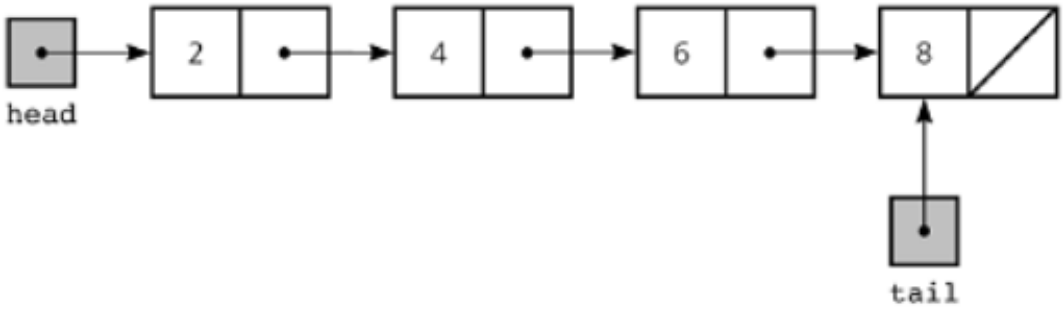
O Store AlgDat Ultimate Cheat Sheet 3000

Datastrukturer:

Lenket liste:

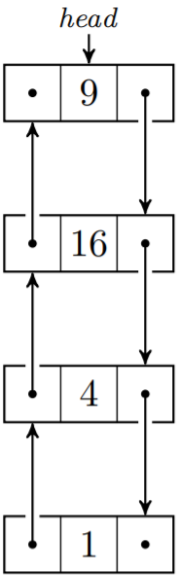
Lenket liste er en rekke node-objekter med to felter: node-verdien og pekeren til neste objekt. Som en liste som bruker pekere til neste objekt istedenfor å aksessere ved indeks. Lista kan være både sortert og ikke sortert, sirkulær og ikke sirkulær. Er den sirkulær peker siste node(tail) på første node(head).



Kjøretider:

- Konstruere listen: O(n-1) //Koble på pekere på alle noder utenom siste.
- Sette inn noe foran(push): O(1) //Nye node peker til første i resten av lista.
- Sette inn bakerst: O(n) //Itererer til bakerste node og setter peker til nytt element.
- Slå opp i liste(List-Search): O(n) //Worst case siden den muligens må iterere/traversere gjennom hele lista for å finne ønsket node.
- Slette element(List-Delete): Θ(n) = O(n) + O(1) //Tiden det tar å søke + tiden det tar å slette. «Få nodene før og etter til å referere til hverandre, slik at man hopper over x. X peker fremdeles på nodene, men har ingen funksjon siden ingen peker på den.

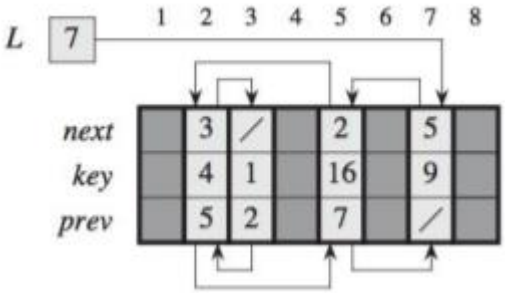
Dobbelt-lenket liste:



Nesten identisk som enkelt-lenket liste, bare at denne også har en *previous*-peker i tillegg.

Man implementerer ofte en sentinel-node som binder *head* og *tail* sammen. Dette forbedrer kjøretiden for å slette det bakerste elementet fra O(n) til O(1).

Man kan implementere lenkede lister selv om det ikke er støttet i alle språk. Man kan ha forskjellige lister som tar hvert sitt attributt. Én liste som tar alle *next*-pekere, én som inneholder alle *keys*(verdier) og én som inneholder alle *previous*-pekere, samt en variabel L som peker på *head*. Det er også mulig å implementere en liste som holder rede på alle ledige indekser i de tre listene.

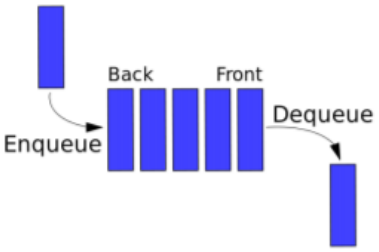


NB! Må ses mer på!

Kø (Queue):

En abstrakt datastruktur der rekkefølgen elementene blir lagt inn i er bevart. Det er to operasjoner i en kø: *enqueue* og *dequeue.* Enqueue legger til elementer bakerst i køen(tail) og dequeue tar ut det fremste elementet(head) som har ligget der lengst. En såkalt FIFO-datastruktur (First In-First Out). Dermed fungerer den likt som en kø i IRL. Køen kan gjerne også inneholde en peek-operasjon som titter på *head* uten å ta det ut av køen.

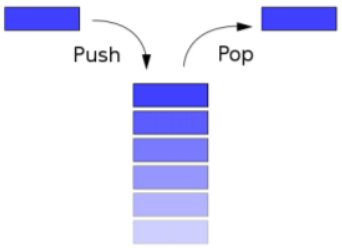
Når Q.head = Q.tail + 1: er køen full.



Implementeres gjerne som dobbelt-lenket liste eller en enkelt-lenket liste med en ekstra peker på bakerste element for å gi kjøretiden til operasjonene *enqueue* og *dequeue* konstant O(1).

Stakk (Stack):

I en stakk har man kun adgang øverst. Dette gir en LIFO-datastruktur(Last In – First Out). En analogi er hvis du har flere mynter oppå hverandre kan du ikke ta ut den nederste før du har fjernet alle som ligger oppå. Operasjonene er *push* og *pop* som henholdsvis legger til og fjerner øverst i bunken. Vi har også en simpel operasjon kalt *stack-empty(S)* som sjekker om lista er tom. Kalles push når lista er full får vi ***stack overflow***. Kalles *pop* når lista er tom får vi ***underflow***.



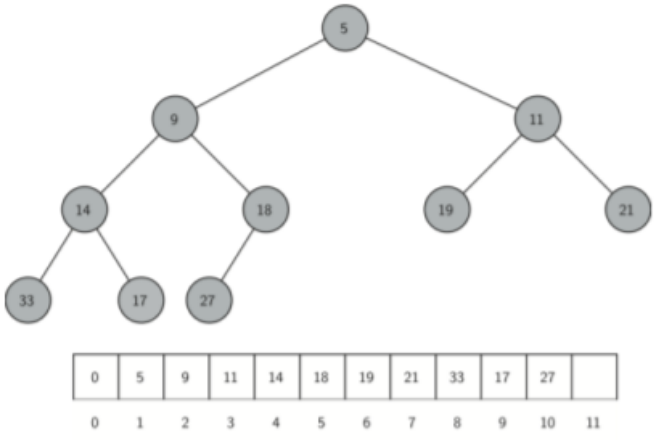
MERK: At når pop kalles fjernes bare indeksen fra lista. Selve satelittdataen/verdien ligger fremdeles i minnet.

Haug (heap):

En haug er ofte implementert som en liste men kan vises som et binærtre. To viktige versjoner: *min-heap* og *max-heap*. For at en datastruktur kan kalles en haug må den oppfylle egenskapene. For *max-heap* kan ingen barn ha større verdi en sin forelder. Motsatt for *min-heap* der ingen barn kan ha lavere verdi enn sin forelder.

For å aksessere barna:

- 2n og 2n+1 hvor n er posisjonen til en gitt node. Eks: finne barna til node 9. (indeks=2) 2*2 g 2*2 + 1 gir 4 og 5 som er node 14 og 18. Som vi ser i binær-treet er dette korrekt. (Heapen er 1-indeksert for å unngå regnefeil).



Figur 1: Min-heap

- For å finne forelderen brukes *floor(n/2)*.
- Hauger brukes som regel til *heapsort* men er også en vanlig måte å implementere en prioritetskø hvor det «viktigste» elementet ligger på toppen. Heap.size = antall elementer i haugen, ikke høyde.

Ulike operasjoner på hauger:

- *Insert:* Setter inn element bakerst i haugen, og kjører *heapify(enten max eller min)* for å plassere elementet korrekt. Kjøretid: O(log n)
- *Heapify (max eller min):* Kjøres når haugen er endret, om noe enten er fjernet eller lagt til. Sammenligner verdien av forelderen til noden som ble satt inn med verdien til denne noden og sørger for at *heap-egenskapen* alltid er oppfylt. I sammenligningen sjekker man om et av barna er større. Er begge det byttes foreldren med det største barnet. Løses etter hvert rekursivt for å ikke lage et ødelagt deltre. Kjøretid: O(log n).
- *Build-heap:* Tar inn en haugtabell, lager et tre og løser rekursivt. Grunntilfelle: Trær av størrelse 1 er alt korrekte. Induktivt premiss: Deltrær er korrekte ---> Induktivt trinn: fiks rota. Kjøretid: O(n) om man setter inn alle tallene med en gang.
- *Delete-max (eller min):* Sletter rotnoden og bruker deretter heapify for å rydde opp.
- *Extract-max (eller min):* Tar ut rotnoden som i delete, men lagrer den. Også kalt pop. Brukes i utførelsen av heapsort.
- Prioritetskø (priority queue):

Implementeres ofte som en haug med pekere til max- eller min-elementet. Avhengig av om det er en min- eller maks-kø. Algoritmer som tar i bruk dette får økt effektivitet da alle nødvendige operasjoner på prioritetskøen kun tar O(log n) tid.

