Algoritmer og datastrukturer

Informasjon

Dette kompendiet er ikke fullstendig og ikke garantert korrekt. Det er ment som støtte og skrevet for egen læring.

Algdat nettside: https://algdat.idi.ntnu.no/
Ordliste en/no: Hva heter ____ på norsk?
Pensumhefte: Algoritmer og datastrukturer
Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf
Sideforskjell = +21
Wikipendium: TDT4120: Algoritmer og datastrukturer

Abdul Bari youtube: https://www.youtube.com/channel/UCZCFT11CWBi3MHNlGf019nw
Wikipedia: https://en.wikipedia.org/wiki/Main_Page
GeeksForGeeks: https://www.geeksforgeeks.org/

Pensum

Læringsmål for hver algoritme:

- Kjenne den formelle definisjonen av det generelle problemet den løser
- Kjenne til eventuelle tilleggskrav den stiller for å være korrekt
- Vite hvordan den oppfører seg; kunne utføre algoritmen, trinn for trinn
- Forstå korrekthetsbeviset; hvordan og hvorfor virker algoritmen egentlig?
- Kjenne til eventuelle styrker eller svakheter, sammenlignet med andre
- Kjenne kjøretidene under ulike omstendigheter, og forstå utregningen

Læringsmål for hver datastruktur:

- Forstå algoritmene (jf. mål Z01–Z06) for de ulike operasjonene på strukturen
- Forstå hvordan strukturen representeres i minnet

Læringsmål for hvert problem:

- Kunne angi presist hva input er
- Kunne angi presist hva output er og hvilke egenskaper det må ha

Algoritmer

Info/beskrivelse/begreper

https://www.youtube.com/watch?v=0IAPZzGSbME&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O

En algoritme er som en funksjon som tar inn 1 eller flere mengder med verdier og gir ut 1 eller flere mengder med verdier etter å ha "transformert" dem ved å gjøre nøye definert/spesifikke operasjoner på dem.

Kjøretid

For kjøretid er det viktig å se på input-size.

Part II Sorting and Order Statistics

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Input-size

The best notion for input size depends on the problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the number of items in the input—for example, the array size n for sorting. For many other problems, such as multiplying two integers, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation. Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph. We shall indicate which input size measure is being used with each problem we study.[1]

Running-time/kjøretid

- The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.

It is convenient to define the notion of step so that it is as machine-independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the ith line takes time ci, where ci is a constant. This viewpoint is in keeping with the RAM model, and it also reflects how the pseudocode would be implemented on most actual computers.

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

Utregning av kjøretid for insertion-sort blir cost ganger times. Dette kan regnes ut for best-case og worst case. Best-case får vi når algoritmen allerede er sortert. Da er tj = 1 for alle tj. Dvs. at summen av Tj-ene blir n-1 og n-2. Men for worst-case vil rekkefølgen være motsatt. Da må hvert tall sammenliknes med alle de forrige. Dermed blir tj = j og summen blir som summen av en rekke tall økende med 1. Utregningen for hvert av tilfellene blir:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

[1] "Insertion Sort - running time." https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=46 . Åpnet 29 okt. 2020.

for best-case og:

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\
&\quad - (c_2 + c_4 + c_5 + c_8)\,.
\end{aligned}
$$

for worst-case.
For best-case blir formelen ax+b som vil si lineær tid. Men for worst-case får vi ax^2+bx+c. Dette blir kvadratisk tid. Den faktiske tiden vil variere med konstantene a,b og c ut fra hvor lang tid hver enkelt operasjon faktisk tar.

Order of growth

For de fleste algoritmer trenger vi ikke en så nøyaktig running-time. Det som er viktig å se på er order of growth til algoritmen (asymptotisk notasjon). Dermed behøver vi bare se på "største" leddet.

Egenskaper

In-place

In-place vil si at den kun trenger en konstant O(1) ekstra minne. I praksis tror jeg det vil si at alle operasjoner foregår på for eks. selve arrayen. Dette er tilfelle ved insertionsort og quicksort, mens merge-sort som lager under-arrayer krever ekstra minne og er dermed ikke in-place.

Online[2]

At den kan fungere mens den mates informasjon.

Stabil sorteringsalgoritme

Elementer av samme grad/verdi vil havne i lik rekkefølge i den sorterte algoritmen som de var i den usorterte.
Eks: A1, A3, B2, B1, A2 -> A1, A3, A2, B2, B1

Sammenlikningsbasert algoritme

Sorteringsalgoritmer som baserer seg på sammenlikning

Rekursiv

Link til del om rekursiv: Rekurrenser

Greedy

Link til del om greedy: Greedy/Grådige algoritmer

Divide-and-conquer/Splitt og hersk

Link til del om splitt og hersk: Splitt-og-hersk/Divide-and-conquer

Dynamisk programmert

Link til del om dynamisk programmering: Dynamisk programmering

Optimal

Beste mulige. Kan være flere optimaler. Optimal løsning er beste løsning. Optimal algoritme er beste mulige algoritme. Optimal worst-case for sammenlikningsalgoritmer er O(n*Log(n))

Time-memory trade-off

Vi kan oppnå bedre tid om vi bruker mer minne. Veldig viktig i CS (computer Science). Brukes i memoisering, dynamisk programmering.

---

[2] "Online algorithm - Wikipedia." https://en.wikipedia.org/wiki/Online_algorithm. Åpnet 29 okt.. 2020.

Parallellisering

Når flere ting kan gjøres samtidig. Hvis man for eksempel kan dele opp deler av en array man skal sortere med splitt-og-hersk og kan da rekursivt kjører flere versjoner av samme algoritme kan man spare tid ved å kjøre de parallellt, ikke sekvensielt. Viktig og brukes mye i moderne CS på moderne maskiner.

Adaptive sort

Når deler er sortert allerede
https://en.wikipedia.org/wiki/Adaptive_sort

Typer algoritmer

Sorteringsalgoritmer

https://www.youtube.com/watch?v=kPRA0W1kECg

- Insertion sort
- Merge-sort
- Quick-sort
- Heapsort
- Bubble-sort (ikke pensum)

Søkealgoritmer

- Linear search
- Binary search

Traverseringsalgoritmer

- Preorder tree walk
- Inorder tree walk
- Postorder tree walk

MST-algoritmer

- Prim
- Kruskal

Shortest-path-algoritmer

- Dijkstra
- Bellman-Ford

Mal

Navn

Video:

Wikipedia:

Boken:

Eks:

Analogi:

Datastruktur

Egenskaper

-

Det generelle problemet den løser

Spesifikke problemer den er best på:

Kjøretid

Sortert	**O(**	
Best-case	**O(**	
Average-case	**O(**	
Worst-case	**O(**	

Størrelse i minnet

Pseudokode

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

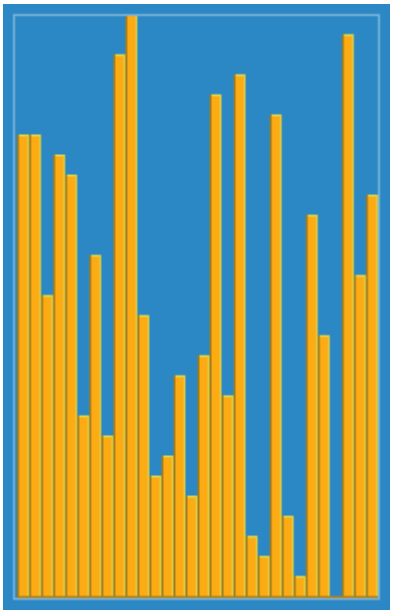
Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Versjoner

-

Insertion-Sort



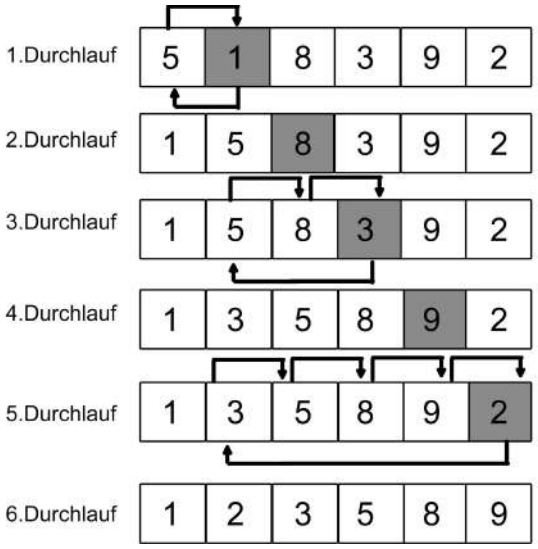
En av de enkleste algoritmene. Man går i en løkke igjennom alle elementene og for hvert element sammenlikner man med de forrige til man finner riktig plass i den da sorterte delen av arrayen.

Video: https://www.youtube.com/watch?v=JU767SDMDvA

Wikipedia: **https://en.wikipedia.org/wiki/Insertion_sort**

Boken: **https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=37**

Eks:



Analogi: Tenk sortering av kort du holder i hånden der du flytter kortene fra høyre mot venstre og sammenlikner med hvert kort du flytter hånden/kortet over.

Finnes varianter av den som er mer effektiv (**Shell-sort**), men det er ikke pensum.

Datastruktur
Den brukes på **arrayer** (pseudokoden er beskrevet på array).

Egenskaper

- Sammenlikningsalgoritme
- Stabil
- In-place
- Online

Det generelle problemet den løser
Det er en enkel sorteringsalgoritme som etter beskrevet av pseudokoden sorterer tall i en stigende rekkefølge.

Spesifikke problemer den er best på:
Det er ikke en særlig effektiv sorteringsalgoritme (bruk heller quicksort eller merge-sort), men den er veldig enkel å implementere, den er intuitiv (lett å forstå) og fungerer greit for problemer med liten input-size.
Den er en av de beste sorteringsalgoritmene hvis man vet at veldig få elementer er på feil plass. For eksempel kun 1 element er feil.

Kjøretid

Sortert	**O(n)**	Som under
Best-case	**O(n)**	Lista er allerede sortert riktig. Må likevel gå gjennom en gang.
Average-case	**O(n^2)**	Her er løkken hvert tall må igjennom for å sammenlikne med de forrige halvparten så stor som i worst-case, men allikevel vil det føre til kvadratisk tid.
Worst-case	**O(n^2)**	Oppstår når lista er sortert i synkende rekkefølge og vi ønsker den sortert i stigende

Størrelse i minnet
Siden den er in-place tar den kun opp plassen til arrayen O(n) plass. Den krever O(1) tilleggsplass.

Pseudokode

```
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
```

Forklaring på hvordan den fungerer, trinn for trinn

Går igjennom ett og ett element i en løkke fra element nr. 2 til det siste:

- Setter det elementet som ***Key***
- Går igjennom alle de tidligere elementene i synkende rekkefølge helt til den når enten index 0 eller et element som er mindre:
    - Når man har nådd enten index 0 eller det mindre elementet putter man ***Key*** på plassen til høyre

Krav for at den skal fungere (evt. tilleggskrav)

Som vist her må den brukes på arrayer.

Korrekthetsbevis, hvordan og hvorfor den virker

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=39

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

2.1 Insertion sort 19

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Styrker (+ i forhold til andre)

Den er veldig enkel å implementere, den er intuitiv (lett å forstå) og fungerer greit for problemer med liten input-size. Bedre skjulte konstanter i forhold til andre kvadratiske sorteringsalgoritmer som bubblesort.

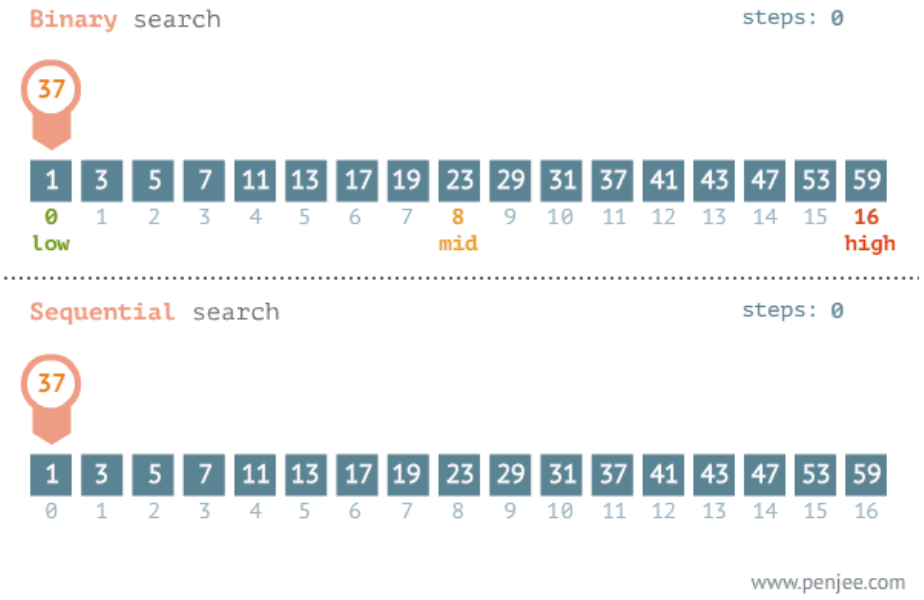
Svakheter (+ i forhold til andre)

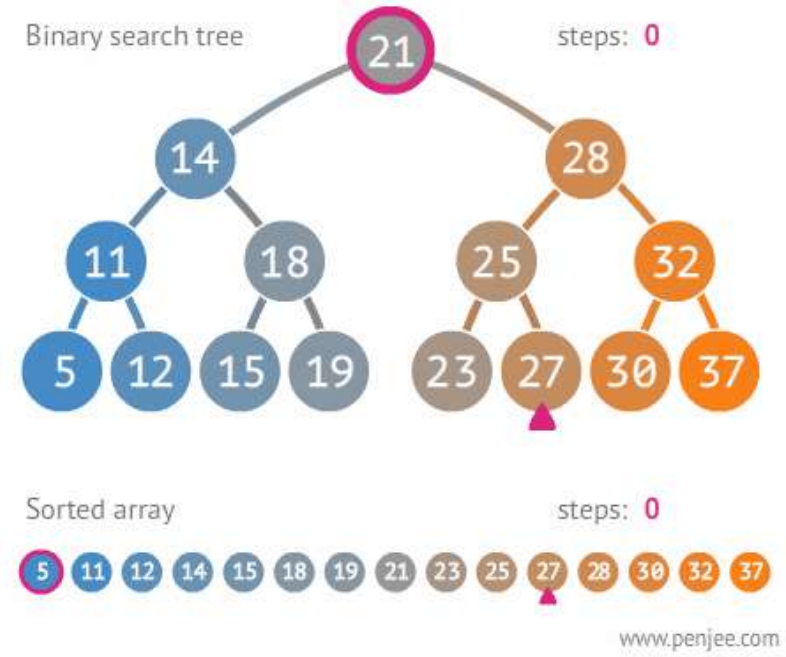
Det er ikke en særlig effektiv sorteringsalgoritme generellt (bruk heller quicksort eller merge-sort)

Versjoner

- Insertion-sort
- Shell-sort (med mange versjoner for valg av mellomrom)

Bisect (binary search/bisection) / binærsøk





Aka. Half-interval search, logarithmic search, binary chop.
Binærsøk er en søkealgoritme for å finne posisjonen til et tall i en sortert array.
Tenk gjetting av tall mellom 1-100 (gjette 50 høyere/lavere 25 høyere/lavere).
Kan også se på det som et binært-tre som er sortert dersom du ser det fra nodenes plassering fra venstre mot høyre. Høyden på treet er lg(n) rundet ned. Og det samsvarer med antall steg man må ta for å finne tallet man søker i en worst-case.
Hvis tallet ikke finnes returneres det at tallet man søkte etter ikke er i arrayen.

Video: https://www.youtube.com/watch?v=C2apEw9pgtw

Wikipedia: https://en.wikipedia.org/wiki/Binary_search_algorithm

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=60
Det er kun 1 i oppgaveteksten algoritmen blir beskrevet

Eks:

Analogi:
Du spør etter et tall i en sortert array også spør du hva som er på den midterste plassen. Hvis du vet at tallet du leter etter er mindre fjerner du den største halvparten av arrayen og gjør det samme igjen, hvis tallet er større så gjør du motsatt.

Datastruktur
Brukes på arrayer.

Egenskaper
- Rekursiv
- Splitt-og-hersk

Det generelle problemet den løser
Søkealgoritmen finner et tall man søker etter (**plassen** og **om den finnes**) i en sortert array.

Spesifikke problemer den er best på:
Jeg tror det er den beste kjente algoritmen for å søke i en sortert array.

Kjøretid

Best-case	**O(1)**	Hvis tallet er akkurat i midten av arrayen og algoritmen treffer riktig på første forsøk
Average-case	**O(lg(n))**	Når man treffer en eller annen gang mellom første forsøk og siste. Det vil være vanligere at det er nære lg(n) fordi hvis vi ser på det som et tre er de fleste nodene samlet mot bunnen (mange blader og foreldre av blader), så det er mest sannsynlig at tallet er langt ned i treet.
Worst-case	**O(lg(n))**	Oppstår når tallet er det siste tallet som treffes eller hvis tallet ikke er i arrayen

T(n)=T(n/2)+O(1)

Størrelse i minnet
O(1) auxiliary. O(n) for arrayen.