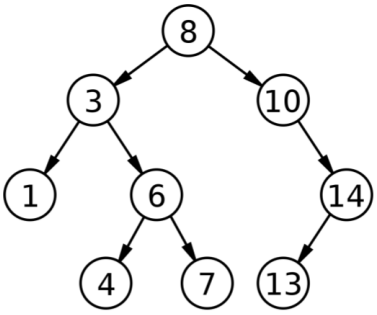
Kan også implementeres på andre datastrukturer som f. eks. array, men haug er mest brukt.

Trær:

I AlgDat er det i hovedsak snakk om binære trær, der hver node kun kan ha to barn. I tillegg til hauger har vi binære søketrær. Er også et binært tre men har ikke haug-egenskapen. *Binær-søketre-egenskapen* må her være oppfylt: La x være en node i treet. Hvis y er en node i det venstre deltreet til x, så er x.key >= y.key. Hvis y er en node i det høyre deltreet til x så er x.key <= y.key.

Kjøretid for søk i binærtre er O(log n).

Å bygge et binært søketre gjør man ved å starte ved rotnoden, og så sammenligne og sortere etter om den er større eller mindre. Er den større plasseres den ned til høyre, er den mindre plasseres den ned til venstre. Dette gjør vi helt til vi kommer til en tom plass. Dette tar O(h) tid der h = høyden til treet.



Representrere trær med lenkede datastrukturer:

Binære trær kan implementeres ved å bruke en dobbelt lenket liste med pekere til forelder og høyre og venstre barn. Listen har en egen peker til roten; T.root.

Samme prinsipp fungerer dersom det ikke er et binærtre også, men at hver node har et endelig antall *k* barn. I stedet for å ha pekere for alle barn vil den ha to pekere i tillegg til pekeren til forelder:

Left-child som peker til det barnet som ligger helt til venstre.

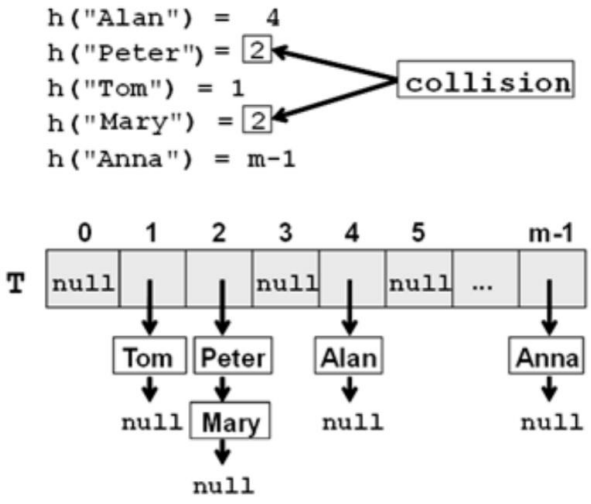
Right-sibling som peker på søskenet som ligger til høyre for noden.

Hauger, som er basert på binærtrær, kan implementeres som en enkel (ikke-lenket) liste. Se under hauger for hvordan de fungerer.

Hashede tabeller:

Hashing er mye brukt til kryptering, fordeling av verdier utover i minnet til pc basert på input etc. Slik det fungerer er at du tar inn et input, for eksempel et passord, så utfører du en funksjon på det og får et output på en gitt (og gjerne ønsket) form. En tabell uten hashfunksjon kalles en *direct-address table* og fungerer bra (O(1)) når universet med elementer du ønsker å lagre ikke er spesielt stort. Navnet forklarer seg egentlig selv, du har en nøkkel som du sender til tabellen og den vil da finne dataene lagret i den cellen i tabllen. Gitt at ingen nøkler er like, og at unvierset som sagt ikke er spesielt stort, så vil dette gi konstant kjøretid, O(1).

Hash tables derimot er det vi bruker når universet blir gradvis større og når det å gi en unik nøkkel til hver rad blir vanskelig. Vi har fortsatt et sett med unike nøkler, men denne gangen utføres en funksjon på et input som gir en spesiell nøkkel. F.eks. hvis en verdi/input er et heltall, så utføres gjerne en modulo (%) på tallet for å få nøkkelen til ønsket celle i tabellen. Her ser vi at modulo operatoren ofte vil gi det samme svaret for forskjellige input (3%2=1 og 5%2=1). Hvis to nøkler peker til samme celle, betyr det at det ligger mer enn et objekt der. Cellen er da gjerne implementert som en lenket liste. Dette kalles *chaining* eller lenking på norsk.



For å unngå *chaining* til en viss grad er det lurt å ta modulo av et primtall.

Disse nøklene lagres i en tabell som inneholder enten:

- Verdien *null* som tyder på at det ikke ligger noe her.
- En peker til en enkelt lenket liste som er n elementer lang.

Figuren illustrerer dette godt.

For at hashing skal være gunsting ønsker vi en best mulig hash-funksjon som gir en mest mulig jevn fordelt liste, altså at ikke alle input blir hashet til samme celle i tabellen og den lenkede listen blir veldig lang. Hvis vi oppnår denne jevne fordelingen vil hashing kunne i O(1) i søkingen etter elementer. Å lagre pekere i tabellen på denne måten er en av flere måter å lagre data etter hashing på. Vi har også metoden som kalles *open addressing*. Det går ut på at det ikke ligger pekere i tabellen, men at faktiske data ligger her. Hvis det da foregår kollisjoner så lager man ikke lenger en lenket liste, men man flytter dataene med en prosess kalt *probing.* Man har flere typer probing; linær probing, quadratic probing og double extendable hashing. Med lineær probing flytter man krasj-dataen man ønsker å sette inn i første ledige celle under nøkkelcellen. Med quadratic probing flytter man det med et eller annet polynomisk mellomrom. Med dobbel/extendable hashing utfører man en hashfunksjon til. Man kan også ha en *dynamisk hashtabell* som øker adresserommet i tabellen ettersom det er behov.

Algoritmer

Sortering ved sammenligning

Sorteringen i disse algoritmene skjer ved sammenligning av to og to elementer, og mange av dem er *inplace*. Enhver sortering ved sammenligning krever worst case kjøretid Ω(n lg n). Det er altså umulig å ha sammenligningsbaserte algoritmer som *alltid* vil kjøre raskere *n lg n*. Dette gjør heapsort og mergesort de mest optimale sammenligninsalgoritmene (WC: O(n lg n)).

TIPS: https://visualgo.net for å få bedre forståelse, merk at IKKE ALLE algoritmene er implementert på samme måte som i boka.

Bubble sort:

- Dårlig, lite effektiv og lite brukt.

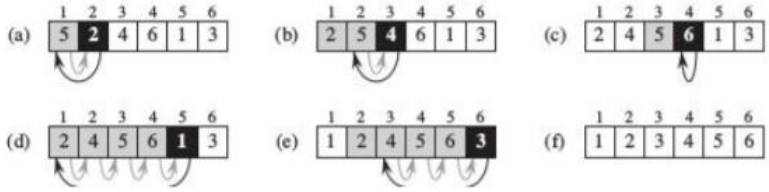
Tester to og to naboelementer og bytter plass dersom det første er større enn det andre elementet. Dette kjører *n* ganger. Den blir veldig treig på store tall med $\Theta(n^2)$, og brukes lite.

Insertion sort:

- Stabil
- Input: Lista A
- In place, deles ikke til subproblemer
- Gjennomsnittslig og verste (omvendt sortert liste) kjøretid $\Theta(n^2)$
- Bestcase Θ(n) (nesten sortert liste)

Algoritmen velger det andre elementet i lista først og sammenligner det med det første. Den itererer seg gjennom lista og flytter elementet nedover eller plasserer det på riktig plass etter sammenligning med elementet foran. Den er effektiv på små datamengder og dersom man har en nesten sortert liste. Si for eksempel at man har en hånd med sorterte kort og får

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



ett kort til, da er det lurt å bruke insertion sort!

Mergesort:

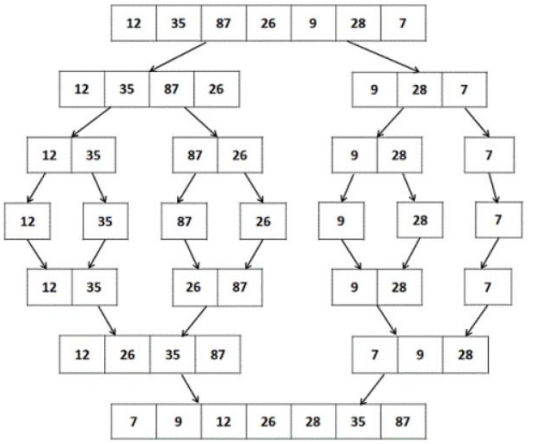
- Stabil
- Input:
- Merge: Liste A, p & q & r er indekser slik at p ≤ q < r
- Mergesort: Liste A, p & r er indekser slikt at p ≥ r → brukes til å skille høyre og venstre delsekvens.
- Ikke in place
- Divide and conquer → deles opp til subproblemer.
- Kjøretid vil ALLTID være Θ(n*lg n)
- Hjelpes av MERGE
- Bruker 'sentinels' = ∞ for at delsekvensene ikke går tom før den andre lista er tom også.

Algoritmen løser sorteringsproblemet ved å dele opp lista A i to nye lister og sorterer sidene rekursivt (kjører mergesort på de to delsekvensene). To og to lister flettes sammen og legges tilbake i A ved at de to første elementene i hver liste sammenlignes og det minste lagres i A. Ved å legge til *sentinel*-elementet i slutten av hver liste, kan vi vite at når dette elementet er øverst i den ene lista legges resten av den andre lista til (unntatt siste element som er ∞), for

verdiene i den gjenværende lista må være mindre enn ∞. Merk at dette er en rekursiv algoritme som kaller seg selv først på venstre side, deretter på høyre side. Kjøretid er alltid Θ(n*lg n), fordi merge har kjøretid Θ(n) (maks antall sammenligninger per nivå) og kjøres lg n ganger (lg n nivåer).