Pseudokode
```
function binary_search(A, n, T) is
    L := 0
    R := n − 1
    while L ≤ R do
        m := floor((L + R) / 2)
        if A[m] < T then
            L := m + 1
        else if A[m] > T then
            R := m − 1
        else:
            return m
    return unsuccessful
```

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)
Arrayen må være sortert i riktig rekkefølge. Evt. at man får vite om target/målet er før eller etter nåværende posisjon for hver posisjon.

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)
Raskere enn linear search. Kan implementere versjoner der man finner tallet lengst til venstre, høyre (hvis duplicates), alle versjonene av tallet man leter etter eller det nærmeste tallet. Det er altså en fleksibel algoritme.

Svakheter (+ i forhold til andre)
Hashing kan være raskere.

Versjoner
- Bisect
- Bisect*

Find-max (maksimum) på array

[3]Finner og returnerer største elementet i en array. Fungerer ved at man "antar" at første element er det største så sammenligner man det en etter en videre og bytter det ut med den man sammenligner med om den er større.
Tenk: Du har en kortstokk som ligger opp ned. Du tar det første kortet og holder på det. Så ser du på neste kort og hvis det er større kaster du det du har, hvis det er mindre kaster du det kortet. Dette repeterer du til du har gått igjennom bunken.

Hvis arrayen er sortert kunne man bare returnert det siste tallet, dermed gir det kun mening å bruke denne algoritmen på en usortert array.

Hvis man har tilleggsinfo som hva som kan være det største tallet kunne man optimalisert med å legge til en break om man når det tallet.

Det finnes andre find-max algoritmer for andre datastrukturer som kan være mer effektive pga. datastrukturens oppsett, men denne er optimal på en array som ikke er en maxheap. Altså på en tilfeldig mengde med tall.

Egenskaper

- Optimal (hvertfall beste kjøretid mulig) for det spesifikke problemet
- Online

Det generelle problemet den løser
Algoritmen finner det største tallet i en array.

Spesifikke problemer den er best på:
Det er den beste algoritmen til å finne det største tallet i en usortert array. (Ikke best i en sortert array, eller i andre datastrukturer nødvendigvis).

Kjøretid

Best-case	**O(n)**	Algoritmen må gjøre n-1 sammenlikninger uansett. Man kan se på det som at hvert tall er med i en turnering. Alle andre enn det største tallet må tape en gang for at vi skal være sikre på at vi har riktig tall.
Average-case	**O(n)**	
Worst-case	**O(n)**	Algoritmen vil aldri trenge å gjøre mer enn n-1 sammenlikninger.

[3] "Minimum" https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=235 . Åpnet 30 okt. 2020.

Størrelse i minnet
Eneste den har i minnet er det foreløpig største elementet (Det man holder) O(1) og arrayen O(n). Hvis arrayen mates online trenger den kun O(1).

Pseudokode
VIKTIG dette er pseudokoden til min. Max er motsatt.

```
MINIMUM(A)
1  min = A[1]
2  for i = 2 to A.length
3      if min > A[i]
4          min = A[i]
5  return min
```

Forklaring på hvordan den fungerer, trinn for trinn
Lager en variabel for maksverdien og setter den til første element i arrayen. Går så igjennom ett og ett element fra element 2 til siste element i en løkke. Hvis det elementet man er ved i løkken er større enn den man har i variabelen for maksverdien så bytter man den ut med det nye største elementet. Til slutt returnerer man variabelen for det største elementet.

Krav for at den skal fungere (evt. tilleggskrav)
Denne algoritmen er laget for å fungere på en array.

Korrekthetsbevis, hvordan og hvorfor den virker
Her kan man tenke seg at alle elementene er i en turnering. Hvis alle unntatt ett element har tapt en gang er det det største elementet.

Styrker (+ i forhold til andre)
Det er en optimal algoritme for å finne største element i en array usortert array.

Svakheter (+ i forhold til andre)
Ikke en hensiktsmessig algoritme å bruke hvis arrayen er sortert, da kan man bare hente ut det siste elementet. Med andre ord så er det ikke lurt å bruke den hvis vi vet hva/hvor det største elementet er fordi den har best-case O(n), ikke O(1). Kan også optimaliseres om man har tilleggsinfo som største tall eller hvis det er på en annen datastruktur.
Kan være smartere å bygge en max-heap for eksempel. Fordi den har andre nyttige egenskaper man kan bruke videre etter å ha funnet det største elementet.

Versjoner
- Vanlig max
- Min (motsatt)

Find-min (minimum) på array

[4]Finner og returnerer minste elementet i en array. Fungerer ved at man "antar" at første element er det minste så sammenligner man det en etter en videre og bytter det ut med den man sammenligner med om den er mindre.
Tenk: Du har en kortstokk som ligger opp ned. Du tar det første kortet og holder på det. Så ser du på neste kort og hvis det er mindre kaster du det du har, hvis det er større kaster du det kortet. Dette repeterer du til du har gått igjennom bunken.

Hvis arrayen er sortert kunne man bare returnert det første tallet, dermed gir det kun mening å bruke denne algoritmen på en usortert array.

Hvis man har tilleggsinfo som hva som kan være det minste tallet (for eks. 0) kunne man optimalisert med å legge til en break om man når det tallet.

Det finnes andre find-min algoritmer for andre datastrukturer som kan være mer effektive pga. datastrukturens oppsett, men denne er optimal på en array.

Egenskaper

- Optimal (hvertfall beste kjøretid mulig)
- Online

Det generelle problemet den løser
Algoritmen finner det minste tallet i en array.

Spesifikke problemer den er best på:
Det er den beste algoritmen til å finne det minste tallet i en usortert array. (Ikke best i en sortert array, eller i andre datastrukturer nødvendigvis).

Kjøretid

Best-case	**O(n)**	Algoritmen må gjøre n-1 sammenlikninger uansett. Man kan se på det som at hvert tall er med i en turnering. Alle andre enn det minste tallet må tape en gang for at vi skal være sikre på at vi har riktig tall.
Average-case	**O(n)**	
Worst-case	**O(n)**	

Størrelse i minnet

[4] "Minimum" https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=235. Åpnet 30 okt. 2020.

Eneste den har i minnet er det foreløpig minste elementet (Det man holder) O(1) og arrayen O(n). Hvis arrayen mates online trenger den kun O(1).

Pseudokode

```
MINIMUM(A)
1  min = A[1]
2  for i = 2 to A.length
3      if min > A[i]
4          min = A[i]
5  return min
```

Forklaring på hvordan den fungerer, trinn for trinn

Lager en variabel for minsteverdien og setter den til første element i arrayen. Går så igjennom ett og ett element fra element 2 til siste element i en løkke. Hvis det elementet man er ved i løkken er mindre enn den man har i variabelen for minsteverdien så bytter man den ut med det nye minste elementet. Til slutt returnerer man variabelen for det minste elementet.

Krav for at den skal fungere (evt. tilleggskrav)

Denne algoritmen er laget for å fungere på en array.

Korrekthetsbevis, hvordan og hvorfor den virker

Her kan man tenke seg at alle elementene er i en turnering. Hvis alle unntatt ett element har tapt en gang er det det minste elementet.

Styrker (+ i forhold til andre)

Det er en optimal algoritme for å finne minste element i en array.

Svakheter (+ i forhold til andre)

Ikke en hensiktsmessig algoritme å bruke hvis arrayen er sortert, da kan man bare hente ut det første elementet. Med andre ord så er det ikke lurt å bruke den hvis vi vet hva/hvor det minste elementet er fordi den har best-case O(n), ikke O(1). Kan også optimaliseres om man har tilleggsinfo som minste tall eller hvis det er på en annen datastruktur.
Kan heller lønne seg å bygge en min-heap for eksempel også bruke egenskapene dens videre etter man har funnet min.

Versjoner

- Vanlig min
- Max (motsatt)

Merge-sort

6 5 3 1 8 7 2 4

Merge-sort deler opp arrayen i mindre like store deler (ved bruk av split) helt til alle arrayene er av lengde 1 eller 0. En array av lengde 1 eller 0 er sortert, så merger den to og to sorterte arrayer oppover igjen til man har merget (slått sammen) alle arrayene og får den fullstendige sorterte arrayen.

Video: https://www.youtube.com/watch?v=mB5HXBb_HY8

Wikipedia: https://en.wikipedia.org/wiki/Merge_sort

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=51

Eks:

Analogi:

Datastruktur
Brukes på arrayer.

Egenskaper

- Divide and conquer
- Rekursiv
- Ikke in-place
- Kan parallelliseres
- Kan implementere memoisering ved natural merge-sort (usikker)
- Stabil sortering

Det generelle problemet den løser

Spesifikke problemer den er best på:

Kjøretid

Sortert	**O(n*lg(n))** **Θ(n*lg(n))**	Selv om den er sortert vil den dele opp hele arrayen i mindre arrayer og merge de. Merge har O(n) tidskompleksitet selv om venstre array er sortert mindre enn høyre.
Best-case	**O(n*lg(n))** **Θ(n*lg(n))**	Det er ingen forskjell på kjøretiden til merge-sort selv om inputen har annen rekkefølge. Det finnes en versjon som heter "Natural merge-sort" som har **O(n)** best-case.
Average-case	**O(n*lg(n))** **Θ(n*lg(n))**	
Worst-case	**O(n*lg(n))** **Θ(n*lg(n))**	https://www.youtube.com/watch?v=alJswNJ4P3U

Størrelse i minnet

O(n) plass. **O(1)** med lenkede lister i følge wikipedia.

Pseudokode

A = Arrayen

p = første element i subarray

r = siste element i subarray

```
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)
```

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)
Krever O(n) tilleggsinfo i minnet ved å måtte håndtere mange subarrayer. Dette kan være problematisk på maskiner med veldig lite minne og da kan for eks. quicksort være et bedre alternativ eller hvis det er krise så kan man bruke insertion-sort som tar veldig lite plass.
Versjoner

- "Vanlig" merge-sort
- Natural merge-sort (med O(n) best-case)
- Parallellisert merge-sort
- Bottom-up
- Top-down

Merge

Slår sammen to sorterte arrayer.
Tenk at du har to kortstokker, du åpner fra den til venstre og den til høyre, minste kortet putter du i den nye stokken.
Navn

Video:

Wikipedia: https://en.wikipedia.org/wiki/Merge_algorithm

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#pgae=51

Eks:

Analogi:

Datastruktur
Brukes på arrayer og lager to nye arrayer for hver av de to delene den skal merge.

Egenskaper

- Stabil
- Sammenlikningsbasert

Det generelle problemet den løser
Slår sammen to sorterte arrayer/to sorterte deler av en array til en sortert array.

Spesifikke problemer den er best på:
Den er veldig god til det den gjør.

Kjøretid

Best-case	**O(n)** **Θ(n)**	
Average-case	**O(n)** **Θ(n)**	
Worst-case	**O(n)** **Θ(n)**	Ingen forskjell i kjøretid.

Størrelse i minnet
O(n) som beskrevet av pseudokode, men kan muligens implementeres i **O(1)**

Pseudokode
A = Arrayen
p = første element i subarray
q = siste element i første halvdel i subarray
r = siste element i subarray

```
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
```

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Hver av subarrayene/delene av arrayen må være sortert. Størrelsen på subarrayene har ikke noe å si.

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)
Vet ikke om andre algoritmer som løser samme problem.

Svakheter (+ i forhold til andre)
Vet ikke om andre algoritmer som løser samme problem.
Versjoner

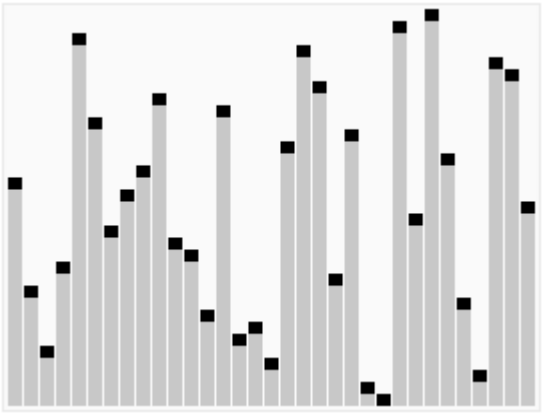
- Merge

Split

Algoritmen for å dele opp en array i to. Blir ikke brukt i vår versjon av merge. Kan brukes i quicksort for å få jevnest mulig split som er bra for kjøretiden, men det finnes mange versjoner, og de kan variere mye med hvor mye vi vet om tallene fra før.

Quick-sort og Randomized-Quicksort

Quick-sort



Også kalt **partition-exchange sort**.
En veldig god algoritme for å sortere arrayer. Selv om worst-case kjøretid er høy (O(n^2)) så er average-case lav (O(n*lg(n)) og de konstante skjulte faktorene er relativt lave, den er inplace og virker bra på virtuelt-minne[5].
Det er en rekursiv algoritme som velger et element, en **pivot**, på et delintervall og putter den på riktig plass i arrayen ved å putte alle lavere tall til venstre og alle høyere tall til høyre.

Den viktige faktoren for effektiviteten til algoritmen er hvordan vi velger ut pivotene. Her finnes typer som “median-of-three”, “random”, eller å bare velge første/siste usortert element om og om igjen (sikkert veldig dumt).

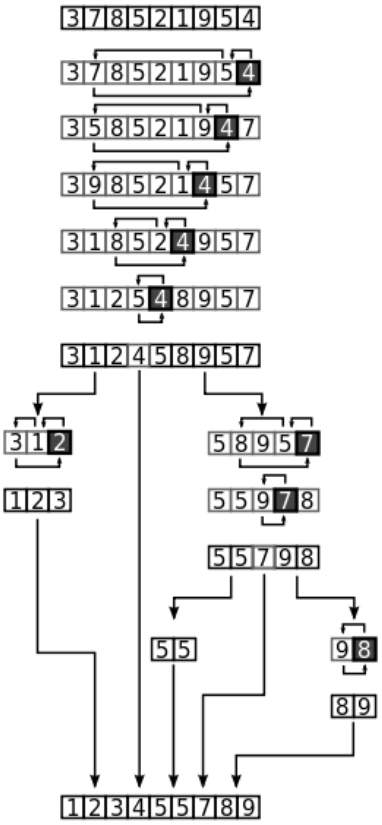
Video: [Abdul Bari Quicksort]
[Abdul Bari Quicksort analysis]

Wikipedia: https://en.wikipedia.org/wiki/Quicksort

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=191

Eks:

[5] "Quicksort." https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=191 . Åpnet 2 nov. 2020.



Analogi:

Vanskelig å komme med en intuitiv forklaring. Her er en morsom video av sorteringen:
[Quick-sort with Hungarian (Küküllőmenti legényes) folk dance]

Datastruktur

Den brukes på usorterte **arrayer** (pseudokoden er beskrevet på array).

Egenskaper

- Sammenlikningsalgoritme
- In-place
- Ikke online
- Ikke stabil
- Divide-and-conquer

Det generelle problemet den løser

Sortering av array.

Spesifikke problemer den er best på:
En av de beste algoritmene på å sortere en array. Det vil alltid være ulike situasjoner i sammenligningbasert sortering der forskjellige algoritmer er best. Men quicksort er generelt en god algoritme til sortering.

Kjøretid

Sortert	**O(n^2)**	Hvis arrayen allerede er sortert vil dessverre hver partition føre til verste tilfelle.
Best-case	**O(n*lg(n))**	Beste mulige kjøretid for en splitt-og-hersk/rekursiv sammenlikningsalgoritme ?
Average-case	**O(n*lg(n))**	Veldig god average case
Worst-case	**O(n^2)**	Sjeldent det oppstår. Det skjer når partition deler i to arrayer på **n-1** og **0** i størrelse hver gang. Da må **n** partitioner kjøres som hver tar **O(n)** tid.

Størrelse i minnet
O(lg(n)) auxiliary worst-case, men man sier allikevel at den er in-place. Auxiliary info kommer nok fra at man må holde på **pivotene**, men ikke helt sikker.

Pseudokode

```
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)
```

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker
https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=194

Styrker (+ i forhold til andre)

Regnes som en av de beste sorteringsalgoritmene fordi den har bra average case, med gode skjulte konstanter. Bedre enn merge-sort og heap-sort. Den krever heller ikke altfor mye auxiliary info, mindre enn Merge-sort. Den er in-place.

Svakheter (+ i forhold til andre)
Den har verre worst-case kjøretid i forhold til sine konkurrenter heapsort og merge-sort. Den er heller ikke stabil, så den bør ikke brukes til å sortere noe som bør være stabilt.
O(n^2) tid for en sortert array er ikke særlig bra, her ville insertion sort gitt **O(n)** i tid.