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

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



Heapsort:

- Ikke stabil
- Input: Liste A
- In place
- Kjøretid O(n lg n)
- Deler opp problemet, men ikke på divide-and-conquer måte.

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

Først bruker algoritmen *BUILD-MAX-HEAP* for å gjøre arrayen A om til en max-heap (ligger under datastrukturer/heap). Deretter byttes det største elementet, A[1] som ligger først i heapen, om med det siste elementet A[n] (OBSOBS! Heaps er 1-indeksert). Vi minker størrelsen på heapen med 1, og kalles *MAX-HEAPIFY.* Da får man en ny max-heap basert på

de n-1 første elementene i arrayen og det siste elementet står uberørt og ferdig sortert. For hver gang blir heapen 1 mindre, og den sorterte lista 1 større del av arrayen.

Kjøretiden er O(n lg n) fordi max-heapify har kjøretid O(lg n) og blir kjørt n ganger, og *BUILD-MAX-HEAP* har kjøretid O(n). → O(n + n lg n) = O(n lg n).

OBS! Merk at *heapsort* visualiseres som et tre men implementeres som en array, derfor er den inplace.

Quicksort:

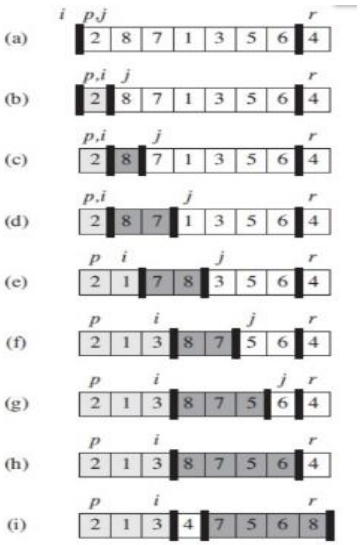
- IKKE stabil
- Input: Liste A, indeks fra p til r
- Kan være in place (men ofte ikke)
- Divide and conquer
- Kjøretiden er avg. og best case $\Theta(n \lg n)$
- Worst case $\Theta(n^2)$
- Hjelpes av *PARTITION*

Algoritmen løser sorteringsproblemet rekursivt ved å kjøre *PARTITION* på arrayen A med start- og sluttindeks henholdsvis p og r og deretter kalle *QUICKSORT* på de to delene. *PARTITION* deler opp lista i to lister hvor den ene lista inneholder elementer som er mindre enn pivot-elementet (bakerste element i opprinnelige lista i vanlig quicksort) og den andre de som er større.

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

PARTITION fungerer slik at den velger siste element A[r] som pivot og går så igjennom lista fra p til r-1. Indeksen *i* betegner hvor skillet mellom den minste og den største lista går og indeksen *j* sier hvor langt i lista vi er kommet. Hvis et element er mindre enn pivot, byttes plassen på dette elementet med det første elementet i delen av elementer som er større enn pivot og *i* og *j* økes med 1. Hvis elementet er større enn pivot får den stå på plassen sin og *j* økes med 1. Til slutt legges pivot elementet mellom de to listene og *PARTITION* returnerer pivots indeks.

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



Kjøretiden til *quicksort* er avg. og best case Θ(n lg n), men som regel bedre enn mergesort fordi den har bedre/mindre konstanter. Partitioning har kjøretid Θ(n) og kjøres lg n ganger. Worst case er $\Theta(n^2)$ hvis lista er sortert og det siste elementet velges som pivot hver gang, da kjøre partition *n* ganger. Men hvis man velger tilfeldig pivot (*randomized quicksort*) kan man unngå dette.

Selection sort:

- Ikke stabil
- Input: Lista A
- Kjøretid $\Theta(n^2)$
- En dårlig algoritme

Løser sorteringsproblemet ved å opprette en tom liste før den velger det minste elementet i lista som skal sorteres og appender den i den nye lista før den fjerne den i den opprinnelige lista. Kjøretiden er $\Theta(n^2)$.

Sortering i lineær tid:

Sortering i lineær tid forutsetter at vi har kunnskap om inputen på forhånd, slik at vi kan senke kjøretidene og unngå sammenligning. Mergesort og quicksort er de beste sammenlignings-algoritmene, fordi de når den optimale kjøretiden for denne sorten. Det har en laveste mulig kjøretid på $\Omega(n \lg n)$.

Counting sort:

- Stabil → viktig da den ofte er brukt i radix sort
- Ikke inplace
- Input: Lista A som har usortert input, Lista B som har sortert output og k er det største elementet i lista.
- Ikke rekursiv
- Kjøretid er $\Theta(k + n)$ men $\Theta(n)$ dersom $k = O(n)$
- Lagringsplassen er også O(k+n)

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

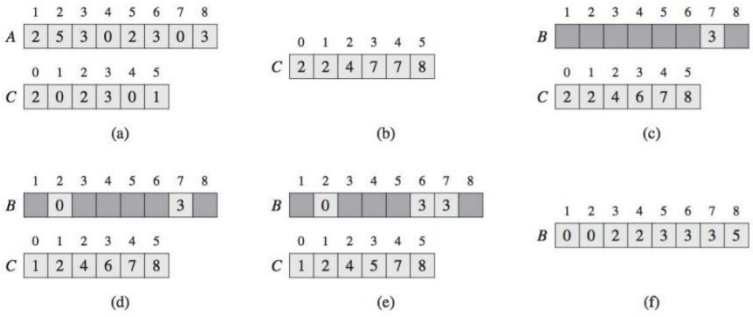


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after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

Lineær sorteringsalgoritme som antar at input A har n elementer med verdi mellom 0 og k. Det lønner seg kun å bruke counting sort dersom k ikke er mye større enn n, slik at k = O(n). Det opprettes en ny liste C med indekser fra 0 til k og det telles opp antallet av hver

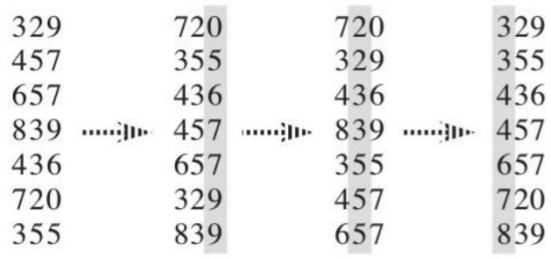
forekomst. Deretter gjøres listen om til å inneholde antallet av tall som er mindre enn seg selv pluss antallet av seg selv, altså er ny verdi på indeks i = C[i+1] + C[i] (C kalles auxiliary array, se figur). En liste B skal fylles opp med den endelige outputen. Da begynner man med bakerste element i A, A[j], og ser etter denne indeksen i C. Der finnes antallet *m* av tall som er mindre eller lik A[j] og elementet A[j] puttes på indeks *m* i listen B. Tallet *m* i C[A[j]] minker med 1 og j minker med 1. Dette fører til at dersom elementene ikke er distinkte er elementet som er sist i A også er sist i output B, som gjør algoritmen stabil. Algoritmen var vanskelig å forklare med ord, se figur under. Kjøretiden for counting sort er Θ(n + k), men i praksis når k = O(n) blir kjøretiden bare Θ(n).

Radix sort:

- Stabil
- Input: Lista A, konstant *d* som er lengden på det største elementet i lista A.
- Ikke in place, fordi den bruker counting sort (eller bucket sort)
- Kjøretid Θ(d(n + k))

Lineær sorteringsalgoritme som antar at input er n elementer med d antall siffer, der hvert siffer kan ha opp til k forskjellige verdier. Algoritmen benytter en annen sorteringsalgoritme (som MÅ være stabil) til å sortere alle elementene på minst signifikante siffer, før den sorterer på nest mest signifikante siffer osv, før c binarden tilslutt sorteres etter det mest signifikante sifferet. Det er også viktig at hjelpesorteringsalgoritmen kjører i lineær tid, for hvis ikke mister radix sort sin effektivitet. Vi kan f. eks. bruke counting sort (mest brukt) eller bucket sort. Kjøretiden er Θ(d(n + k) dersom hjelpesorteringen har kjøretid Θ(n + k).

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



Bucket sort:

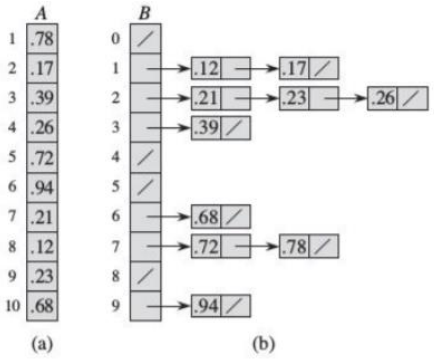
- Stabil
- Input: Lista A
- Kjøretiden er Θ(n), med worst case $\Theta(n^2)$

Lineær algoritme om antar at input A er fordelt unformt og uavhengig over et intervall [0,1). Fordelingen *må* ikke være uniform, men dersom den er uniform vil bucket sort kjøre i lineær tid. I motsetning til counting sort trenger vi ikke å ha heltall, men kan sortere rasjonale tall. Liste B opprettes med n = A.length like store bøtter som dekker et intervall på størrelse 1/n. Elementene fordeles i bøtta de hører hjemme før de der blir sortert innad med en annen sorteringsalgoritme, f. eks. insertion sort. Hvis det er intervaller som har høyere frekvens av verdier kan disse bli delt med flere bøtter. Til slutt settes bøttene sammen i rekkefølge fra B[0] til B[n-1].