Versjoner

- Quick-sort
- Randomized-Quicksort
- Median of three

Partition

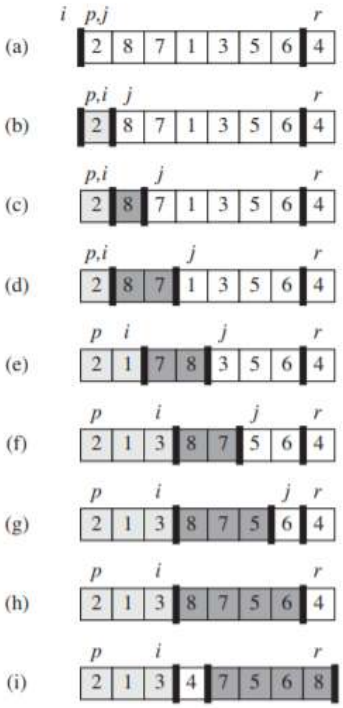


Figure 7.1 The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have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

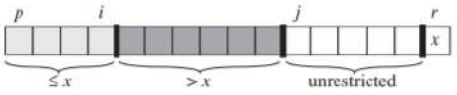


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p..r]$. The values in $A[p..i]$ are all less than or equal to x, the values in $A[i+1..j-1]$ are all greater than x, and $A[r] = x$. The subarray $A[j..r-1]$ can take on any values.

Navn

Som beskrevet ved bildet kan vi tenke oss at vi har 4 områder. 1 område for den **"lille" arrayen** (den med små verdier), 1 område for den **"store" arrayen**, 1 område for usorterte tall og siste plassen for **pivoten**. **Pivoten** er elementet som bestemmer grensen mellom den **lille arrayen** og den **store arrayen,** altså at alle tall mindre eller lik **pivoten** havner i lille og resten i store. Vi kan tenke oss at for hver iterasjon i løkken til **Partition** så putter vi neste tall

i det usorterte området i den arrayen den skal ha. Det som faktisk skjer er litt mer komplisert ved at grensene for de 4 områdene flyttes på og tall flyttes slik at det går opp.

Den andre måten å forklare er ved å ha bigger from left og smaller from right forklaringen: https://www.youtube.com/watch?v=Hoixgm4-P4M

Video: **Abdul Bari Quicksort**
Abdul Bari Quicksort analysis

Wikipedia:

Boken: **https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=192**

Eks:

Analogi: Beskrevet litt abstrakt i innledningen. Vanskelig å komme med noen intuitiv forklaring.

Datastruktur
Den brukes på **arrayer** (pseudokoden er beskrevet på array).

Egenskaper
- In place

Det generelle problemet den løser
Deler opp en array i to subarrayer der alle tall i den ene er mindre enn i den andre der grensen er satt av en **Pivot**. I dette tilfellet er pivoten det siste elementet i arrayen.

Spesifikke problemer den er best på:
Det er den kjente algoritmen for å partere en array i 2 basert på at den ene skal kun inneholde mindre tall enn den andre.

Kjøretid

Sortert	**Θ(n)**	
Best-case	**Θ(n)**	
Average-case	**Θ(n)**	
Worst-case	**Θ(n)**	

Størrelse i minnet

O(n) ingen auxiliary, men quicksort som er rekursiv må huske på pivoter fra partisjonene sine, tror jeg.

Pseudokode

```
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
```

p,r,i og j beskriver POSISJONER, ikke verdier!

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Versjoner

- Partition

Randomized-Quicksort

Ranomdized-quicksort er som quicksort, bare at den bruker randomized-partision som henter ut en tilfeldig pivot. Da vil man garantere avg-casen med n*lg(n).

Bubble-sort (ikke pensum)

En dårlig algoritme, men den brukes i pedagogikk for læring av hvordan algoritmer fungerer. Sammenlikner to og to gjennom en hel array, så gjør den det om og om igjen.
https://en.wikipedia.org/wiki/Bubble_sort
Video: https://www.youtube.com/watch?v=lyZQPjUT5B4 (02.11.20)
Eks:

Bogo-sort (ikke pensum)

O(n!) lol

Quadsort (ikke pensum)

bedre enn quicksort. Stabil
Kom i 2020
https://github.com/scandum/quadsort

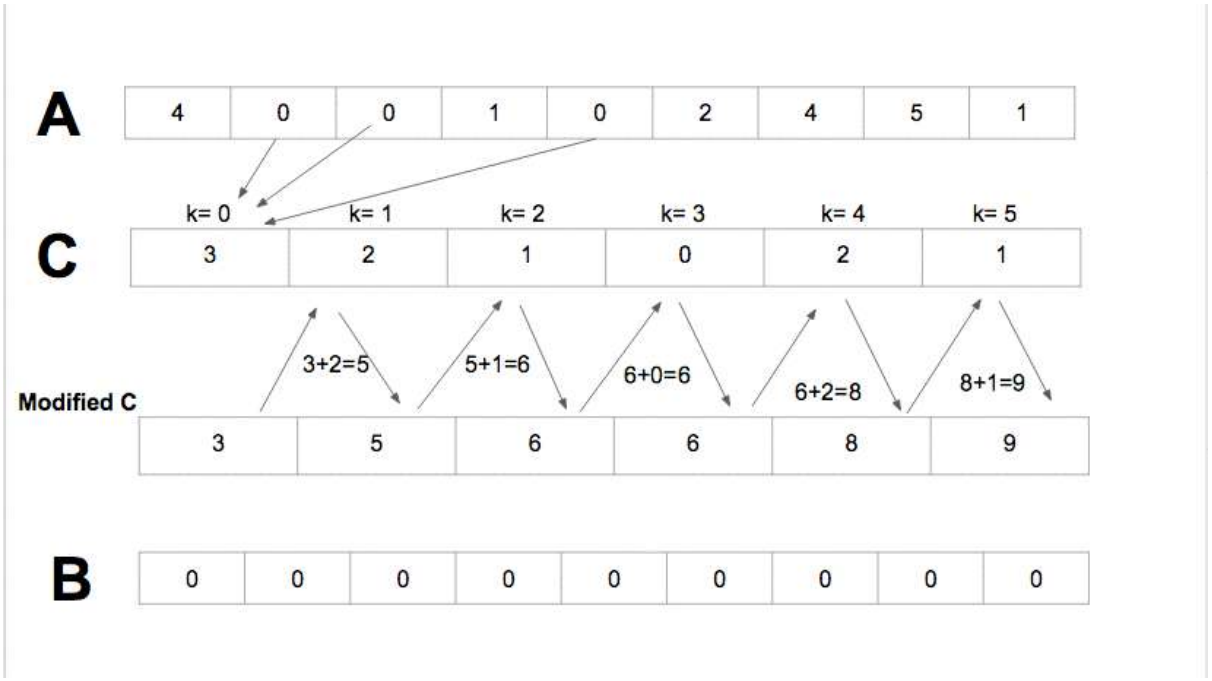
Timsort (ikke pensum)

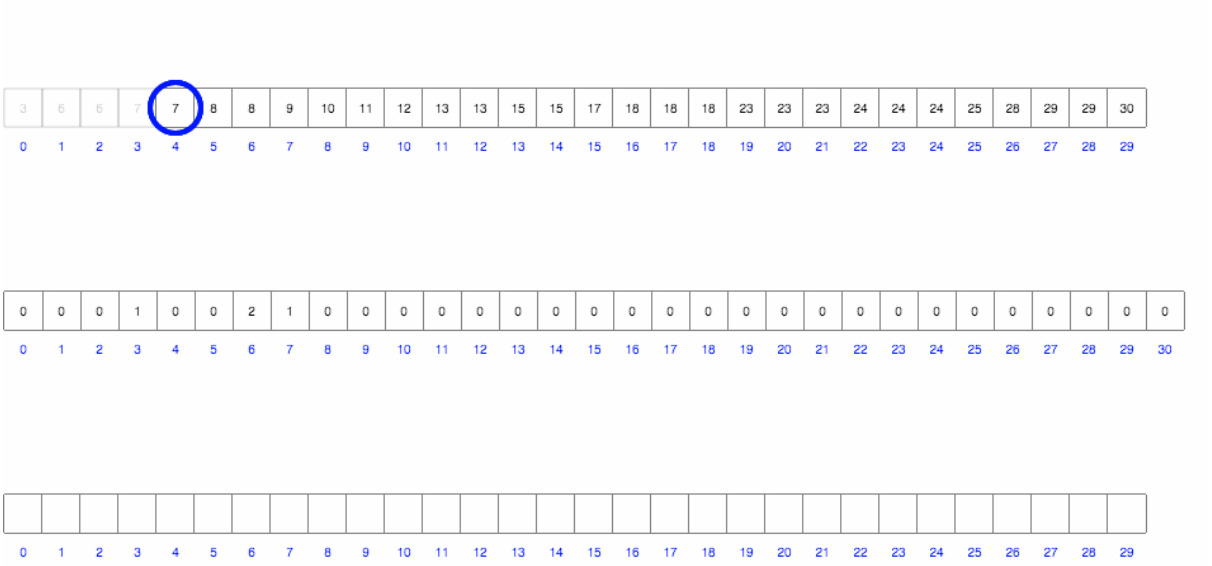
bedre versjon av mergesort 2002

Strassen's matrix multiplication

En god algoritme for matrise multiplikasjon. Vanlig matrise multiplikasjon gjøres i **O(n^3)**, men Strassen's bruker **O(n^2.81).**
https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=96

Counting-sort





https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=215

Video: https://www.youtube.com/watch?v=7zuGmKfUt7s

En sorteringsalgoritme for arrayer med tall. Den går ut på å telle opp hvor mange av hvert tall det er og putte det i en ny array (k (k-lengde der k er høyeste mulige tall). Videre putter man inn tall fra den nye arrayen k inn i en tom array av samme størrelse som n.
Denne algoritmen kan variere og optimaliseres utifra hvor stor n og k er og hvilke datastrukturer/arkitektur den kjøres på.

Den lette måten vi kodet selv med lav k stor n er at man teller opp antall k-ere i n og så itererer igjennom k og inserter antall i nye n.

Sånn som gifen over er og som det er beskrevet andre steder, vil man gå igjennom k en gang og kumulativt legge sammen de forrige tallene. For så å iterere gamle n baklengs og se i k på plassen den skal legges inn i nye n. Og da trekke fra 1 på k-plassen. Man vil da legge til hvert element fra gamle n i nye n i riktig plass, men 'tilfeldig' rekkefølge i nye n.

Eks: Egen metode. Kaste 1000 terninger. Telle antall 1-6 ere. Legge til antall 1ere i starten av arrayen så legge til antall 2…

Eks: Offisiell metode: som gif i starten.

Datastruktur

Den brukes på **array**.

//Men bør kanskje være hashtabell/annen datastruktur med direkte aksessering så man
//slipper å iterere. og nye n bør også kunne direkte aksesseres nyeN[plass] = k.
//kanskje ikke likevel

Egenskaper

- Ikke sammenlikningsbasert
- "Matematisk" basert

Det generelle problemet den løser

Sorterer array. (tall, men kan brukes på bokstaver og annet om man kan omgjøre dem/skille på størrelsen deres)

Spesifikke problemer den er best på:

Den sorterer i lineær tid og kan funke veldig bra med liten k og kjempestor n.

Kjøretid

Med offisiell metode:
Hvis k = O(n) så kjører den i O(n) tid.

Sortert	**O(**	
Best-case	**O(n)**	Hvis k = O(n) så kjører den i O(n) tid.
Average-case	**O()**	
Worst-case	**O(n+k)**	

Størrelse i minnet

Worst-case størrelse i minnet O(n+k)

Pseudokode

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // 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
```

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Den sorterer i lineær tid og kan funke veldig bra med liten k og kjempestor n. Enhver

Svakheter (+ i forhold til andre)

Versjoner

-

Radix-sort

Wikipedia: “Har også blitt kalt **bucket sort** og **digit sort**.”

Radix-sort fungerer ved at man sorterer siffervis i alle tallene som skal sorteres. Man starter - diskursiv/kontraintuitivt (antonymet til intuitivt) - i sifferet på den minste plassen. Grunnen til at dette stemmer er fordi vi må bruke en stabil sortering slik at plassene til de lavere sifrene holder seg riktig. Radix-sort er en “gruppe” som kan benytte seg av underalgoritmer til å sortere.

Video: https://www.youtube.com/watch?v=XiuSW_mEn7g

Wikipedia: https://en.wikipedia.org/wiki/Radix_sort

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=218

Eks:

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

Analogi:

Datastruktur

Brukes på array.

Egenskaper

- Ikke sammenlikningsbasert algoritme
- Framework - krever underalgoritme
-

Det generelle problemet den løser

Sortering av data som kan sorteres leksikografisk. Dvs. tall, bokstaver, ord, hullkort osv.

Spesifikke problemer den er best på:

Kjøretid

O(d(n+k)) hvis den stabile sorteringsalgoritmen den bruker kjører i **Theta(n+k).**

Sortert	**O(**	
Best-case	**O(**	
Average-case	**O(**	
Worst-case	**O(d(n+k))**	Hvis den stabile sorteringsalgoritmen den bruker kjører i **Theta(n+k)**

Størrelse i minnet

Pseudokode

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

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Kan sortere i lineær tid i worst case som er bedre enn alle sammenlikningsbaserte algoritmer. Funker bra til å sortere store tall/ord i forhold til vanlig counting sort.

Svakheter (+ i forhold til andre)

Ikke vits å bruke hvis det er få sifre i elementene.

Versjoner

- Radix-counting sort (Radix-sort med counting sort som sorteringsalgoritme)

Bucket-sort

Bucket sort, også kalt **bin sort**, antar at inputen er uniformt fordelt og kan dermed sortere i average case **O(n)**. Akkurat som counting sort, antar bucket sort noe om dataen, og til gjengjeld kan de begge sortere veldig raskt.
Den lager **n** bøtter med uniform størrelse. Itererer gjennom n en gang og putter hvert tall i en passende bøtte. Pga. uniform fordeling vil det havne få tall i hver bøtte. Insertion sort i hver bøtte vil gå veldig raskt og vi vil få average tid **O(n)**.

Bøttene er lenkede lister.

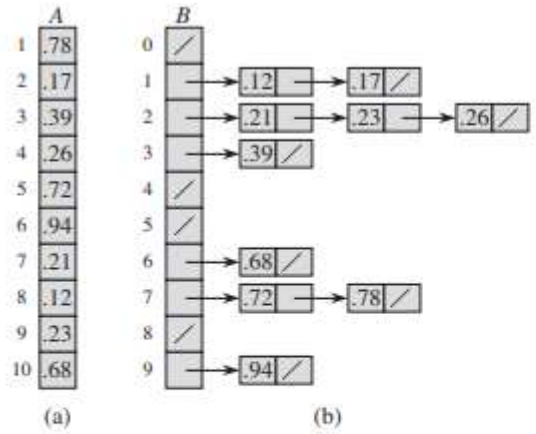


Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i + 1)/10)$. The sorted output consists of a concatenation in order of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

Video: [Bucket Sort | GeeksforGeeks]

Wikipedia: https://en.wikipedia.org/wiki/Bucket_sort

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=221

Eks:

Analogi:

Datastruktur
Brukes på arrayer.
Bøttene er lenkede lister.

Egenskaper

- Ikke sammenlikningsbasert (utenom insertionsort)

Det generelle problemet den løser

Sortering av uniformt fordelte tall.

Spesifikke problemer den er best på:

Laget for å sortere tall uniformt fordelt på det halvåpne intervallet [0-1). Veldig god average time.

Kjøretid

Sortert	**O(n)**	Blir som insertion sort sin **O(n)**
Best-case	**O(n)**	Det havner et tall pr bøtte, insertion sort behøver ikke gjøre noe. Listen er allerede sortert.
Average-case	**O(n)**	Fordi vi antar at tallene er uniformt fordelt, vil de fleste havne riktig når vi putter de i bøtter siden bøttene vil få få tall. Insertion Sort behøver nesten ikke gjøre noe.
Worst-case	**O(n^2)**	Hvis alle tallene havner i samme bøtte i feil rekkefølge må vi sortere de worst-case med insertion sort som er **O(n^2)**

Størrelse i minnet

O(n) auxiliary for bøttene

Pseudokode

```
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[⌊n A[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
```

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)
Veldig god average time på uniformt fordelte tall.

Svakheter (+ i forhold til andre)
Snevert bruk. Vi må vite at tallene er uniformt fordelt.

Versjoner

- Bucket sort

Select og Randomized-Select

Oppgaven til en select algoritme er å angi plassen på det k-ende største/minste elementet.

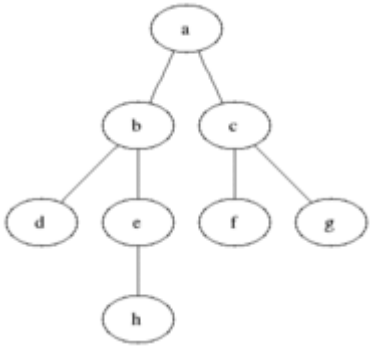
Select

Randomized-select

Heapsort

[Heapsort]

BFS (Breadth-first search)



“Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Prim’s MST(Minimum-spanning-tree) algorithm (Section 23.2) and Dijkstra’s single-source shortest-paths algorithm (Section 24.3) use ideas similar to those in breadth-first search.” - **side 615**

Veldig enkelt at du ser på en rotnode **s (source node)** (eller hva som helst annen node i en graf) så på hva den er connected til, så i tur og orden hva de er connected til. På et tre blir det å se på alle elementene i array-rekkefølgen deres.

Vi definerer noder/elementer som hvite, grå og sorte utifra om de er “uoppdaget”, “sett” eller “tatt med” respektivt.

Det blir generert et bredde-først tre ved at hver node blir lagt til når den blir svart ved at man igjen oppdager hvite noder “under”/videre fra den. Da legger man til edgen og parent noden. Siden en node kan maks bli oppdaget 1 gang, vil alle noder ha maks 1 parent.

“Froentier” vil si alle nodene på et nivå. BFS får navnet sitt fra at den oppdager alle nodene i frontieret **uniformt** før den flytter videre til neste nivå.

Hvis den brukes på en skog vil den ikke nå alle nodene.

Den funker på directed og undirected graphs.

Video: [5.1 Graph Traversals - BFS & DFS -Breadth First Search and Depth First Search]

Wikipedia: [Breadth-first search]

Boken: [https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=615]

Eks:

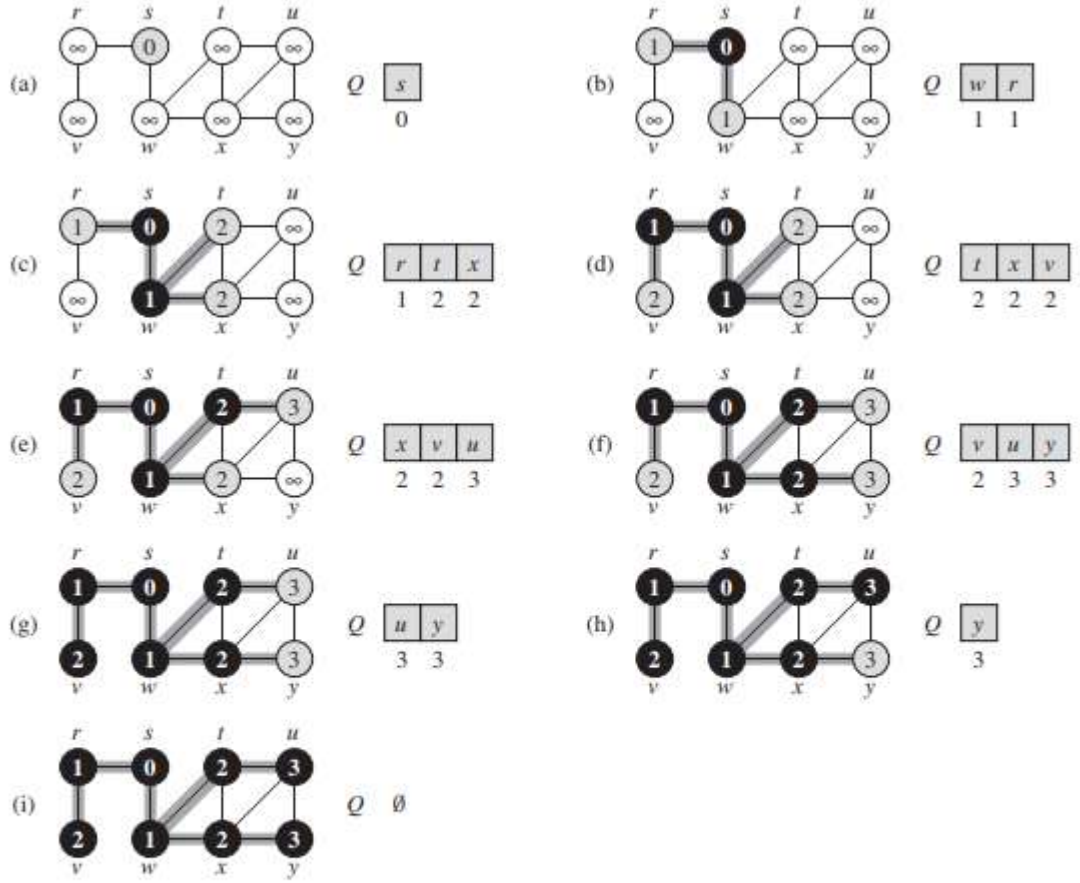


Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. The value of $u.d$ appears within each vertex u. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear below vertices in the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

Analogi: For et vanlig tre vil hver node som tas med i BF-treet være av rekkefølgen til arrayen som beskriver treet.

Datastruktur

Brukes på **grafer**. Funker på **directed** og **undirected**. Den tar ikke hensyn til **vekter** hvis det er noen, men den funker fortsatt. Den funker på **sykliske** grafer.
Pseudokoden er beskrevet på en graf som bruker **adjacency list**.

Innad i algoritmen lagrer den data i en **queue**. Dette er for å beholde rekkefølgen på hvilke noder man skal se på. Man starter med å putte inn **s**. Så putter man inn barna til **s**. Så deres barn osv. PGA **queue**-strukturen vil man da alltid ende med å dequeue det som kom inn tidligst altså barna nærmest **s**!

Egenskaper

- Ikke rekursiv

- Ikke greedy (men den finner ikke noder som ikke henger sammen med andre som skjer i skoger)
- Ikke in-place

Det generelle problemet den løser

Den finner alle nodene som er connected til source-noden **s**. Den finner også distansen til hver node (antall kanter, ikke med vekter). Den lager også et "Breadth-first tree" som inneholder alle nodene som er mulig å nå fra **s**.

Spesifikke problemer den er best på:

Andre bruksområder:

- Copying garbage collection, Cheney's algorithm
- Finding the shortest path between two nodes *u* and *v*, with path length measured by number of edges (an advantage over depth-first search)[12]
- (Reverse) Cuthill–McKee mesh numbering
- Ford–Fulkerson method for computing the maximum flow in a flow network
- Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.
- Construction of the *failure function* of the Aho-Corasick pattern matcher.
- Testing bipartiteness of a graph.

Kjøretid

Sortert	**O(**	
Best-case	**O(V+E)**	
Average-case	**O(V+E)**	
Worst-case	**O(V+E)**	Etter at alle nodene er farget hvite i starten vil de kun enqueues 1 gang (bli grå) og dequeues 1 gang (bli sort). Kø operasjonene tar O(1) hver så tilsammen O(V) tid. Adjacencylisten blir skannet når den dequeuer som vil si at hver kant blir skannet 1 gang, så tilsammen O(E) tid. Tilsammen O(V+E) tid.

Størrelse i minnet

O(|V|) den tar bare plassen av antall noder, fordi den lager et nytt tre med alle nodene. Kan også ha en stakk ved siden av som beskriver grå noder. Den er da ikke in-place.

Pseudokode