Kjøretiden er Θ(n), men har worst case på $\Theta(n^2)$ når input er dårlig fordelt og alle havner i samme bøtte (når fordelingen ikke er uniform).

OBS! En bøtte implementeres ofte som en lenket liste.

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



(a) (b)

Selektering i lineær tid

Handler om å finne det n-te minste/største elementet i en liste. Dette kan vi også gjøre ved å sortere listen og returnere elementet på plass n, men det tar lengre tid.

Randomized select

- In place
- Input: Lista A, p & r er A[p…r], i er rang
- Divide and conquer, løses rekursivt
- Kjøretid O(n) avg.
- Samme problem som quicksort, worst case O(n) (Om lista er sortert og det siste elementet blir valt som pivot hver gang)
- Forventer at elementene er distinkte

Randomized select brukes for å finne f.eks det i-te minste/største elementet i en liste ved å delvis sortere den ved hjelp av *RANDOMIZED-PARTITION* (Se under Quicksort). Den velger et tilfeldig pivot-element og slenger alle elementer mindre enn pivot-elementet på en side og alle de større på den andre siden. Så sjekker den om pivot-elementet er element nummer n (som vi leter etter) i den nye listen som er halvsortert. Hvis pivot-elementet ikke er det vi leter etter sjekker vi om elementnummeret vi leter etter er større eller mindre enn indeksen til pivot-elementet. Vi kjører så randomized-select igjen på den delen av listen som elementet ligger i (den siden med større eller den med mindre elementer). Gjentar prosessen til pivot-elementet er det i-te elementet i listen. Du har da funnet det i-te minste eller største.

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

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

Select:

- In place
- Input: Lista A, p & r er A[p…r], i er rang
- Kjøretid: O(n) på alle
- Hører til divide and conquer

Bruker median som pivot. I select er god partisjonering nøkkelen! Forskjellen på S og RS er måten å velge pivot på

Bisect (binærsøk):

- Divide and conquer.
- Input: Lista A, p → venstre, r → høyre, v → søkeverdi
- Kjøretid er Θ(lg n)

Søkealgoritme som rekursivt utføres på **sortert** liste ved at den sjekker om søke-elementet er større eller mindre enn midtelementet i lista. Så gjøres det samme søket på nederste eller øverste halvdel om søke-elementet er henholdsvis mindre eller større enn midtelementet. Søket avsluttes når søket blir gjennomført i liste av størrelse 2.

Kjøretiden for denne algoritmen er Θ(lg n)

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

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

Optimaliseringsalgoritmer

Disse algoritmene brukes til å løse forskjellige problemer hvor vi trenger å finne den *beste* av potensielt mange løsninger.

Rod cutting

Skal selge en stan på *n* meter og har forskjellige prise per delstykke.

Optimalisering

Bruker dynamisk programmering. Deler i forskjellige deler og sammenligner, kalles deretter rekursivt som gir optimal substruktur. I de rekursive kallene kan samme problem bli kalt flere ganger, altså overlappende delstruktur. Lager en slags lagring av svarene på en bottom-up måte.

Med memoisering og bottom-up har den kjøretid $\Theta(n^2)$

r_n: maks avkastning for stang av lengde n. $r_n = \max(p_n + r_{n-i})$ for i fra 1 til n

p_n: prisen for en stang av lengde n

Knapsack 0-1

Merk! 0-1 betyr enten/eller

Analogi: En tyv med sekk som tar et antall kg/liter og tyven vil stjele ting til høyest mulig verdi. Han kan bare ta med seg én av hvert objekt, og ikke dele objektene i mindre deler. Koker ned til: Skal vi ta med objektet, eller skal vi ikke ta med objektet? Bruker vi grådig løsning her får vi ikke til optimal løsning.

GRÅDIGHET VIRKER IKKE, BRUK DYNAMISK

Fractional knapsack

Har grådighetsegenskapen. Kan dele objektene i flere mindre deler. Her finner vi først W = verdi/vekt. Tyven tar så mye av den med mest W, før han går på den med nest mest mulig W osv. helt til han ikke kan bære mer.
GRÅDIGHET VIRKER

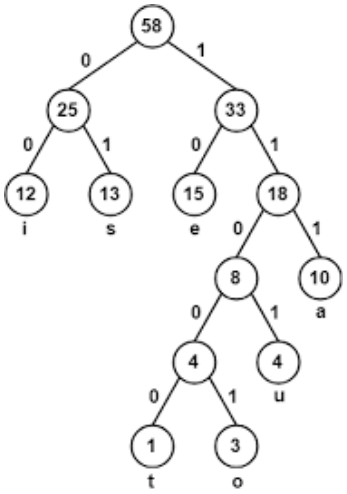
Denne grådige algoritmen kjører på O(n lg n).

Aktivitetsplanleggeren

Grådige valg: Finner til enhver tid aktiviteten som slutter først og velger den.

Huffman

Hvis du skal kode en melding i bits, hvordan kan du gjøre slik at bitstrengen blir så kort som mulig? Dette er en grådig algoritme som konstruerer optimale prefix-koder (koder som **ikke** er prefixer av hverandre) slik at man alltid vet hvilken bokstav som er kodet. Vi kan lage et Huffman-tre! Det første du gjør er å telle opp antall ganger de forskjellige bokstavene/elementene forekommer i meldingen din. De som forekommer flest ganger har såkalt høyest frekvens (og motsatt). De to med lavest frekvens blir løvnoder og foreldrenoden får summen av løvenes frekvenser. Det er viktig at man er konsekvent med å plassere den minste på venstre side i hele treet. Du sjekker så de to nodene eller trærne med lavest frekvens og slår de sammen (da er foreldrenoden til de to minste i første steg inkludert som en egen node). Slik fortsetter du til alt er slått sammen. Til slutt må du gi nodene sine prefixer. Du jobber deg nedover i treet og gir kantene når du går mot venstre 0 og 1 når du går mot høyre. Veien til ønsket node i treet vil da gi riktig prefix-kode.



Topologisk sortering

En topologisk sortering er en spesiell rekkefølge noder må komme i fordi de på et vis er avhengige av hverandre. Tar utgangspunkt i DAG, en rettet og asyklisk graf. Et hverdagslig eksempel på dette er når vi kler på oss. Vi må ta på t-skjorte før vi tar på genser, underbukse før bukse osv. Alt må ikke være avhengig av alt. F. eks. du kan ta på klokke når du vil.
Topologisk sortering utfører DFS og nodene med lavest finish times legges til først i den lenkede lista (den blir helt til høyre i sorteringen).
Kjøretid: O(V+E) som DFS og det tar O(1) tid å legge til element foran i den lenkede listen.

BFS (bredde først)

- Kjøretid: O(V+E)
- Finner korteste vei i en urettet, uvektet(eller at alle har samme vekt!) graf

Gitt en startnode *s* så vil BFS først se på alle barna(noder) til *s* og lagre dem i den rekkefølgen de oppdages. Vanligvis har nodene fått en eller annen verdi som sier noe om hvilken rekkefølge du skal besøke dem dersom du ikke vet hvilket barn du vil se på først. La oss si at *s* har tre barn. Når du har oppdaget alle barna til *s*, så ser du på det første av barna til *s* og oppdager alle barna til den noden. Så vil du se på det andre barnet og oppdage den sine barn. Så vil den gjøre det for det tredje barnet og slik fortsetter den prosessen. Algoritmen benytter en *queue*, slik at den noden som er først i køen er den vi utforsker naboene til. De ubesøkte nabonodene legges til bakerst i køen. Når vi har besøkt alle naboene til en node, fjernes denne fra køen og markeres med svart, som ferdig oppdaget.

```
BFS(G, s)
1  for each vertex u ∈ G.V − {s}
2      u.color = WHITE
3      u.d = ∞
4      u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13         if v.color == WHITE
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)
18     u.color = BLACK
```

DFS (dybde først)

- Kjøretid: O(V+E)

Traverser grafen ved å gå så langt som mulig i en retning uten å lage sykler, går så et steg tilbake og prøve å gå videre så dypt som mulig der, før man går tilbake igjen. Repeteres til alle noder er besøkt.

Kan brukes til *topologisk sortering* hvor vi er avhengig av at noen steg utføres i en spesiell rekkefølge, mens andre steg ikke er så nøye med. Det må da bestå av en rettet, asyklisk graf (DAG). Kan implementeres med en stack.

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

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

Generic-MST

Generell forklaring på de grådige algoritmene Kruskal og Prim som finner et minimalt spenntre for en graf. Den har et sett med kanter A, og for hver iterasjon finner vi en kant som gjør at A fortsatt er et subset av MST.

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

Både Kruskal og Prim er spesialtilfeller av *GENERIC-MST*, og de bestemmer om en kant er trygg ved:

Kruskal: Den kanten som har lavest vekt og kobler sammen to komponenter uten å lage en sykel.

Prim: Den kanten som har lavest vekt og som setter sammen treet med en node som ikke er i treet.

MST-Kruskal

- Kjøretid: O(E lg V)
- Har kanter i *min-priority-queue*
- Grådig algoritme

Lager minimalt spenntre av en graf ved at den fjerner alle kantene og gjeninnfører de grådig etter stigende vekt (uten å lage sykler!) til alle nodene er nådd. For å implementere trenger man en avansert datastruktur som kalles *disjoint set* for å raskt finne ut om å legge til en kant vil danne sykler eller ikke. Derfor går vi ikke så inn i selve implementasjonen.