```
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
```

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Må brukes på riktig datastruktur, men har lite krav (les under datastruktur).

Korrekthetsbevis, hvordan og hvorfor den virker

The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted black (in line 18). The results of breadth-first search may depend upon the order in which the neighbors of a given vertex are visited in line 12: the breadth-first tree may vary, but the distances d computed by the algorithm will not. (See Exercise 22.2-5.)"

"Although we won't use this loop invariant to prove correctness, it is easy to see that it holds prior to the first iteration and that each iteration of the loop maintains the invariant. Prior to the first iteration, the only gray vertex, and the only vertex in Q, is the source vertex s. Line 11 determines the gray vertex u at the head of the queue Q and removes it from Q. The for loop of lines 12–17 considers each vertex in the adjacency list of u. If is white, then it has not yet been discovered, and the procedure discovers it by executing lines 14–17. The procedure paints vertex gray, sets its distance :d to u:dC1, records u as its parent :, and places it at the tail of the queue Q. Once the procedure has examined all the vertices on u's 22.2 Breadth-first search 597 adjacency list, it blackens u in line 18. - **Side 617-618**

Theorem 22.5 (Correctness of breadth-first search)
Side 620-621

Styrker (+ i forhold til andre)
Veldig lett å forstå. Kan enkelt regne ut lengder fra en til mange i antall noder.

Svakheter (+ i forhold til andre)
Man når ikke noder som ikke henger sammen med de andre. Det gjør **DFS**.

Versjoner
- BFS
- DFS (motsatt)

DFS

DFS er som en prefix path traversering i et tre. Men kan brukes på grafer (directed, ikke-directed, vektet og uvektet(vekt har ikke noe å si)) for å finne alle nodene i forhold til

startnoden, når alle de er funnet bytter den startnode om det fortsatt er flere noder igjen i grafen. Den søker så dypt så mulig og lagrer data for foreldre (grå, hvite og sorte noder osv.). Så når den har søkt så dypt den kan i en retning backtracker den til den kan søke i nye retninger. Ut av DFS får man en **DFS-skog** (som inneholder PI (foreldrenoder)) og **discoverytime u.d** (når den blir grå) og **finishtime u.f** (når den blir sort).

Vi kan få forskjellige DFS avhengig av hvilke veier vi velger hvis vi har flere i en gitt node.

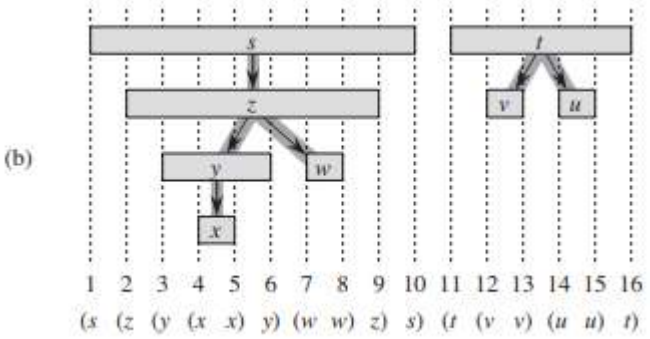
Discovery time/Starttid

u.d er når en node "males" grå, altså at den blir sett første gang.

Finish time

u.f er når en node "males" svart, altså at den legges til i DFS-treet.

Parentesform



Discovery time og finishtime har **parenthesis form**.

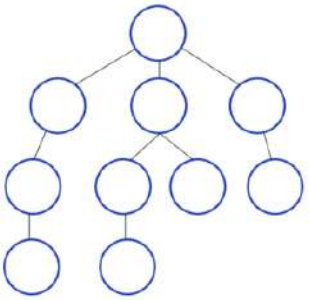
u.d < u.f
alle d og f mellom 1 og 2|V|

Kantklassifisering

DFS kan også brukes til **kantklassifisering.**
Tre-kanter er kantene i DFS-skogen.
Bakover-kantene er hvis en node peker til en ancestor node. En måte å identifisere DAGs er ved at de ikke har bakover-kanter.
Foroverkanter ikke-tre-kantene som sammenkobler en node men en descendant.
Kryss-kanter er alle andre kanter. Altså at det krysser fra "grener" i et tre.

Vi kan få et intrykk av hva slags kant vi har støtt på allerede første gang vi ser den i en DFS hvis **v** er i en kant **(u,v)**:
Hvis den er hvit -> tre-kant

Grå -> bakover kant
Sort -> forover kant eller krysskant



Video: https://www.youtube.com/watch?v=pcKY4hjDrxk

Wikipedia: https://en.wikipedia.org/wiki/Depth-first_search

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=624

Eks:

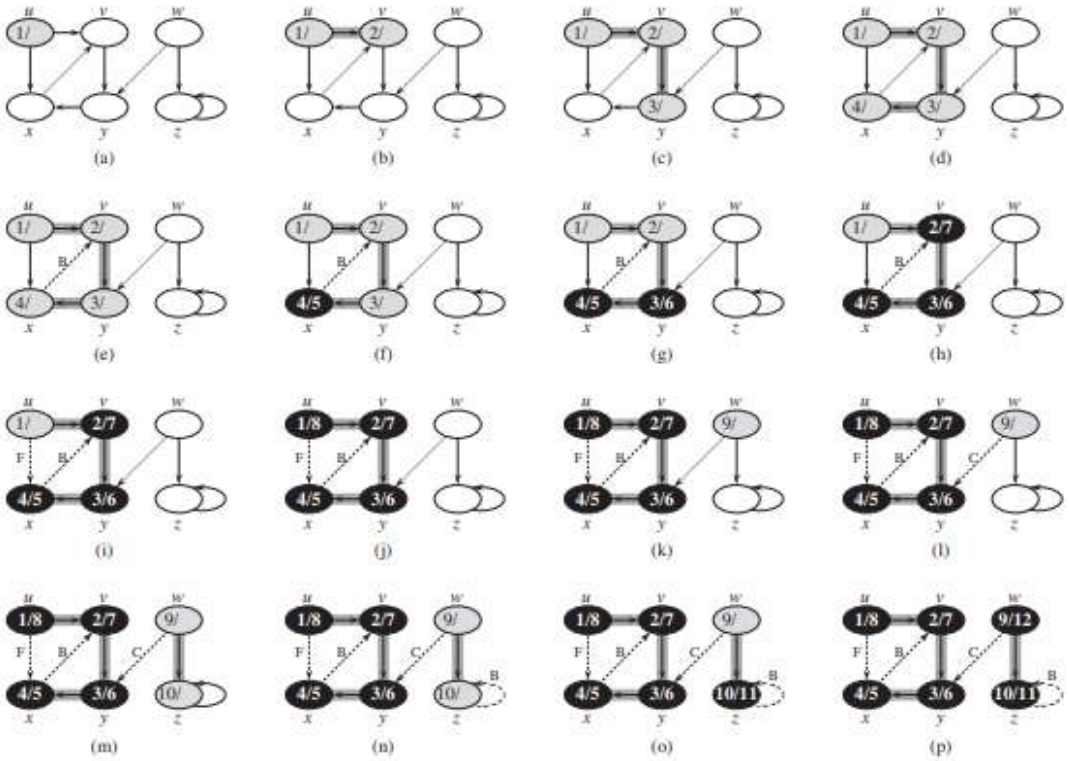


Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges.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

Analogi:

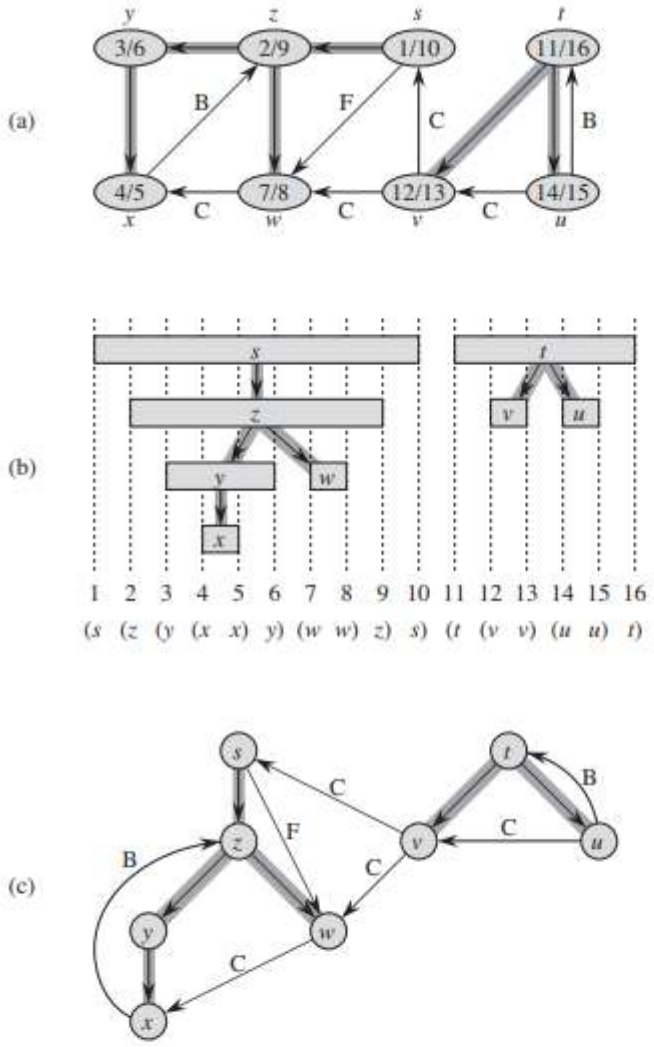


Figure 22.5 Properties of depth-first search. **(a)** The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 22.4. **(b)** Intervals for the discovery time and finishing time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finishing times of the corresponding vertex. Only tree edges are shown. If two intervals overlap, then one is nested within the other, and the vertex corresponding to the smaller interval is a descendant of the vertex corresponding to the larger. **(c)** The graph of part (a) redrawn with all tree and forward edges going down within a depth-first tree and all back edges going up from a descendant to an ancestor.

Datastruktur

Brukes på grafer. Kan være directed og undirected, weighted og ikke.

Lager et DFS-tre.

Kan tenke at den bruker en **stakk** som struktur. Ved at for hver visit legger man til noden i stakken også fjerner man den når den males svart.

Egenskaper

- Rekursvi (DFS-visit delen)

Det generelle problemet den løser

Lager en **DFS-skog** av en graf. Brukes for å se alt som henger sammen med en startnode (**traversering** og søk fra en node), **kantklassifisering** og **topologisk sortering**.

Spesifikke problemer den er best på:

- Kan brukes i kunstig intelligens og labyrint søk, men kan støte på **non-termination** som gjør at man kanskje bare søker til en viss dybde.

Kjøretid

Sortert	**O(-**	
Best-case	**O(V+E)**	
Average-case	**O(V+E)**	
Worst-case	**O(V+E)**	Omtrent ingen variasjon. Den eneste forskjellen er hvilken vei en DFS skal ta om det er flere å velge mellom.

Størrelse i minnet

O(|V|)

Pseudokode

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

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Vil finne alle noder i en skog eller rettet graf uansett i motsetning til **BFS.**

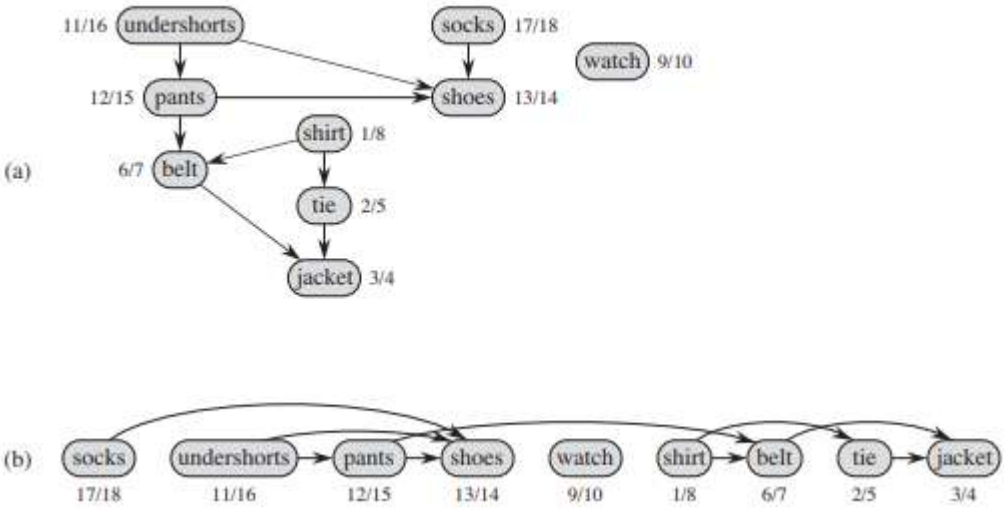
Svakheter (+ i forhold til andre)

Versjoner

- DFS
- BFS (Motsatt)

Topological-sort

Topologisk sortering er en dybde først sortering som brukes på en DAG (Directed-asyclic-graph). Tenk klesplagg.



Discovery time/finish time til **DFS** (Depth first search) på grafen.

Video:

Wikipedia: https://en.wikipedia.org/wiki/Topological_sorting

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=633

Eks:

Analogi: Tenk at du skal kle på deg. Du må ta på antrekket i en bestemt rekkefølge. Sokker før sko osv.

Datastruktur
Brukes på **DAG** (Directed-acyclic-graph).

Egenskaper

-

Det generelle problemet den løser
Den setter opp en **DAG** i en akseptabel rekkefølge på et horisontalt plan (i en lenket liste). Det kan finnes flere, men en topological sort vil gi en riktig mulighet.

Spesifikke problemer den er best på:
Vet ikke om andre algoritmer som kommer med en akseptabel rekkefølge for en **DAG**. Så i såfall er det mulig at dette er den eneste som er beskrevet i pensum.

Kjøretid

Sortert	**Θ(-**	Hvis grafen allerede er sortert etter DFS og vi har finish times, vil det bare være å putte de i en lenket liste/ e.l.(rekkefølge) Da må vi spesifikt presisere at vi ikke skal kjøre DFS, ellers vil det uansett ta **Θ(V+E)** tid.
Best-case	**Θ(V+E)**	Må uansett kjøre en DFS som tar **Θ(V+E)** tid.
Average-case	**Θ(V+E)**	
Worst-case	**Θ(V+E)**	We can perform a topological sort in time Θ(V + E), since depth-first search takes Θ(V + E) time and it takes O(1) time to insert each of the |V| vertices onto the front of the linked list.

Størrelse i minnet
O(|V|)

Pseudokode

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Theorem 22.12 TOPOLOGICAL-SORT produces a topological sort of the directed acyclic graph provided as its input.
https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=635

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Versjoner

-

Kruskal's algorithm

Kruskals algoritme brukes for å lage et **MST** (Minimum spanning tree). Det er en grådig algoritme som går ut på å velge den korteste kanten hver gang som fortsatt opprettholder MST-strukturen.
Man lager |V| trær som til sammen blir en skog. Så velger man de minste kantene for å "lappe sammen" skogen til et tre.

Video: https://www.youtube.com/watch?v=4ZlRH0eK-qQ

Wikipedia: https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=652

Eks:

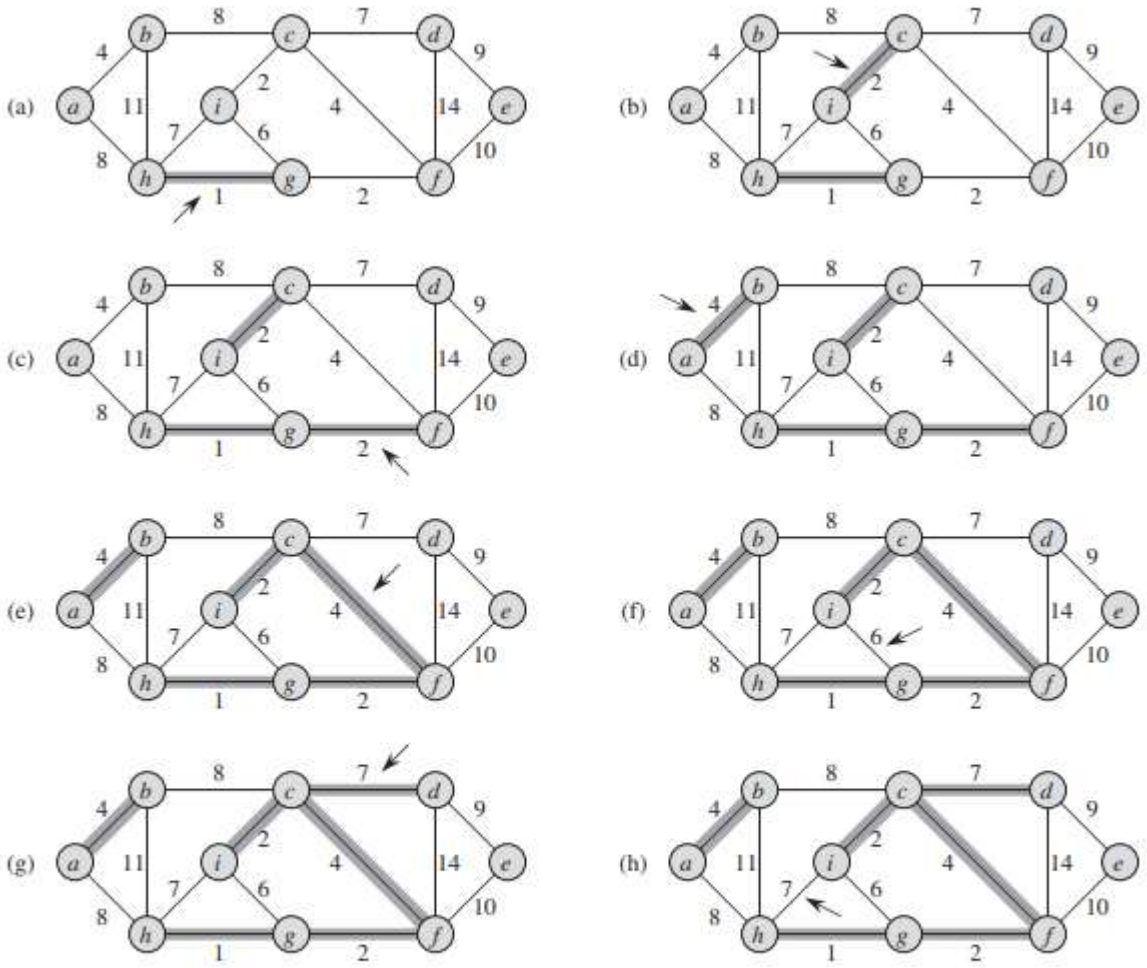


Figure 23.4 The execution of Kruskal's algorithm on the graph from Figure 23.1. Shaded edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Analogi: Tenk at du strør krydder så bare dukker det tilfeldig opp et tre.

Datastruktur

Brukes på en vektet graf for å lage en **MST** (Minimum spanning tree)

Egenskaper

- Greedy

Det generelle problemet den løser

Lager et MST av en graf med vekter.

Spesifikke problemer den er best på:

Kjøretid

O(E*lg(E)), men man sier det under

O(E*lg(V)) med binary heap

Sortert	**O(-**	
Best-case	**O(-**	
Average-case	**O(-**	
Worst-case	**O(|E|*lg(|V|)**	

The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on how we implement the disjoint-set data structure. We assume that we use the disjoint-set-forest implementation of Section 21.3 with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known. Initializing the set A in line 1 takes $O(1)$ time, and the time to sort the edges in line 4 is $O(E \lg E)$. (We will account for the cost of the $|V|$ MAKE-SET operations in the **for** loop of lines 2–3 in a moment.) The **for** loop of lines 5–8 performs $O(E)$ FIND-SET and UNION operations on the disjoint-set forest. Along with the $|V|$ MAKE-SET operations, these take a total of $O((V + E)\,\alpha(V))$ time, where α is the very slowly growing function defined in Section 21.4. Because we assume that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\alpha(V))$ time. Moreover, since $\alpha(|V|) = O(\lg V) = O(\lg E)$, the total running time of Kruskal's algorithm is $O(E \lg E)$. Observing that $|E| < |V|^2$, we have $\lg |E| = O(\lg V)$, and so we can restate the running time of Kruskal's algorithm as $O(E \lg V)$.

Størrelse i minnet

O(G) pluss den lager V trær.

Pseudokode

```
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
```

MAKE-SET(x) creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.

UNION(x, y) unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. We assume that the two sets are disjoint prior to the operation. The representative of the resulting set is any member of $S_x \cup S_y$, although many implementations of UNION specifically choose the representative of either S_x or S_y as the new representative. Since we require the sets in the collection to be disjoint, conceptually we destroy sets S_x and S_y, removing them from the collection $\mathcal{S}$. In practice, we often absorb the elements of one of the sets into the other set.

FIND-SET(x) returns a pointer to the representative of the (unique) set containing x.

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=582

Forklaring på hvordan den fungerer, trinn for trinn

1-3 initialiserer en MST **A** og |V| trær med kun en node i hver. For-løkken fra 5-8 tar for seg en og en kant i økende vekt og legger den til om den ikke lager en sykel (altså at den ikke er en forover eller bakover kant i noen av trærne som er laget), den legger sammen et og et tre i skogen.

9 returnerer MST-en **A**.

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Hvis man implementerer Prim men en fibonacci heap og |V| er mye mindre enn |E| så er kjøretiden til Prim bedre enn Kruskal.

Versjoner

- Kruskals algoritme

Prim

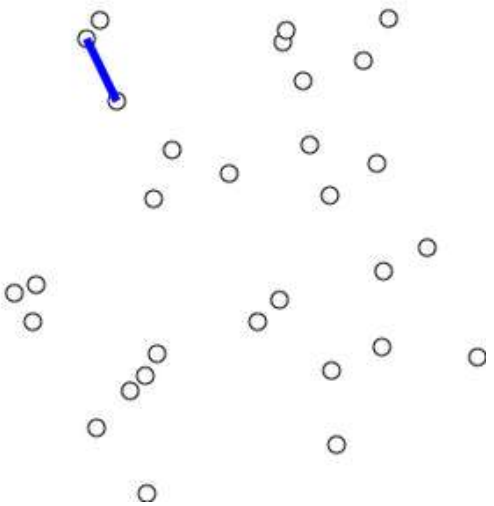
I prims algoritme starter man med en rotnode **r** og lager et tre fra den. Man legger så til en og en node som er nærmest til treet (og ikke med i det allerede). Alle nodene man ikke har lagt til ligger i min-prioritets køen **Q**.
Algoritmen minner veldig om Dijkstra for SSSP.

Video: https://www.youtube.com/watch?v=4ZlRH0eK-qQ

Wikipedia: https://en.wikipedia.org/wiki/Prim%27s_algorithm

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=652

Eks:



Analogi: Tenk at du smører ut prim.

Datastruktur

Brukes på grafer for å lage en MST (Minimum spanning tree).

Q er en min-prioritets kø.

Egenskaper

- Greedy

Det generelle problemet den løser

Lager en MST

Spesifikke problemer den er best på:

Ved bruk av fibonacci heap så får den bedre kjøretid enn Kruskal hvis |E| >> |V|

Kjøretid

O(E*lg(V)) med binary heap

O(E+V*lg(V)) med fibonacci heap som er bedre hvis |V| er mye mindre enn |E|

Sortert	**O(**	
Best-case	**O(**	
Average-case	**O(**	
Worst-case	**O(E*lg(V))** **O(E+V*lg(V))**	Binary heap Fibonacci heap

Minimum edge weight data structure	Time complexity (total)
adjacency matrix, searching	$O(\|V\|^2)$
binary heap and adjacency list	$O((\|V\| + \|E\|) \log \|V\|) = O(\|E\| \log \|V\|)$
Fibonacci heap and adjacency list	$O(\|E\| + \|V\| \log \|V\|)$

The running time of Prim's algorithm depends on how we implement the min-priority queue Q. If we implement Q as a binary min-heap (see Chapter 6), we can use the BUILD-MIN-HEAP procedure to perform lines 1–5 in $O(V)$ time. The body of the **while** loop executes $|V|$ times, and since each EXTRACT-MIN operation takes $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$. The **for** loop in lines 8–11 executes $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$. Within the **for** loop, we can implement the test for membership in Q in line 9 in constant time by keeping a bit for each vertex that tells whether or not it is in Q, and updating the bit when the vertex is removed from Q. The assignment in line 11 involves an implicit DECREASE-KEY operation on the min-heap, which a binary min-heap supports in $O(\lg V)$ time. Thus, the total time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asymptotically the same as for our implementation of Kruskal's algorithm.

We can improve the asymptotic running time of Prim's algorithm by using Fibonacci heaps. Chapter 19 shows that if a Fibonacci heap holds $|V|$ elements, an EXTRACT-MIN operation takes $O(\lg V)$ amortized time and a DECREASE-KEY operation (to implement line 11) takes $O(1)$ amortized time. Therefore, if we use a Fibonacci heap to implement the min-priority queue Q, the running time of Prim's algorithm improves to $O(E + V \lg V)$.

Størrelse i minnet

Pseudokode