ENKLE poeng på eksamen. Bruk fargeblyant til å markere hvilke du har valgt 😊 Hold tunga rett i munn! Analogi: strør kanter utover som (Kruska)kli-mel.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

MST-Prim

- Kjøretid: O(E lg V)
- O(E+V lg V) om fibonacci-heap brukes istedenfor binary min-heap.
- Har ubrukte node i *min-priority queue*

Grådig algoritme. Lager minimalt spenntre av en graf ved å fjerne alle kantene, velge tilfeldig startnode og fra den velge den kanten med minst vekt som er tilknyttet startnoden. Deretter velges kanten som har minst vekt tilknyttet hvilken som helst av nodene som allerede ligger i spenntreet. Gjentar dette til alle nodene er nådd.

Analogi: Smøres utover fra første node som prim.

```
MST-PRIM(G, w, r)
1   for each u ∈ G.V
2       u.key = ∞
3       u.π = NIL
4   r.key = 0
5   Q = G.V
6   while Q ≠ ∅
7       u = EXTRACT-MIN(Q)
8       for each v ∈ G.Adj[u]
9           if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

Algoritmer for korteste vei – Én til alle

Algoritmer som løser problemet med å finne korteste vei fra én node til alle andre noder.

Bellman-Ford

Én til alle: Algoritme som løser korteste vei-problemet, ved at *relaxer* (står om relaxing lenger ned) nodene i en graf *E* ganger og det gjøres *V-1* ganger. Må være en rettet graf og

algoritmen har kanter i prioritetskø. Takler negative kanter, men **rapporterer** dersom det finnes negative sykler.

Kjøretid: O(VE)

```
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
```

DAG – Shortest path

Rettet, asyklisk graf som kan ha negative kanter (men ikke sykler, for den er jo asyklisk). Topologisk sorterer alle nodene, så initialiseres single-source. For hver node fra venstre blir hver kant til den noden relaxet. Kjøretid er Θ(V + E) pga topologisk sortering.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4      for each vertex v ∈ G.Adj[u]
5          RELAX(u, v, w)
```

Dijkstra

Kan ikke ha negative kanter og må ha rettede kanter. Grådig algoritme. Legg alle noder i en min-prioritetskø og pop én og én node fra køen. Relax alle kantene til den noden, pop neste node og fortsett slik. Stopper når alle noder er poppet. Kjøretid O(E lg V) med binary heap.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Algoritmer for korteste vei – alle til alle

Finne korteste vei fra mellom alle nodene i en graf. Er også mulig å bruke dijkstra på alle nodene, dersom det ikke er noen negative kanter. Dersom E lg V ≤ V^2 er Dijkstra bedre enn Floyd-Warshall. Da blir kjøretiden til Dijkstra O(VE lg V) som er raskere enn Floyd-Warshall.

Johnson

- Bra for en sparse graph (minimalt med kanter)

Floyd-Warshall

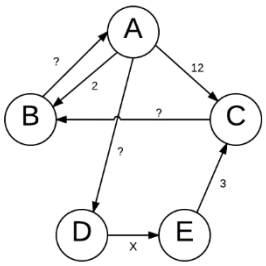
- Kjøretid $O(V^3)$
- Ingen negative sykler
- Bra for en dense graph(nesten alle noder har kant til alle)

Dynamisk bottom-up-algoritme som bruker memoisering i distansematrise og en sekvensmatrise. På den måten kan man slå opp på element $d_{ij}^{(k)}$ i sekvensmatrisa for å finne korteste vei mellom node i og j, via k osv. Dersom det er kortere, settes k som parten i sekvenstabellen.

$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}, d_{kj}^{(k-1)})$ if $k \geq 1$

Hvis grafen er uretta, husk å presisere hva som er fra og til av rad og kolonne i sekvenstabellen. Konvensjonen er rader fra, og kolonner til.

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

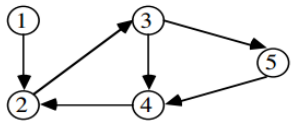


	A	B	C	D	E
A	0	2	10	5	7
B	3	0	13	8	10
C	7	4	0	12	14
D	12	9	5	0	X
E	10	7	3	15	0

Transitive Closure

Bruker samme prinsipp som Floyd-Warshall, men tar bare hensyn til om vekten finnes eller ikke. Dvs. 1 dersom det er en vekt mellom nodene (hvis det er en rettet graf må vekten også gå riktig vei) og 0 dersom det ikke er en vekt mellom dem. Etter iterasjonene vil man finne ut hvilke noder det er veier mellom ved ett 1-tall i matrisen, mens 0 vil si at det ikke er mulig å komme seg fra den aktuelle noden til den andre.

$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$ om $k \geq 1$, der $\vee$ (logisk ELLER) og $\wedge$ (logisk OG).



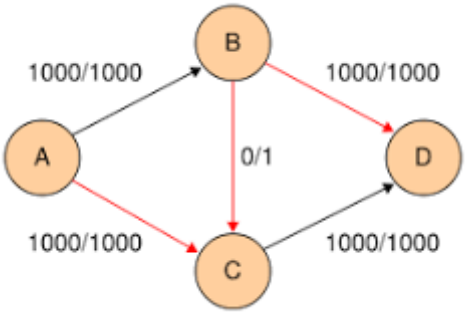
A	1	2	3	4	5
1		T			
2			T		
3				T	T
4	T				
5				T	

C	1	2	3	4	5
1	T	T	T	T	T
2		T	T	T	T
3		T	T	T	T
4		T	T	T	T
5		T	T	T	T

NB! Må ses mer på

Ford-Fulkerson

Brukes til å beregne maksflyt, men er ikke en algoritme i seg selv. Metoden går ut på å finne en flytforøkende vei og øke flyten langs den. Repeteres helt til det ikke finnes en flytforøkende vei, og da er maksflyt beregnet. En måte å finne flytforøkende vei er å bruke BFS, algoritmen kalles da Edmonds-Karp-algoritmen.




```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p in the residual network G_f
3      augment flow f along p
4  return f
```

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

Longest common subsequence

Dynamisk programmering for å finne lengste subsekvens mellom S_1 og S_2, hvor bokstavene ikke trenger å ligge rett etter hverandre. (Atomrakett & tomat)

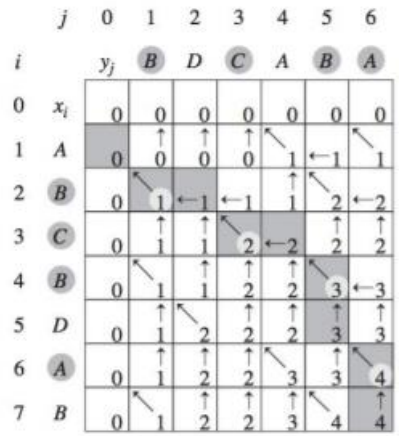


Figure 15.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the sequence is shaded. Each "↖" on the shaded sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

Algoritmeoversikt (Optimaliseringsalgoritmer)

	Kjøretid	Bruksområde	Kommentar
BFS	O(V + E)	Søk	Uvektet graf
DFS	O(V + E)	Søk	
MST-Kruskal	O(E lg V)	MST	
MST- Prim	O(E lg V) / O(E+V lg V)	MST	E lg V med binary-min-heap E+V lg V med fibonacci-heap
Bellman-Ford	O(VE)	SSSP	Kan ha negative kantvekter, og vil rapportere negative sykler.
DAG	Θ(V+E)	SSSP	
Dijkstra	O(E lg V)	SSSP	Kan bare ha positive kantvekter.
Floyd-Warshall	$O(V^3)$	ASPS	
Ford-Fulkerson	O(E|f*|)	Max-flow	f* = maksimal flyt. Implementert med BFS blir det $O(VE^2)$
Topologisk sort	Θ(V + E)		
Huffman	O(n lg n)	PBC	Finne prefiks-kode.
Transitive Closure	$Θ(n^3)$		

MST = Minimalt spenntre, SSSP = Korteste vei fra én til alle, APSP = Korteste vei alle til alle,
PBC = Finne den korteste binære prefikskoden som representerer en tekst.

Grunnlag for kjøretider

BFS: Legge til og ta ut av kø tar O(1) tid, så det blir O(V) tid i BFS. Skanner nabolisten for hver node når den er dequeuet, og lengden av alle nabolistene er Θ(E). Skanningen tar da O(E).
O(V + E)

DFS: Fargelegger hver hvit i starten tar O(V), søker så gjennom nabolistene(DFS-Visit) O(E).
O(V + E)

Kruskal: Sorterer med en av de bedre algoritmene til O(n lg n). Men må søke gjennom kantene → O(E lg E). Dette er unøyaktig, kan gjøre det raskere ved **O(E lg V)**

Prim: Bruker *build-min-heap* til O(V) så vi kan bruke *extract-min* til O(lg V), while loopen extract-min er inne i tar |V| tid, så samlet O(V lg V) for alle extract-min-operasjoner. I en for-loop senere sjekker den alle kanter O(E). Så alt blir O(V lg V + E lg V) = **O(E lg V).**

Bellman-Ford: Init tar Θ(V) tid og linje 2-4 sjekker kantene altså Θ(E), og neste for-loop tar O(E). Dermed **O(VE)**

DAG: Init med topologisk sortering tar Θ(V+E) tid. Init-S-S tar Θ(V) tid. Alt inne i for-loopen tar O(1) tid, så dermed **Θ(V + E)**

Dijkstra: *Extract-min* tar O(lg V) tid, og det er |V| slike operasjoner. *Build-min-heap* tar og O(V) tid. Hver *decrease-key-operasjon* tar O(lg V) tid og det er |E| antall slike operasjoner. Totalen blir da O((V + E) lg V) = **O(E lg V)**

Floyd-Warshall: Siden operasjonen i linje 7 inne i de nøstede løkkene tar O(1). Derfor tar algoritmen $\Theta(n^3)$ fordi det er 3 nøstede løkker.