```
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)
```

Forklaring på hvordan den fungerer, trinn for trinn

1-3 setter om key til alle noder som evig og at ingen noder har foreldre. Linje 4 setter key til roten til 0. Linje 5 lager en min-prioritetskø av alle nodene. Linje 6-11 kjører en løkke til alle

nodene er inkludert i MST-en der man i linje 7 alltid velger elementet med minst key **u** (rooten i første tilfelle). Så ser man på naboene til den hvis en nabo **v** ikke er med i treet og vekten mellom den valgte noden **u** og naboen **v** er mindre enn keyen til naboen **v** så settes den valgte noden **u** som forelder og keyen til naboen **v** blir oppdatert til vekten mellom **u og v**.

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

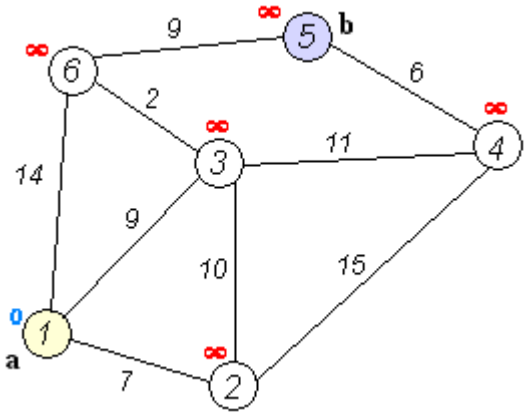
Styrker (+ i forhold til andre)
Hvis man implementerer Prim men en fibonacci heap og |V| er mye mindre enn |E| så er kjøretiden til Prim bedre enn Kruskal.

Svakheter (+ i forhold til andre)

Versjoner

- Prim med binary heap
- Prim med fibonacci heap

Dijkstra



Dijkstra's algoritme er en grådig algoritme for å finne korteste vei fra en-til-mange (SSSP). Den fungerer ved at man kjører algoritmen/funksjonen relax på hver node 1 gang, der man starter i startnoden og fortsetter alltid i noden med kortest distanse.

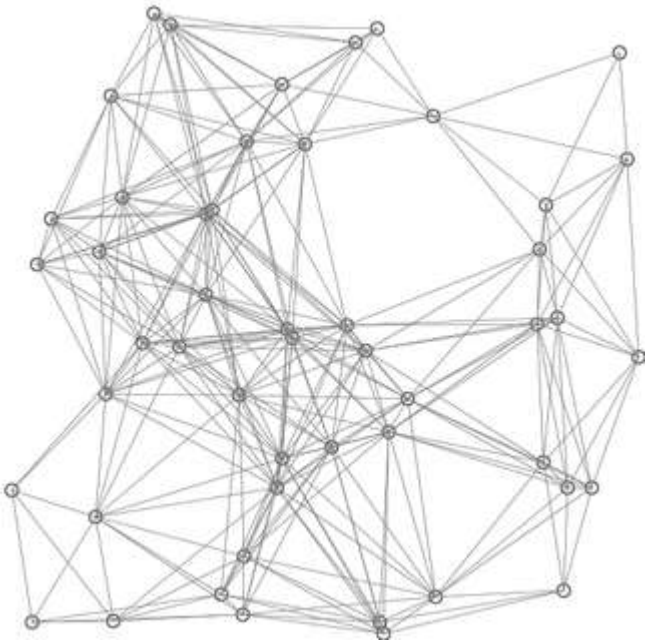
Selv om det er en grådig algoritme vil den gi riktig svar på en graf uten negative vekter. Men dersom vi har negative vekter KAN den gi riktig svar, men det er ikke sikkert. Vi sier derfor at vi kun skal bruke den på en graf med positive vekter.

Video: 3.6 Dijkstra Algorithm - Single Source Shortest Path - Greedy Method

Wikipedia: Dijkstra's algorithm

Boken: **https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=679**

Eks:



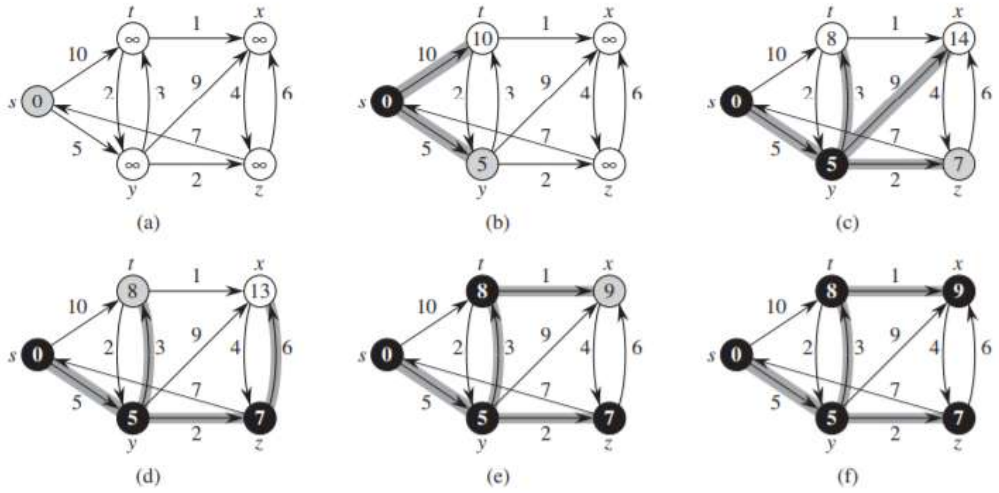


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d values and predecessors shown in part (f) are the final values.

Analogi:

Datastruktur

Brukes på grafer med vekter (kan være directed og ikke directed) som ikke er negative. Vi kan bruke den på grafer med negative vekter, men da er det ikke sikkert vi får riktig resultat og vi sier derfor at vi ikke kan bruke den på slike grafer.

Egenskaper

- Greedy
- Sammenlikningsalgoritme

Det generelle problemet den løser

SSSP (Single source shortest path). Den finner korteste vei fra en node til alle de andre i en graf med positive vekter.

Spesifikke problemer den er best på:

The idea of this algorithm is also given in Leyzorek et al. 1957. Fredman & Tarjan 1984 propose using a Fibonacci heap min-priority queue to optimize the running time complexity to **Theta(|E|+|V|*log|V|)**. This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights.

However, specialized cases (such as bounded/integer weights, directed acyclic graphs etc.) can indeed be improved further as detailed in Specialized variants.

Kjøretid

Sortert	**O(**	
Best-case	**O(**	
Average-case	**O(**	
Worst-case	**O(**	

Størrelse i minnet

Inneholder et sett **S** med alle nodene som allerede er bestemt distansen til. O(V) auxilliary

Pseudokode

```
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)
```

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Hvis den alltid skal gi riktig svar må det være kun ikke-negative tall.

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Raskere enn Bellman-Ford på grafer med kun positive vekter.

Svakheter (+ i forhold til andre)

Funker ikke/gir ikke alltid riktig svar på grafer med negative vekter. Det gjør Bellman-Ford.

Versjoner

- Dijkstra

Tilhørende metoder:

Initialize-Single-Source

Standard å starte med på alle SSSP problemer. Den setter alle foreldre noder til **NIL** og alle lengder til **uendelig** så retter man det opp underveis i algoritmen. Algoritmen til startnoden blir satt til 0.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

After initialization, we have $v.\pi = \text{NIL}$ for all $v \in V$, $s.d = 0$, and $v.d = \infty$ for $v \in V - \{s\}$.

Relaxation

```
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
```

Bellman-Ford

Bellman-Ford er en algoritme for SSSP. Den tar for seg alle kantene i grafen i en rekkefølge også relaxerer den alle |V| - 1 ganger.

Video: 4.4 Bellman Ford Algorithm - Single Source Shortest Path - Dynamic Programming

Wikipedia: **https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm**

Boken: **https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=672**

Eks:

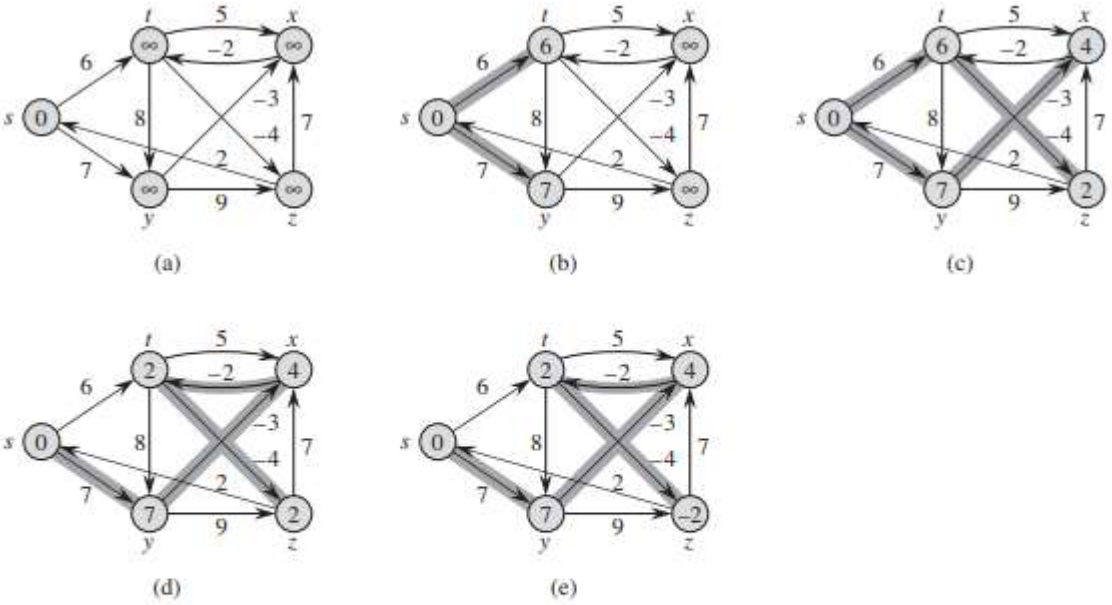


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

Analogi:

Datastruktur

Brukes på en rettet vektet graf. Grafen kan ha positive og negative kanter.

Egenskaper

-

Det generelle problemet den løser

Den løser den generelle versjonen av SSSP der kantvektene kan være negative.
Bellman-ford returnerer en boolsk verdi for om det finnes en negativ-sykel som kan nås fra startnoden **s.** Hvis det er tilfelle vil false returneres og det kan ikke dannes en løsning på SSSP. Ellers vil man få lengden fra **s** til hver av de andre nodene.

Spesifikke problemer den er best på:

Fungerer bedre enn Dijkstra når kantvektene kan være negative fordi den garanterer riktig resultat. Dijkstra kan ha rett og fungere raskere, men Dijkstra kan også gi feil resultat.

Kjøretid

O(VE)

O(n^3) for complete graph - Abdul bari

Sortert	**O(**	
Best-case	**O(|E|)**	
Average-case	**O(**	
Worst-case	**O(|V||E|)**	

The Bellman-Ford algorithm runs in time $O(VE)$, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges in lines 2–4 takes $\Theta(E)$ time, and the **for** loop of lines 5–7 takes $O(E)$ time.

Størrelse i minnet

Pseudokode

```
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
```

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=673

Styrker (+ i forhold til andre)

Fungerer på grafer med negative kantvekter i motsetning til Dijkstra.

Svakheter (+ i forhold til andre)

Hvis det kun er positive kantvekter, vil Dijkstra være raskere.

Den kan finne riktig svar på første rundt, men må gå igjennom |V| - 1 ganger uansett...

Versjoner

- Bellman-Ford

Relaxation

Link til relax: Relax

Initialize-single-source

Link til: Initialize-Single-Source

DAG-Shortest-Path

Hvis man relaxerer alle edges i topologisk order i en DAG, vil man få riktig SSSP i **O(|V|+|E|)** tid.

Video:

Wikipedia:

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=676

Eks:

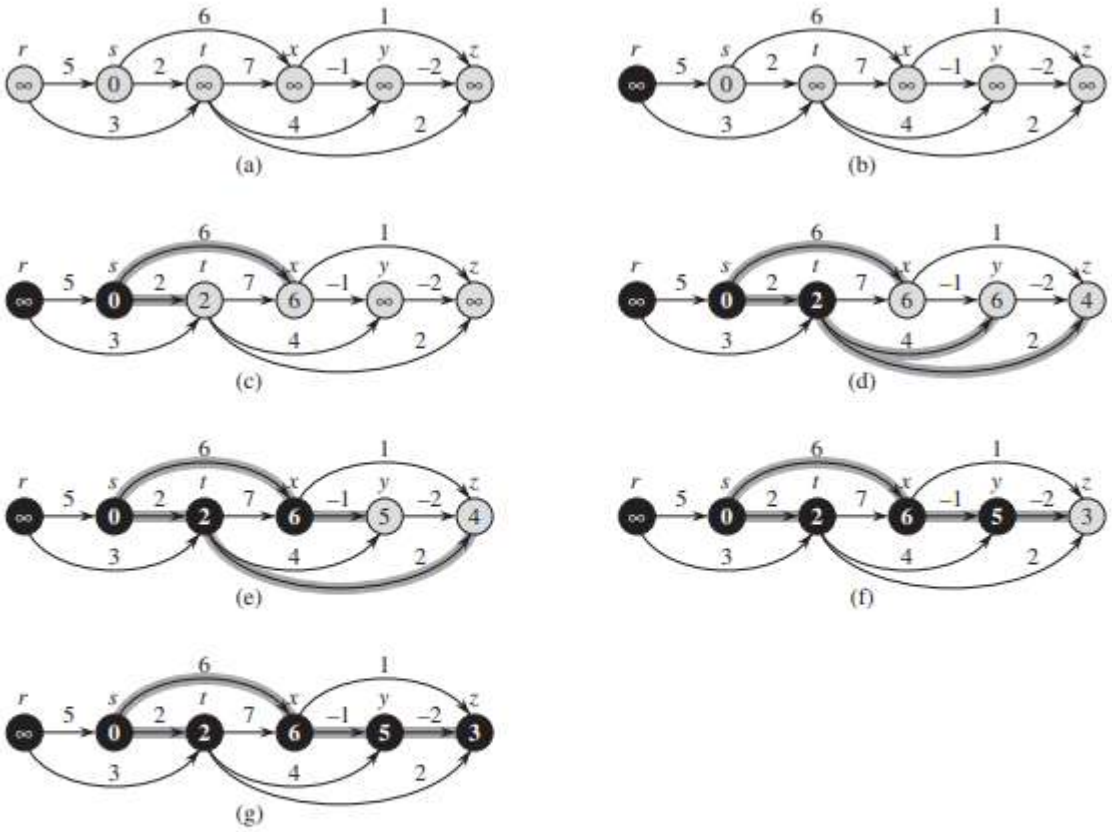


Figure 24.5 The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is s. The d values appear within the vertices, and shaded edges indicate the π values. **(a)** The situation before the first iteration of the **for** loop of lines 3–5. **(b)–(g)** The situation after each iteration of the **for** loop of lines 3–5. The newly blackened vertex in each iteration was used as u in that iteration. The values shown in part (g) are the final values.

Analogi:

Datastruktur

Brukes på en DAG for å få en SSSP.

Egenskaper

-

Det generelle problemet den løser

Løser SSSP problemet for en DAG.

Spesifikke problemer den er best på:

Den har veldig god kjøretid for en DAG. Lineær tid i forhold til en adjacency-list representasjon av grafen.

Kjøretid

O(|V|+|E|)

Sortert	**O(-**	
Best-case	**O(**	
Average-case	**O(-**	
Worst-case	**O(|V|+|E|)**	

The running time of this algorithm is easy to analyze. As shown in Section 22.4, the topological sort of line 1 takes $\Theta(V + E)$ time. The call of INITIALIZE-SINGLE-SOURCE in line 2 takes $\Theta(V)$ time. The **for** loop of lines 3–5 makes one iteration per vertex. Altogether, the **for** loop of lines 4–5 relaxes each edge exactly once. (We have used an aggregate analysis here.) Because each iteration of the inner **for** loop takes $\Theta(1)$ time, the total running time is $\Theta(V + E)$, which is linear in the size of an adjacency-list representation of the graph.

Størrelse i minnet

Pseudokode