Topologisk: Bruker DFS som tar Θ(V + E) og det tar O(1) å sette inn hver |V| i starten av den lenkede listen. Dermed **Θ(V + E)**

Ford-Fulkerson: While-loopen i linje 3-8 blir gjort |f*| antall ganger. Tiden det tar å finne en sti i ett residualnettverk er derfor O(V + E') som er O(E) om vi bruker DFS eller BFS. Så både init og selve loopen tar O(E) tid så derfor blir det en total på **O(E|f*|)**

Huffman: Bruker O(n) tid på *build-min-heap.* Utfører en for-loop som trenger O(lg n) på hver iterasjon. Gir **O(n lg n)**

Konsepter

Amortisert analyse

Metode for å analysere tidskompleksiteten til en algoritme. Tar gjennomsnittet av operasjonene basert på forskjellige input og andre faktorer som spiller inn på tidsbruk. Forskjellig fra vanlig gjennomsnitt-analyse ved at den ikke bruker sannsynlighet. F. eks. i dynamisk tabell.
«Amortisert analyse garanterer gjennomsnittlig ytelse for hver operasjon i worst case».

Asymptotisk notasjon

Skal finne tiden for algoritmen når n → ∞

Vi måler antall fundamentale operasjoner, ikke faktisk tid.

Bryr oss ikke om konstanter og polynomer av mindre grad enn den største.

Stor O-notasjon: Upper bound, sier kun noe om øvre grense. «For en stor nok n er kjøretiden maks k*f(n) for en konstant k.»

O: f(n) = O(g(n)) , f ≤ g

Stor Ω-notasjon: Lower bound, sier kun noe om en nedre grense, brukes ikke veldig mye. «For en stor nok n er kjøretiden minst k*f(n) for en konstant k.»

Ω: f(n) = Ω(g(n)) , f ≥ g

Stor Θ-notasjon: Tight bound, her er f(x) skvist mellom to funksjoner. «For en stor nok n er kjøretiden minst k_1*f(n) for en konstant k_1 og maks k_2*f(n) for en konstant k_2.»

Θ: f(n) = Θ(g(n)) , f = g()

o: f(n) = o(g(n)) , f < g

ω: f(n) = ω(g(n)) , f > g

Vokser sakte (i stigende rekkefølge):

- Log n, ln n (Logaritmer vokser like fort uavhengig av grunntall):
  Logaritmisk kjøretid, skjer når en halverer problemstørrelsen hver gang man tester et element. F. eks. når man søker etter et element i en liste som er sortert og man kan se bort fra mange elementer for hver gang man tester.
- √n
- n (polynomisk kjøretid. Her går man gjennom alle elementene en gang.)
- n log n
- n^2 (polynomisk kjøretid. Om man skal gå gjennom en matrise med n kolonner og n rader.)
- n^3 (For eksempel hvis man skal gå gjennom en matrise med n kolonner og n rader, n ganger (Floyd-Warshall).

Vokser raskt (i stigende rekkefølge):

- 2^n (Eksponentialfunksjoner, denne vil man unngå‼)
- n!
- n^n

Caser:

Har med hvordan inputen ser ut.

Best case: Beste mulige kjøretid en algoritme kan ha.

Worst case: Verste mulige kjøretiden en algoritme kan ha.

Får du for eksempel oppgitt at best case har kjøretid Θ(n) og worst case har kjøretid Θ(n^2).

Da kan vi si at algoritmen er O(n^2) og Ω(n), fordi den aldri kan kjøre 3Gre enn worst case eller raskere enn best case.

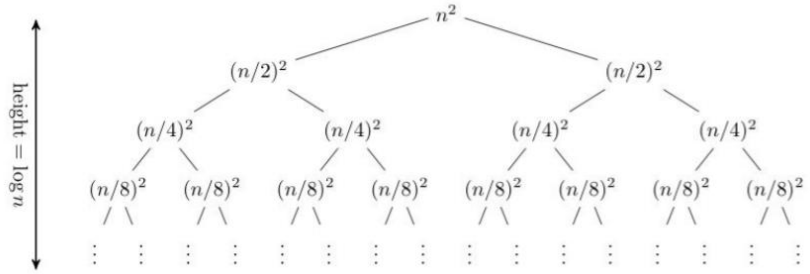
Rekurrensløsning (m/mastermetoden)

Bruk variabelskifte for å forenkle funksjonene!

For å finne kjøretiden til rekurrensligninger finnes tre metoder:

- Substitusjonsmetoden
    - Gjetter på løsningen og beviser ofte ved matematisk induksjon. Noen forslag til gjetninger er ofte av Θ av n, n^2, n^3, enten alene eller multiplisert med log(n)
- Rekurrens3metoden
    - Konverterer rekurrensen til et tre hvor hver node representerer kostnaden av et enkelt underproblem. Summer kostnadene per nivå og deretter alle nivåene for å finne total kostnad.

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



- Masterteoremet

- Krav for å bruke masterteoremet: Ligning på formen **T(n) = aT(n/b) + f(n)**, med a ≥ 1 og b > 1 (konstanter‼) og f(n) en asymptotisk positiv funksjon. Da har vi tre caser:
  - $f(n) = O\left(n^{\log_b a - \varepsilon}\right) for\ \varepsilon > 0 \rightarrow T(n) = \Theta(n^{\log_b a})$
  - $f(n) = \Theta\left(n^{\log_b a}\right) \rightarrow T(n) = \Theta(n^{\log_b a} * \log(n))$
  - $f(n) = O\left(n^{\log_b a + \varepsilon}\right) for\ \varepsilon > 0\ hvor\ a * f\left(\frac{n}{b}\right) \leq c * f(n)\ for\ en\ konstant\ c < 1\ for\ alle\ store\ n \rightarrow T(n) = \Theta(f(n))$

Eksempel:

T(n) = 9T(n/3) + n

a = 9, b = 3, f(n) = n

Da ser vi at $n^{\log_b a} = n^{\log_3 9} = n^2$, og siden $n^2 > f(n) \rightarrow T(n) = \Theta(n^2)$

Lær **variabelskifte** i tilfelle man har $t(\sqrt{n})$ f. eks. hvis ikke er du n00b

Noen rekurrenser kan ikke løses med mastermetoden

The following equations cannot be solved using the master theorem:[3]

- $T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$
  a is not a constant; the number of subproblems should be fixed
- $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$
  non-polynomial difference between f(n) and $n^{\log_b a}$ (see below)
- $T(n) = 0.5T\left(\frac{n}{2}\right) + n$
  $a < 1$ cannot have less than one sub problem
- $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$
  f(n), which is the combination time, is not positive
- $T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n)$
  case 3 but regularity violation.

Disjunkte set data-strukturer

- MAKE-SET(x):
  Lager et nytt set hvor x er det eneste elementet og dermed også representant.
  Krever også at x ikke er i noen andre set
- UNION(x,y):
  Setter sammen set x og y, gjerne S_x og S_y. Forventer på forhånd at settene er

disjunkte. Teoretisk fjerne vi begge settene og lager et nytt i universet, men i praksis absorberer ofte den ene den andre.

- FIND-SET(x):
  Returnerer en peker til representanten av det unike settet x er element i.

Binære søketrær

- Forventet høyde er O(lg n)
- Operasjonene er proporsjonalt med høyden
-

Splitt og hersk

DIVIDE-CONQUER-COMBINE

Divide: Del problemet i mindre biter som er instanser av samme problem

Conquer Løs delproblemene rekursivt helt til man kommer til case

Combine: Kombiner løsningene

Eks: En kortstokk hvor alle trekker ut det minste kortet, til slutt blir de samlet inn igjen i motsatt rekkefølge → Den er nå sortert

Lavere worst case tid enn f. eks. insertion sort.

Brukes blant annet i mergesort og quicksort

Dynamisk programmering

Brukes til optimaliseringsproblemer hvor vi ønsker å finne den beste løsningen. Ønsker å dele opp problemet i mindre delproblemer, slik som i splitt og hersk. I motsetning til splitt og herk hvor delproblemene er disjunkte, er de i dynamisk programmering overlappende og flere overordnede problemer kan avhenge av samme delproblem. Utnytter at delproblemene er overlappende og sparer tid på å slippe å regne ut de samme tingene flere ganger. Vi vil ikke ha eksponentiell kjøretid!

For at man skal kunne bruke dynamisk programmering må problemet ha en **optimal substruktur**. Det vil si at den optimale løsningen på delproblemene dermed kan brukes til å finne løsningen på problemet. Optimal delstruktur innebærer også at delproblemene må være uavhengige. Et eksempel er om vi skal finne korteste sti fra A til C via B, da inneholder løsningen den korteste veien fra A til B og fra B til C.