```
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)
```

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Bedre tid (worst-case) enn Dijkstra og Bellman-Ford for en DAG. Funker også for negative kanter (noe Dijkstra ikke alltid gjør, men jeg tror Dijkstra gir riktig i en dag man starter samme sted som denne og med negative kanter, litt usikker da)

Svakheter (+ i forhold til andre)

Funker kun for DAGs, i motsetning til Bellman-Ford og Dijkstra.

Versjoner

- DAG-Shortest-Path

Floyd-Warshall

En algoritme for å løse APSP-problemet. Man starter opp med å sette alle distansene i utgangspunktet i en distansematrise D^(0) og predecessor matrisen PI^(0) vil bare få samme tall som raden den er i om det finnes en forelder. Så oppdaterer man iterasjonsvis om det går raskere å gå igjennom k (nummeret i iterasjonen) for å komme til target **j.**
Vanskelig å forklare, du skjønner det om du regner det ut.
For å finne en total sti til slutt må du gå baklengs fra target der PI til target er den forrige noden man kommer fra (siste noden man er i når man går til target j) også må man se på PI til den og jobbe seg bakover.

Video: https://www.youtube.com/watch?v=oNI0rf2P9gE

Wikipedia: https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=714

Eks:

Analogi:

Datastruktur
Brukes på en graf. Man får to matriser en distansematrise og en foreldrematrise.

Egenskaper

- Dynamic-programming ("Different dynamic-programming formulation")

Det generelle problemet den løser
Den løser APSP (All-pairs shortest path).

Spesifikke problemer den er best på:

Kjøretid
Theta(V^3)

Sortert	**O(**	
Best-case	**O(**	
Average-case	**Theta(V^3)**	
Worst-case	**Theta(V^3)**	

Størrelse i minnet

Pseudokode

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Versjoner

-

Transitive Closure

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=718

“Repeated squaring” (Ikke pensum?)

for APSP

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=707

Ford-Fulkerson

Video: **Ford-Fulkerson in 5 minutes — Step by step example**

Wikipedia:

Boken:

Eks:

Analogi:

Datastruktur

Egenskaper

-

Det generelle problemet den løser

Spesifikke problemer den er best på:

Kjøretid

Sortert	**O(**	
Best-case	**O(**	
Average-case	**O(**	
Worst-case	**O(**	

Størrelse i minnet

Pseudokode

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Versjoner

-

Edmond Karp

Video:

Wikipedia:

Boken:

Eks:

Analogi:

Datastruktur

Egenskaper

-

Det generelle problemet den løser

Spesifikke problemer den er best på:

Kjøretid

Sortert	**O(**	
Best-case	**O(**	
Average-case	**O(**	
Worst-case	**O(**	

Størrelse i minnet

Pseudokode

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Versjoner

-

Datastrukturer

En datastruktur er en struktur av data. :)

Comparison of list data structures

	Linked list	Array	Dynamic array	Balanced tree	Random access list	Hashed array tree
Indexing	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$[4]	$\Theta(1)$
Insert/delete at beginning	$\Theta(1)$	N/A	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(n)$
Insert/delete at end	$\Theta(1)$ when last element is known; $\Theta(n)$ when last element is unknown	N/A	$\Theta(1)$ amortized	$\Theta(\log n)$	N/A [4]	$\Theta(1)$ amortized
Insert/delete in middle	search time + $\Theta(1)$[5][6]	N/A	$\Theta(n)$	$\Theta(\log n)$	N/A [4]	$\Theta(n)$
Wasted space (average)	$\Theta(n)$	0	$\Theta(n)$[7]	$\Theta(n)$	$\Theta(n)$	$\Theta(\sqrt{n})$

	Linked list	Array	Dynamic array	Balanced tree	Random access list	Hashed array tree

Indexing	Θ(*n*)	Θ(1)	Θ(1)	Θ(log n)	Θ(log n)[4]	Θ(1)
Insert/delete at beginning	Θ(1)	N/A	Θ(*n*)	Θ(log n)	Θ(1)	Θ(*n*)
Insert/delete at end	Θ(1) when last element is known; Θ(*n*) when last element is unknown	N/A	Θ(1) amortized	Θ(log *n*)	N/A [4]	Θ(1) amortized
Insert/delete in middle	search time + Θ(1)[5][6]	N/A	Θ(*n*)	Θ(log *n*)	N/A [4]	Θ(*n*)
Wasted space (average)	Θ(*n*)	0	Θ(*n*)[7]	Θ(*n*)	Θ(*n*)	Θ(√*n*)

Mal

Navn

Video:

Wikipedia:

Boken:

Eks:

Analogi:

Egenskaper

-

Representasjon i minnet:

Brukes til:

Algoritmer:

-

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Metoder:

Dynamic set

Dynamic set kommer fra at vi har set med elementer i matematikk og det dynamiske er at de kan endres. De er finite.
https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=250
Hvert element i et dynamisk sett kan tenkes på som et objekt som kan ha ulike **attributter**. Hvis hvert element har et unikt **.key** element kan man tenke på settet som et sett av **keys**. I all praktisk sammenheng er disse **keysene** verdiene til objektet i hver datastruktur.
X er plassen til elementet.

Key

Verdien til et objekt i et set.

Satellite data/Sattellittdata

Data som ikke blir rørt av de implementerte metodene i det dynamiske settet.

Dictionary

Insert, **delete** og **testing av membership** (search) er metodene støttet av en dictionary.

Metoder

Metodene vi har på dynamiske set. Kan løses i **O(1)**, **O(lg(n))** eller **O(n)** tid. Rød-svarte trær kan løse alle metodene i **O(lg(n))** worst-case. Binære søketrær løser alt i **O(lg(n))** hvis treet er balansert.

Queries

Queries er metoder som returnerer informasjon om settet.

Modifying Operations

Modifying Operations er metoder som endrer på data i settet.

Search

SEARCH(S, k)
A query that, given a set S and a key value k, returns a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S.

Insert

INSERT(S, x)
A modifying operation that augments the set S with the element pointed to by x. We usually assume that any attributes in element x needed by the set implementation have already been initialized.

Delete

DELETE(S, x)
A modifying operation that, given a pointer x to an element in the set S, removes x from S. (Note that this operation takes a pointer to an element x, not a key value.)

Minimum

MINIMUM(S)
A query on a totally ordered set S that returns a pointer to the element of S with the smallest key.

Maximum

MAXIMUM(S)
A query on a totally ordered set S that returns a pointer to the element of S with the largest key.

Predecessor

PREDECESSOR(S, x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or NIL if x is the minimum element.

Kan også modifiseres så den vil gå gjennom alle elementene fra toppen hvis det er ikke-distinkte elementer.

Successor

SUCCESSOR(S, x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

Kan også modifiseres så den vil gå gjennom alle elementene fra bunnen hvis det er ikke-distinkte elementer.

Stack (Stakk)

Tenk bøtte/krukke

Tenk på en stack som en kortstokk som ligger på et bord. Legger du et element på toppen av bunken er det, det kortet du må løfte av først for å få tilgang til de andre.

En stakk har **LIFO** egenskapen, altså at det siste elementet du legger til blir det første du fjerner.

Insert metoden kalles ofte for **push** i en stack og **delete** metoden kalles **pop**. Dette kommer fra båndene med tallerkener i en kafeteria. Men dette er jo som en **array/list** i de fleste programmeringsspråk.

Toppen lagres i en variabel som peker til plassen til det siste elementet. **S.top = n**.

Når **S.top = 0** er altså stakken tom.

Video:

Wikipedia: **https://en.wikipedia.org/wiki/Stack_(abstract_data_type)**

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=253

Eks:

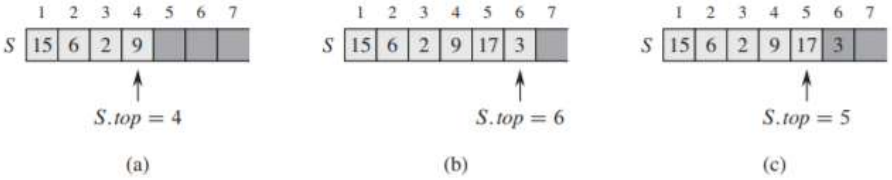


Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH(S, 17) and PUSH(S, 3). **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

Analogi: Den kjente analogien er å tenken på tallerkener som er stablet oppå hverandre.

Egenskaper

- Last-in-first-out (LIFO)

Representasjon i minnet:
Man lager en stakk som en array **S** med elementene **[1...n]**. Den tar altså opp akkuratt antall plasser som elementer. Så har den også en pointer til toppen. **S.top = n**.

Styrker (+ i forhold til andre)
Fjerner og legger til elementer i O(1) tid, men bare på sine spesifikke steder.

Svakheter (+ i forhold til andre)
Hvis man skal endre på noe i midten av stakken må man fjerne og legge til masse. Det kan bli veldig mange operasjoner man må gjøre hvis man skal kun endre en ting i bunnen for eksempel.

Metoder:

Stack-Empty

Returnerer **True** dersom stakken er tom. Dette er tilfelle hvis **S.top = 0**.

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

O(1)

Push (insert)

Legger til et nytt element (på toppen) av stacken.

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

O(1)

Pop (delete)

Fjerner det øverste elementet, dvs. siste elementet som ble lagt til i stakken.

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

Figure 10.1 shows the effects of the modifying operations PUSH and POP. Each of the three stack operations takes $O(1)$ time.

O(1)

Queue/Kø

En kø er et dynamisk sett som støtter alle de dynamiske operasjonene.
Tenk på det som en helt vanlig kø i en dagligvarebutikk
Den første personen som steller seg i køen er også den første personen som forlater køen selv om det har kommet mange andre i kø etter.
En kø har en **hale** og et **hode**. **Q.tail** og **Q.head**. Når man legger til noe kommer det på plassen til halen og halen økes med 1. Når man fjerner noen øker man plassen på hodet med 1, altså blir ikke den første personen en del av køen lengre.
Hvis **hode** = **hale** er køen tom. En kø starter med **Hode** = **Hale** = **1**.
Hvis man prøver å fjerne fra en kø som er tom vil man få **underflow**. Ved å legge til i en full kø vil man få **overflow**. Operasjonene som er beskrevet med pseudokode forhindrer at det skal skje.

Video:

Wikipedia: **https://en.wikipedia.org/wiki/Queue_(abstract_data_type)**

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=255

Eks:

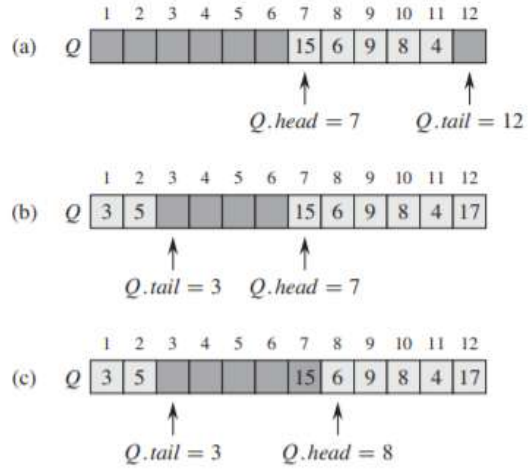


Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

Analogi: Som en kø irl.

Egenskaper

- First-in-first-out (FIFO)

Representasjon i minnet:

Representeres med en array. Størrelsen på arrayen er bestemt på forhånd. Man må vite ca. hvor mye plass man trenger på det meste. Man har også de to variablene **Q.head** og **Q.tail** som sier hvor køen starter og slutter.

Styrker (+ i forhold til andre)

Kan legge til og fjerne elementer i O(1) tid, men bare på sine spesifikke steder.

Svakheter (+ i forhold til andre)

Vanskelig å operere på elementer midt i/ på slutten av køen. Man er nødt til å gjøre veldig mange operasjoner for å endre på veldig lite.

Enqueue

Legger til et element bakerst i køen, på tail.

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

O(1)

Dequeue

Fjerner det første elementet i køen.

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

Figure 10.2 shows the effects of the ENQUEUE and DEQUEUE operations. Each operation takes $O(1)$ time.

O(1)

Linked lists/Lenkede lister

Lenkede lister er en datastruktur der hvert element har minst en peker til et annet element. Rekkefølgen i en lenket liste er lineær, men den er ikke bestemt av plassen i arrayen, men av hvem den blir pekt på og hvem den peker videre på. I en enkel lenket liste har hvert element en peker framover i listen. I en dobbeltlenket liste har hvert element to pekere - en framover og en bakover.

I tillegg til at hver lenket liste inneholder et **key**/verdi objekt og minst en peker, kan de inneholde satellittdata i tillegg.

Lenkede lister kan implementere alle de dynamiske set metodene, selv om det ikke alltid er veldig effektivt.

Boken går mest i detalj på dobbelt lenkede lister.

Andre lenkede lister er:

Multippel lenket liste
Dette er en lenket liste der hvert element har minst to pekere. En dobbel lenket liste er en form for multippel lenket liste. Dette kan brukes til å sortere en mengde elementer på flere kriterier som alder, sted, verdi osv.
Circular linked list
Dette er en lenket liste der siste element peker til det første elementet igjen.

Sentinel node

En ekstra node i starten og/eller slutten uten data for å sikre at alle elementer peker på noe og at vi ikke får noe out of bounds problemer.

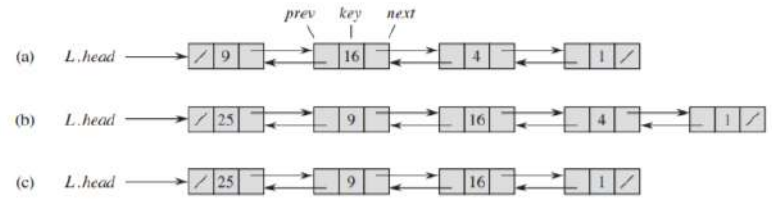


Figure 10.3 (a) A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. (b) Following the execution of LIST-INSERT(L, x), where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. (c) The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

Enkel lenket liste

I en enkel lenket liste har hvert element en **key**/verdi og en peker til neste objekt. Vi har også en **L.head**. Hvis listen er sortert er er den minste **key**/verdien i hodet og den største er i halen.

Video:

Wikipedia: https://en.wikipedia.org/wiki/Linked_list

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=257

Eks:

Analogi:

Egenskaper

- Lineær aksess

Representasjon i minnet:

Måten vi har lært å implementere lenkede lister er med arrayer. Kan enten gjøres i en array eller i flere der en array er for **keysene** og de andre er for pekerne. Hvis det gjøres i en pleier man å ha et key element i plass **k, next** i **k+1**.

Brukes til:
Kan brukes til å implementere andre datastrukturer.

Algoritmer:

Styrker (+ i forhold til andre)
Fordelen over en vanlig array er at ingen elementer bytter plass i minnet hvis de flyttes, legges til eller fjernes. Fordi det er abstrakte pekere kan et element som flyttes fra slutten til starten bare bytte "abstrakt" plass ved å endre på noen av pekerne. I en array måtte en masse elementer flytte plass i minne for å gjøre lignende ting.

Svakheter (+ i forhold til andre)
Det er vanskelig å aksessere elementer midt i listen, man må gå sekvensielt gjennom hele listen til man ankommer elementet. I Random Access kan man direkte aksessere elementer midt i en array.

Metoder:

List-search

Returnerer plassen/elementet på **keyen** det søkes etter.

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3       x = x.next
4  return x
```

To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

O(n)

List-insert

List-delete

O(n) tid utenom å finne element x som også tar **O(n)** tid med List-Search. Grunnen er fordi man må iterere gjennom listen til man finner forrige objekt for å sette riktig next på den.

Dobbel lenket liste

Hvert element i en dobbelt lenket liste har to pekere, en til neste element og en til forrige. I tillegg har listen en hale og et hode. Hvis et element har **x.prev = NIL** så vil det si at det er det første elementet, hodet. Hvis **x.next = NIL** er det det siste elementet altså halen. Hvis **L.head = NIL** så er listen tom.

Video:

Wikipedia: https://en.wikipedia.org/wiki/Linked_list

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=257

Eks:

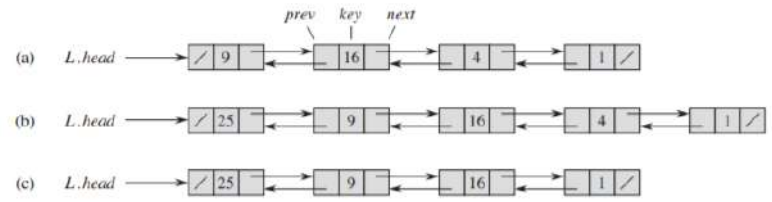


Figure 10.3 (a) A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. (b) Following the execution of LIST-INSERT(L, x), where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. (c) The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

Analogi:

Egenskaper

-

Representasjon i minnet:
Måten vi har lært å implementere lenkede lister er med arrayer. Kan enten gjøres i en array eller i flere der en array er for **keysene** og de andre er for pekerne. Hvis det gjøres i en pleier man å ha et key element i plass **k, next** i **k+1** og **prev** i **k+2**.

Brukes til:
Å holde på data eller til å implementere andre datastrukturer.

Algoritmer:

Styrker (+ i forhold til andre)
Kan slette i **O(1)** tid i motsetning til enkel lenket liste.
Fordelen over en vanlig array er at ingen elementer bytter plass i minnet hvis de flyttes, legges til eller fjernes. Fordi det er abstrakte pekere kan et element som flyttes fra slutten til starten bare bytte “abstrakt” plass ved å endre på noen av pekerne. I en array måtte en masse elementer flytte plass i minne for å gjøre lignende ting.

Svakheter (+ i forhold til andre)
Det er vanskelig å aksessere elementer midt i listen, man må gå sekvensielt gjennom hele listen til man ankommer elementet. I Random Access kan man direkte aksessere elementer midt i en array.

Metoder:
Metodene med ‘ er på lister med sentinel objekter.

List-search

Returnerer plassen/elementet på **keyen** det søkes etter.

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3       x = x.next
4  return x
```

To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

O(n)

```
LIST-SEARCH'(L, k)
1  x = L.nil.next
2  while x ≠ L.nil and x.key ≠ k
3       x = x.next
4  return x
```

List-insert

Putter inn et element x på plassen til head og flytter alle andre elementer sådan en plass ned i listen, men bare ved å endre pekeren til den forrige headen.

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3       L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

(Recall that our attribute notation can cascade, so that $L.head.prev$ denotes the $prev$ attribute of the object that $L.head$ points to.) The running time for LIST-INSERT on a list of n elements is $O(1)$.

O(1)

```
LIST-INSERT'(L, x)
1  x.next = L.nil.next
2  L.nil.next.prev = x
3  L.nil.next = x
4  x.prev = L.nil
```

List-Delete

Fjerner et element fra listen ved at det forrige og neste objektet peker på hverandre istedenfor å peke på elementet.

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```

Gjøres i **O(1)** tid men man må søke etter elementet i **O(n)** tid først.
Den kan forenkles ved at man legger til en sentinel element. I boken gjør de det men en sirkulær liste og legger det sentinelle elementet i mellom head og tail.
Da blir list-delete:

```
LIST-DELETE'(L, x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

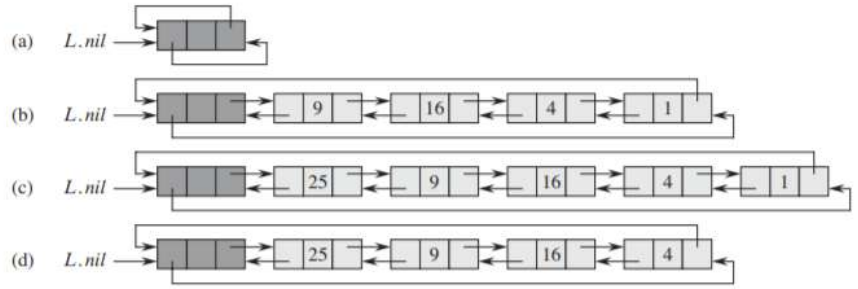


Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $L.nil$ appears between the head and tail. The attribute $L.head$ is no longer needed, since we can access the head of the list by $L.nil.next$. **(a)** An empty list. **(b)** The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT$'(L, x)$, where $x.key = 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4.