Man kan også bruke splitt-og-hersk på slike problemer, men det vil være mer effektivt å bruke dynamisk programmering. Med splitt-og-hersk ville man ha funnet løsningen på samme delproblem flere ganger, men hvis man bruker **memoisering/top-down** (lagring av løsningen på delproblemene i en tabell vil man unngå dette ved å slå opp i tabellen før man evt. finner løsningen. Dette er lurt dersom man ikke trenger å løse absolutt *alle* underproblemene for å løse hovedproblemet. En annen mulighet er å bruke **bottom-up** hvor vi bygger nedenfra fra base case og regner ut alle verdier opp til vi har løst problemet.

Fremgangsmåte for dyn. prog.:

1. Karakteriser strukturen til en optimal substruktur med overlappende delproblemer. Dvs. finn ut om problemet har overlappende delproblemer, og hvordan de ser ut.
2. Rekursivt finn løsning på den optimale substrukturen. Begynn med base-case og jobb bottom-up. Og husk valgene slik at du for i eksempel *stavkutting* husker hvor det er best å kutte.
3. Finn optimal løsning på problemet ved å kombinere løsningene på delproblemene.

Algoritmer som bruker dynamisk programmering: stavkapping, knapsack 0-1

Grådighet

Løser samme type problem som dynamisk programmering, men problemet må i tillegg til å ha en optimal substruktur også ha **grådighetsegenskapen**. Denne går ut på at den optimale løsningen inneholder det valget som virker best i øyeblikket, uten å ta hensyn til hvilke valg den har i fremtiden.
Grådige algoritmer er som regel enklere å implementere og bruker mindre ressurser enn dynamiske algoritmer.

Algoritmer som bruker grådighet: fractional knapsack, aktivitetsplanlegger, Huffman

(Grafteori)

Implementasjon av grafer bruker enten nabolister eller -matriser for å vise naboskap. Lister brukes helst ved enkle grafer hvor |E| er mye mindre enn $|V|^2$. En naboliste bruker lenkede liste hvor liste på posisjon *i* forteller hvilke noder som er naboene til node *i*.

Traversering

En graf kan traverseres ved å gå gjennom alle kantene for å utforske alle nodene. Det er to tilnærminger for traversering; bredde først og dybde først. Se algoritmekapittelet.

Minimale spenntrær

Grafer kan forenkles til minimale spenntrær ved at noen kanter fjernes. Et minimalt spenntre er et tre som besøker alle nodene nøyaktig en gang. Det minimale spenntreet finnes som en delmengde av den opprinnelige grafen. Hvis det er n noder der det n-1 kanter, og et minimalt spenntre vil også ha at summen av kantenes vekter er minimal. Det kan gjerne finnes flere MST for samme graf. Kan bruke enten Kruskal eller Prim for å lage et minimalt spenntre.

Korteste vei – én til alle

Finne korteste vei fra én node til alle de andre nodene i grafen. Negative kanter kan være ok, men negative sykler er problematisk. Kan benytte relaxing for å finne ut om å gå via en spesifikk node vil gjøre veien kortere. Nyttig i ruting o.l.

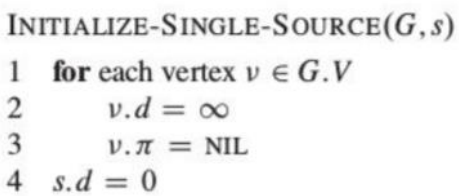
Algoritmer: Dijkstra, Bellman-Ford, DAG, BFS.

Korteste vei – alle til alle

Finne korteste vei fra mellom alle nodene i en graf. Bruker dynamisk programmering, med sekvens- og naboskapsmatrise.
Algoritmer: Floyd-Warshall

Relaxing

1. Initialize-single-source: Setter avstanden til alle noder til ∞ og avstanden til startnoden til 0. Målet med relaxing er å finne korteste avstand fra en node til en annen, og ved å sette den til uendelig vil vi alltid finne



noe bedre dersom det finnes.

2. Relax: Sjekker om den beste distansen til v (v.d) som vi har funnet til nå er større enn distansen til u (u.d) + veien/kanten fra u til v (w(u,d)), og oppdaterer v.d dersom dette var en mindre kostbar vei.

```
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
```

Maksflyt

I et maksflyt-problem skal man finne den største flyten fra en gitt kilde til et gitt sluk uten å overskride noen kapasitetskrav. Kapasitetskravene er som følger:

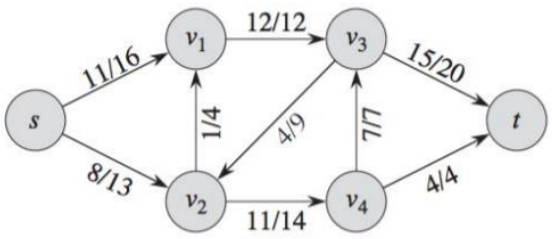
1. *Kapasitet:* flyten må ikke overstige kapasiteten
2. *Symmetri:* Positiv flyt én vei tilsvarer like stor negativ flyt andre vei. Tenk på strøm av elektriske kretser
3. *Bevaring av flyt:* Flyten **inn** må være like stor som flyten **ut** av en node. Tenk Kirchoffs lover.

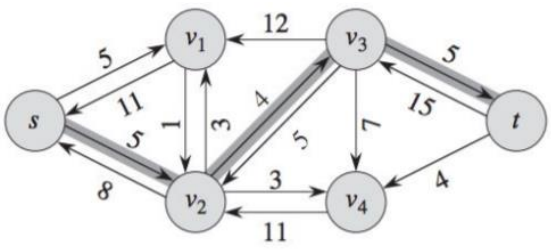
Flytnettverk

Et flytnettverk G = (V,E) er en rettet graf hvor hver kant har en ikke-negativ kapasitet. *Kilden* s produserer flyt og *sluket* t tar imot flyt. Flyt står over streken og kapasitet står under. Se figuren under til venstre. Det kan være flere kilder og flere sluk. Da kan det være lurt å legge til en superkilde og et supersluk slik at vi kan tenke på det som kun én kilde og ett sluk.

Residualnettverk

Residual = rest. Et nettverk som beskriver gjenværende flyt etter at noe av kapasiteten er brukt. Figuren til høyre er residualnettverket til figuren til venstre. Vi kaller det opprinnelige nettverket G og residualnettverket G_f





Flytforøkende vei

En flytforøkende vei er en enkel sti fra kilde til sluk i residualnettverket som forteller oss at vi kan sende mer flyt langs denne stien (Se grå sti i G_f over).

Snitt

Et snitt er en partisjon av nettverket slik at kilden og sluket ligger på hver sin side av snittet. Her er det forskjellige begreper vi må kunne

- Nettoflyt over et snitt: Summen av flyten inn i kuttet minus flyten inn i kuttet fra andre siden
- Kapasitet over et snitt: Summen av kapasitetene til kantene som går FRA u i S til v i T
- Minimalt snitt: Et snitt med minimal kapasitet

Det å finne et minimalt snitt = maksimal flyt! Dette kan vi bruke *FORD-FULKERSON*-metoden til å løse.

NP-kompletthet

Vil i hovedsak handle om NPC-problemer

NP: Problemer som kan verifiseres i polynomisk tid (gitt en løsning).
P: Problemer som kan løses (og dermed verifiseres) i polynomisk tid, altså easy.
NPH: Problemet er minst like vanskelig som alle andre problemer i NP, altså hard.
NPC: Problemet er minst like vanskelig som alle andre problemer i NP og kan verifiseres i polynomisk tid. Disse er ofte veldig like problemer som man vet kan løses i polynomisk tid, men med en liten forskjell. Det er ukjent om disse problemene er easy eller hard.
Co-NP: P + (NP)'. Dvs det motsatte av NP, men inkludert P.

1 Problemklassifisering:

- Lett/easy: Kjøretiden kan begrenses av et polynom på formen n^k der k er en konstant.

- Vanskelig/hard: Bruker superpolynomisk tid, som vil si at det IKKE har en kjøretid som kan uttrykkes på formen $T(n) \leq n^k$ for en konstant k.

- Bestemmelsesproblem (BP): Har som svar JA eller NEI. F. eks finnes det en vei mellomnode A og B. NPC problemer er BP.

- Optimaliseringsproblem (OP): Vil finne løsningen som er best. F. eks. hva er korteste vei

mellom node A og B? NPC er ikke OP, men man kan ta i bruk forholdet mellom BP og OP likevel ved å omformulere et OP til en BP. F. eks. finnes det en vei som har færre kanter enn k mellom node A og B? Optimaliseringsproblemer kan IKKE være NPC, men er ofte NPH.

2 Hvilke problemer gjelder NP-Complete for?

Teorien for NPC gjelder bare for bestemmelsesproblemer, men vi vil ofte vite hvorvidt vi kan finne en algoritme for et optimaliseringsproblem er at for å verifisere om en løsning er optimal må man finne den optimale løsningen. Dvs. det er like vanskelig å verifisere et OP som å løse det. NPC-problemer kan verifiseres i polynomisk tid, så dermed kan ikke OP være NPC, men enten NPH eller P. Derfor finner man ofte et relatert BP, og så finner man ut hvorvidt det er NPC. En kan vite at et bestemmelsesproblem er hvertfall ikke vanskeligere enn et relatert OP, så forholdet mellom vanskelighetsgraden deres kan beskrives OP ≥ BP. Dette brukes til å vise at dersom et BP er vanskelig er et relatert OP også vanskelig, og motsatt: hvis et OP er lett, er tilhørende BP også lett. Dermed har NPC et bruksområde for optimaliseringsproblemer, selv om teorien bare gjelder for bestemmelsesproblemer. Dersom et BP er NPC er tilhørende OP NPH. Dersom et OP er P er tilhørende BP også P.

3 Hvordan vise at et problem er NPC?

En vanlig måte å vise at et problem A er NPC er å finne et annet problem B som er NPC. Dersom vi klarer å redusere fra B til A vet vi at et A er minst like vanskelig som B, og at A også er NPC.

4 Hva er fordelen med å klassifisere et problem som NPC?

Da trenger man ikke sløse tid på å alge en generell algoritme som løser den type problemer, men lage en approksimasjon av løsningen (i små systemer kan problemet kanskje løses nøyaktig). Derimot hvis man skulle funnet en løsning på ett NPC-problem får man 1 000 000 dollar og det er vel nok til å løse alle dine prrrroblemar😊, og alle andre NPC-problemer (og NP-problemer for så vidt).

Reduksjon

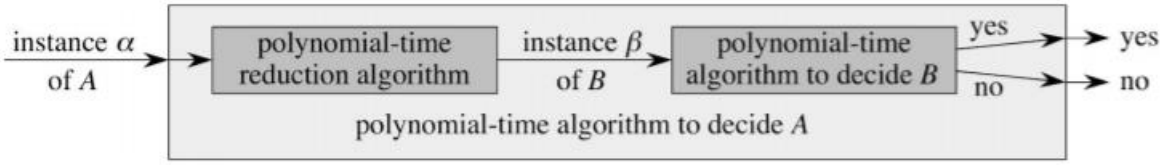
Et problem A kan reduseres til et problem B hvis det finnes en algoritme som løser problem B og denne algoritmen også kan brukes som subrutine(delalgoritme,delfunksjon) for å løse

A.

NB!: Viktig å skille mellom vanskelighetsgrad og kjøretid.

Hvis vi reduserer A til B vet vi:

- B er minst like vanskelig som A



Kostnader for å løse **A** = M*(kostnad for å løse **B**) + kostnad av reduksjonen (M er her antall ganger **B** må løses, det kan ikke være alt fra 1 til n^k ganger, men hvis M ikke kan skrives på formen n^k kan ikke A reduseres polynomialt til B).

Reduksjon betyr ikke nødvendigvis å gjøre et problem lettere å løse, men at man forandrer det til formen på et annet problem, slik at den kan løses som det andre problemet. Reduksjonen må kunne gjøres i polynomisk tid.

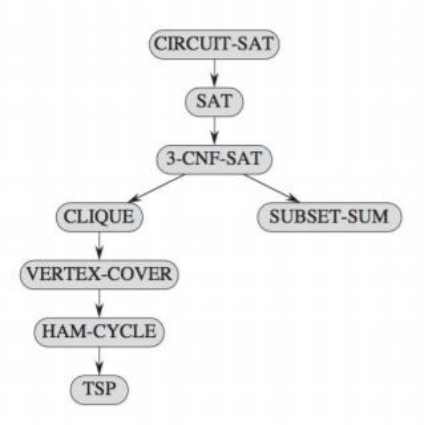
Eksempel 1:

> Gitt at A er problemet med å løse 2. gradsligninger, der a, b og c er input og x1 og x2 er output.
>
> Da kan problemer B på formen bx + c = 0 reduseres til alle inputer av problem A slik: a=0, b=b og c=c. Slik kan man redusere alle lineære ligninger til 2. gradsligninger.

Eksempel 2:

- Gitt ett problem A i NP. Hvordan kan vi vise at A er i NPC?
  Se for deg 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.
- Dette gjør vi ved å redusere **B** til **A**. Da vet vi at A er minst like vanskelig som B, og dermed at A også er NPC.

NPC/NPH problemer:



- **0-1 *Knapsack***
    - Kjøretid $\Theta(nW)$ eller $T(n,m) = \Theta(n^2m)$, der n er antall gjenstander, W er størrelse på ryggsekken og m er antall bits i W.
    - Bestemmelsesproblemet (Kan man oppnå en verdi som er minst V, gitt en maksvekt W?) er NPC, mens optimaliseringsproblemet (Hva er den største verdien V gitt maksvekt W?) er NPH. Dette kommer av at BPet kan verifiseres i polynomisk tid, mens optimaliseringsproblemet kan ikke det.
- **Lengste enkle vei**
    - Kan vises at er NPH ved at Hamiltonian-path (Som er ganske likt Ham-cycle) kan reduseres til det 'Lengste enkle vet'-bestemmelsesproblem (der man sjekker om en graf inneholder en enkel vei av lengde k eller lenger)
    - Optimaliseringsproblemer kan ikke være NP-komplette, og det er minst like vanskelig som tilhørende bestemmelsesproblem → Lengste enkle vei er NPH.

- **CIRCUIT-SAT**
    - Problem:

      Gitt en boolsk kombinatorisk krets bestående av OG, ELLER, og IKKE porter, er det mulig å få output 1?
    - Dette problemet brukes i hardware optimalisering, for hvis en krets alltid gir output 0 kan den erstattes av en simplere krets.
    - Det finnes 2^k kombinasjoner, der k er inputs
    - Bevis:

        - Bevis at det er mulig å sjekke at en løsning gir output 1 i polynomisk tid → Problemet er NP
        - Bevis at problemet er minst like vanskelig som alle andre i NP, og siden det er en del av NP og NPH → NPC. (L ≤ CIRCUIT-SAT for alle språk L ∈ NP)

- **SAT**
    - Problem: Likt som CIRCUIT-SAT, bare gjelder for generelle boolske formler.

      Består av:
        - n boolske variabler: x_1, x_2, …, x_n
        - m boolske funksjoner med én eller to boolske input, én output, som (AND), (OR), (NOT), (implication), (if and only if)
        - parenteser
    - Bevis: Likt som CIRCUIT-SAT for å vise at det er i NPH, og så vise CIRCUIT-SAT ≤ SAT, slik at SAT er NPC
- **3-CNF-SAT**
- **CLIQUE**
    - Problem:

      Gitt graf G, finnes det en delgraf G' av størrelse *k* der alle nodeparene har en kant mellom seg?
    - Bevis:

      I NPH: Redusere 3-CNF-SAT til CLIQUE

      I NP: Kan i polynomisk tid sjekke at for hver u, v ∈ G' er (u, v) ∈ E
- **VERTEX-COVER**
    - En Vertex-cover av en urettet graf G(V,E) er et subsett V' ⊆ V slik at for kantene (u, v) ∈ , så er u ∈ V' eller v ∈ V' (eller begge)
    - Problem: Finnes det en vertex cover av minimum størrelse k i en gitt graf?
- **Hamiltonian cycle**
    - Problem:

      Finne en enkel sykel i graf G=(V,E) som inneholder alle nodene i V
    - Bevis:

      I NPH: Redusere vertex-cover til Ham-cycle

I NP: Følge sykelen og sjekke at sykelen inneholder hver node én og bare en gang, med første node repetert til slutt.

- **TSP (Travelling Salesman Problem)**
    - *Problem*: En handelsmann skal besøke n byer, skal innom hver by nøyaktig én gang, med kostnad ≤ k. Man skal altså finne en hamilton-sykel.
    - *Bevis:* Først vise at TSP er i NP: Hvis vi gjetter på at en løsning som skal være mindre enn x km, da kan vi i polynomisk tid verifisere om denne veien er kortere enn x.

      Vise TSP i NPC: Vi ønsker egentlig å finne en hamilton-sykel med kostnad ≤ k. Og dersom vi finner en algoritme for å løse dette problemet vil det også kunne løse hamilton-sykel-problemet, som vi vet er NPC. Dermed er TSP også NPC. (Redusere Ham-cycle til TSP → TSP er NPC)
- **Subset-sum**
    - Problem:

      Gitt en mengde tall S og et tall t > 0, finnes en delmengde av S, S', at summen av tallene i S' er lik T

<u>BEVIS</u>

- Bevise at noe er P:

  Finne en algoritme som løser problemet i polynomisk tid
- Bevise at noe er NP:

  Finne en algoritme som sjekker om en gjettet løsning er riktig, og denne algoritmen kjører i polynomisk tid
- Bevise at noe er NPC:

  Vise først at problemet er i NP (Kan verifiseres i polynomisk tid), og deretter at det er minste like vanskelig som noe annet i NPC.

  (Transformere(redusere vel?))

  Lemma 34.8: If L is a language such that L' ≤ L for some L' ∈ NPC, then L is NP-hard. If, in addition, L ∈ NP, then L ∈ NPC.

Kantklassifisering

Tre-kanter: Kanter i dybde-først skogen

Bakoverkanter: Kanter til en forgjenger i DF-skogen (har sykler). *Kant som går til en forgjenger.*

Foroverkanter: Kanter utenfor DF-skogen som går fremover. *Kant som går til en etterkommer.*

Møter hvit node: Tre-kant

Møter grå node: Bakoverkant

Møter svart node: Forover- eller krysskant

Ordbok:

In place:
Sorteringsalgoritmer er kun *in place* om det kun er et konstant antall tall elementer lagret utenfor input-lista til enhver tid.

Stabil:
Elementer med lik verdi har samme rekkefølge i den sorterte lista som den opprinnelige

N00b: Ho ha! What is it good for? Absolutely nothing!

Adjacent: Nærliggende (nabo)

Intermediate: Mellomliggende

MST: Minimalt spenntre

NP: Non-deterministic polynomial

Dynamisk tabell:
Vi setter inn dobbelt så lang liste hver gang den er full, da får innsetting allikevel kjøretid O(1) for å sette inn et element på slutten

Problem: Ønsker forhold mellom input og output

Instans: Problem med et gitt input man kan løse problemet med

Problemstørrelse: Størrelsen på input (n)