Hashtabeller

Hashtabeller er en dictionary datastruktur.
I worst-case tar det **O(n)** å slå opp et element i en hashtabell, men i snitt tar det **O(1).** En hashtabell er basert på sannsynlighet.
Istedenfor å ha dårlig utnyttelse av plass som en direkte aksess tabell har, kan en hashtabell ha O(|K|) i plass der K er keysene. Den faktiske plassen til en hashtabell

Simple uniform hashing

Simple uniform hashing vil si at det er lik sjanse for en hashverdi å havne i hver slot uavhengig av hva som har vært tidligere.

Uniform hashing

Vi kan bruke en randomized hash function for å garantere god average case og unngå at en adversarial med malicious intents gjør så hash tabellen får O(n) kjøretid.

Hash funksjon

Funksjonen for å regne på slotten/plassen en key vil få.
h(k) = plassen til k (**Hash value**)

Direkte aksess tabell

En direkte aksess tabell har keyen til et element på plassen til keyen, men det er veldig upraktisk for store tall og hvis man har kun en liten mengde tall. Det kan være umulig på flere maskiner hvis en av keysene kan være for store. Storage O(|U|) der U er Universet av nøkler.

Collision

Når to nøkler **hashes** into the same slot/place. We solve it by chaining.

Chaining

When multiple keys **hashes** to the same slot, we create a linked list and add the new element. Den b

Perfect hashing

Med perfekt hashing oppnår vi O(1) i worst-case på de ulike metodene.

Video:

Wikipedia: https://en.wikipedia.org/wiki/Hash_table

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=274

Eks:

Analogi:

Egenskaper

-

Representasjon i minnet:
I en array med direkte aksess. Man må bruke en hash funksjon for å finne plassene i arrayen som keysene har.

Brukes til:
Dictionairy. Veldig god oppslagstid i snitt.

Algoritmer:

Styrker (+ i forhold til andre)
Veldig god oppslagstid. Fungerer bedre enn andre datastrukturer som dictionairy.

Svakheter (+ i forhold til andre)
Har ikke støtte for alle dynamiske datastruktur operasjoner.

Metoder:

h(k) = slot

Ved kollisjon:

CHAINED-HASH-INSERT(T, x)
1 insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)
1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)
1 delete x from the list $T[h(x.key)]$

Priority queue/Prioritetskø

Max-priority queue og min-priority queue. Implementeres med max-heap og min-heap. Elementet med høyest prioritet (enten lavest eller høyest **key**/verdi avhengig av type kø) skal hentes ut først. Det gjøres jo enkelt ved en heap.

Video: https://youtu.be/HqPJF2L5h9U?t=2817

Wikipedia: https://en.wikipedia.org/wiki/Priority_queue

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=183

Eks:

Analogi:

Egenskaper

-

Representasjon i minnet:

Brukes til:

- **Planlegging av jobber (max)**
- **Event-driven simulator (min)**

Algoritmer:

-

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Metoder:

- Insert
- Maximum/Minimum
- Extract-Max/min
- Increase/decrease-key

Graphs/Grafer

Node

Element i en graf.

Kant

Veien mellom to noder

Rettet/Urettet

Rettede kanter kan kun traverseres/gå en vei. Man kan ha flere rettede kanter mellom to noder. Urettede kanter kan traverseres begge veier.

Vektet

En vekt er en tilhørende verdi til en kant og kan for eksempel representere kostnaden det tar å traversere den veien/avstanden mellom nodene.

Vertex cover

Hvis vi velger en delmengde av alle vertices så vil de fortsatt grense til alle kanter i grafen.

DAG (Directed-acyclic-graph)

DAG er en graf med rettede kanter og ingen sykler. Dette kan sjekkes av definisjonen at et **DFS** ikke gir en bakoverkant (dvs. ingen sykler).

Lemma 22.11 A directed graph **G** is acyclic if and only if a depth-first search of **G** yields no back edges.

Proof =>: Suppose that a depth-first search produces a back edge **(u,v)**. Then vertex **v** is an ancestor of vertex **u** in the depth-first forest. Thus, **G** contains a path from **v** to **u**, and the back edge **(u,v)** completes a cycle.
<=: Suppose that **G** contains a cycle **c**. We show that a depth-first search of **G** yields a back edge. Let **v** be the first vertex to be discovered in **c**, and let **(u,v)** be the preceding edge in **c**. At time **v.d**, the vertices of **c** form a path of white vertices from **v** to **u**. By the white-path theorem, vertex **u** becomes a descendant of **v** in the depth-first forest. Therefore, **(u,v)** is a back edge.

Residual-graph

Predecessor subgraph

Transitive Closure Graph

G*

Tree/Tre[6]

Et tre er en graf som ikke inneholder sykler. DVS. at det er kun 1 **sti** mellom hver node i treet.

Node

Et element i et tre kalles en node

Leaf/Bladnode/Perifær node?/løvnoder?

En bladnode er en node uten barn. De vil være helt nederst i sin del av treet.

Internal node/Intern node

En node som har barn. Dvs. at den er "inni" treet, ikke helt nederst.

Forest/Skog

Hvis en eller flere noder ikke henger sammen med de andre kalles det en skog.

Free/Fritt

Et fritt tre har ingen dybde. Motsatt av rooted/rotfestet.

Rooted/Rotfestet

Der noder har dybder og **roten** er øverst med dybde 0. Da vil hver annen node inneholde et **subtre** som er rotfestet i den noden.

Parent/Forelder

En node **B** er forelder til en annen node **A** hvis **B** er på nøyaktig ett høyere nivå enn **A** og de har kobling til hverandre. Det motsatte blir da at **A** er barnet til foreldrenoden **B**.

Child/Barn

En node **A** er barn til en annen node **B** hvis **A** er på nøyaktig ett nivå dypere enn **B** og de har en kobling til hverandre. Det motsatte blir da at **B** er foreldrenoden til **A**.

---

[6] https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=1194

Neighbor/nabo

Foreldre og barn er naboer.

Ancestor/Forfader/ascendent

Alle foreldre og foreldre av foreldre til noden hele veien opp til rotnoden. **Inkludert seg selv**.

Proper ancestor

Samme som ancestor, men uten inkludert seg selv

Descendant/Etterkommer?

Alle barne noder og barn av barne nodene. **Inkludert seg selv**. Dvs at hele treet er descendant til rotnoden.

Proper descendant

Samme som descendant, men uten inkludert seg selv.

Sibling/søsken

Nodene på samme nivå/dybde som har samme foreldernode.

Subtre

Et tre som er rotfestet i en annen node en roten i det originale treet.

Depth/Level/Nivå/Dybde

Dybden til en node er det samme som hvor mange “proper ancestors” den har. 0 i den øverste, 1 i barna dens osv.

Height/Høyde

Høyden til et tre er hvor mange edges den har oppover, som er det samme som dybden til en leaf/blad på laveste nivå. Så en tre med 1 forelder og 1 barn har høyde 1.

Representasjon i minnet

Man kan representere trær på mange forskjellige måter. For eksempel med lenkede lister. For binære søketrær kan man da ha en p.left og p.right som er hvert av deltrærne(barna). Hvis et tre ikke er binært kan man peke til leftChild og rightSibling.
https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=267

DFS-forest/DFS-skog

Shortest paths tree

Huffman tree

Binary tree/Binært tre[7]

En node kan maks ha 2 barn. Man deles opp i left og right der hvert barn igjen er et binært tre. Hver node vil inneholde et binært subtre som er rotfestet i den noden.
Viktige formler på binære trær:
h er høyde
n er antall noder
i er antall interne noder
l er antall blader

Antall noder i et fullt binærtre er MINST	**(2*h) + 1 (h er høyde)**
Antall noder i et fullt binærtre er MAKS	**2^(h+1) - 1**
Antall blader i et perfekt tre er	**(n+1)/2**
Antall noder i et perfekt tre er	**2*l - 1**
Antall interne noder i et NESTEN KOMPLETT binærtre er (alle binærtrær?)	**floor(n/2)**
Antall blader i et binærtre er	**i + 1**

Binære trær blir ofte representert i arrayer. Her er nyttige formler for å regne på posisjoner i arrayen der n er posisjonen på current node:

Left child/Venstre barn	**2*n**
Right child/Høyre barn	**2*n + 1**
Parent/forelder	**Floor(n/2)**

[7] "Binary tree - Wikipedia." https://en.wikipedia.org/wiki/Binary_tree. Åpnet 6 nov.. 2020.

Full/Fullt

Alle noder har 2 barn eller 0 barn.

Complete/Komplett[8]

I vårt tilfelle er det et perfekt binært tre.
Dvs. at alle interne noder har 2 barn og alle løvnoder er på samme nivå. Trekantformet.

Nearly complete

Alle unntatt siste nivå er perfekte, og de siste nodene er til venstre slik at en array som hadde representert den hadde ikke hatt mellomrom.

Mange ville kalt det for complete, og det vi kaller complete for en perfekt.

Perfekt

Alle løvnoder på samme nivå og alle foreldre har to barn. Trekantformet.

Balansert

Alle bladnoder er på "nesten" samme nivå. Kan for eksempel være 1 forskjell, men man kan også definere det med mer enn 1 forskjell i nivå/dybde.

[8] https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=172

Binary search tree

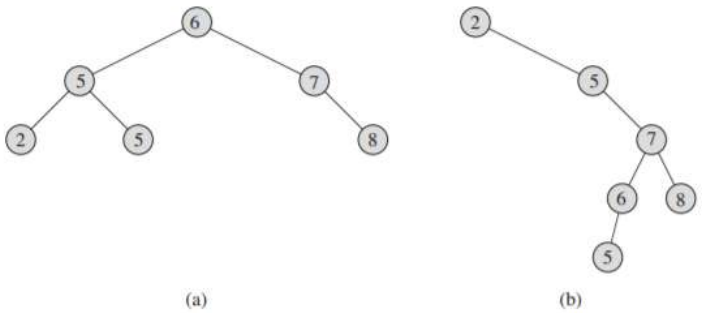


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

I et binært søketre er nodene i venstre subtre til **v** maks verdien av **v** og verdiene i høyre subtre minst verdien av **v**.
Dette gjør at vi får en sortert array om vi hadde lest av tallene i rekkefølgen venstre til høyre uavhengig av høyde om treet var perfekt stilt opp.

Et binært søketre støtter alle de dynamiske operasjonene. Snitt tiden er **O(lg(n))** hvis søketreet er balansert, men worst-case er **O(n)**.

Vi får ut en sortert rekkefølge ved **Inorder tree-walk!**
(de to andre er **Preorder tree walk** og **postorder tree walk)**

Binary-search-tree property:

The keys in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree property***:

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

Video:

Wikipedia:

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=307

Eks:

Analogi:

Egenskaper

-

Representasjon i minnet:

Brukes til:

Algoritmer:

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Metoder:

Inorder tree walk

En metode for å skrive av alle nodene i et tre i en bestemt rekkefølge. Bra å bruke på binary search tree fordi man får ut en sortert rekkefølge på tallene.

Hvis man tenker at alle noder har tre sider (venstre, under og høyre) vil dette være at du skriver ned andre gang du møter noden.

Preorder tree walk er første gang du møter noden og **postorder tree walk** er siste gang du møter noden.

Video:

Wikipedia:

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=309

Eks:

Analogi: Tenk at du går forbi alle nodene i et tre i motsatt vei av klokken. Når du står under en node (sett ovenfra) skriver du den ned.

Hvis man tenker at alle noder har tre sider (venstre, under og høyre) vil dette være at du skriver ned andre gang du møter noden.

Datastruktur

Brukes på **trær**. Både binærtrær og andre typer.

Egenskaper

- Rekursiv

Det generelle problemet den løser

Skriver ut alle nodene i et tre i en bestemt rekkefølge. (når man "står" under noden)

Spesifikke problemer den er best på:

Veldig god å bruke på binært søketre fordi du får ut en sortert rekkefølge av tallene.

Kjøretid

Sortert	**O(**	
Best-case	**O(**	
Average-case	**O(**	
Worst-case	**O(**	

Størrelse i minnet

Pseudokode

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)
Bedre enn de to andre tre traversingene/graf traverseringene for trær på et binært søketre fordi du får ut den sorterte rekkefølgen på tallene.

Svakheter (+ i forhold til andre)

Versjoner

- Inorder tree walk
- Preorder tree walk (da skriver man av første gang man møter)
- Postorder tree walk (da skriver man av siste gang man møter)

Tree-Search

Iterative-Tree-Search

Tree-Minimum

Tree-Maximum

Tree-Successor

Tree-Predecessor

Tree-Insert

Transplant

Tree-Delete

Heap/Haug

Det finnes to typer binære hauger[9]: max-heap og min-heap. De har egenskapen at alle forgjengere er enten større eller mindre enn etterkommerne. Hvis man rotfester et nytt sted vil man få et subtre som også er samme type heap.
Bør være nesten komplett binærtre. (usikker for vårt pensum)
Brukes til å lage prioritetskøer.
Arrayen som representerer heapen har attributene **A.length** og **A.heap-size.** A.length er lengden av hele arrayen (kan innholde slettede elementer) og A.heap-size er størrelsen på heapen, det er alle elementene som er med i binærtreet.
A.height er antall kanter til den dypeste bladnoden.

Operation	find-max	delete-max	insert	increase-key	meld
Binary[8]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[8][9]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[b]	$\Theta(\log n)$	$O(\log n)$[c]
Fibonacci[8][10]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Pairing[11]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$o(\log n)$[b][d]	$\Theta(1)$
Brodal[14][e]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[16]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Strict Fibonacci[17]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2–3 heap[18]	$O(\log n)$	$O(\log n)$[b]	$O(\log n)$[b]	$\Theta(1)$	?

Left child/Venstre barn	**2*n**
Right child/Høyre barn	**2*n + 1**
Parent/forelder	**Floor(n/2)**

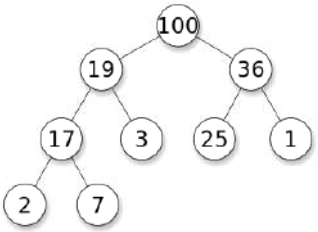
Video: https://www.youtube.com/watch?v=HqPJF2L5h9U

Wikipedia: https://en.wikipedia.org/wiki/Heap_(data_structure)

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=172

Eks:

[9] https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=173



En max-heap.

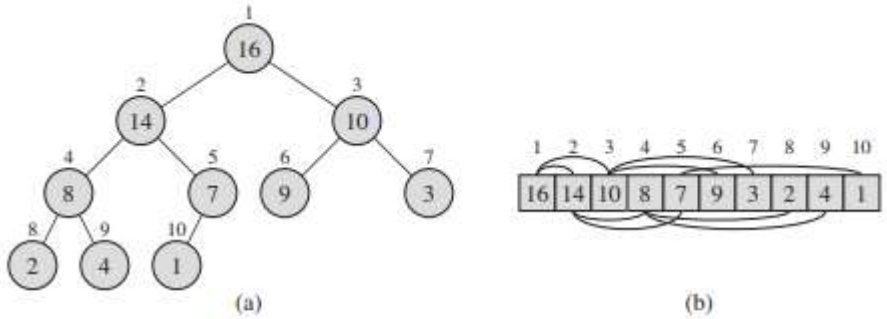


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships;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

En annen maxheap + arrayen dens.

Enhver sortert array representert som en heap er en type min-heap.

Analogi:

Egenskaper

- Heap-egenskapen
- Nesten komplett binærtre (usikker for vårt pensum)

Representasjon i minnet:

Representeres i en array. Som binærtre form.

Brukes til:

- **Prioritetskøer (Min-heap og max)**
- **Heapsort (Max-Heap)**

Algoritmer:

- Heapsort

Styrker (+ i forhold til andre)
Lettere å lage enn et binært søketre og har veldig mange metoder vi kan modifisere den med.

Svakheter (+ i forhold til andre)
Et binært søketre har raskere funksjoner for visse ting. Hvilke?

Metoder:

Parent

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=173

```
PARENT(i)
1  return ⌊i/2⌋
```

O(1)

Left

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=173

```
LEFT(i)
1  return 2i
```

O(1)

Right

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=173

```
RIGHT(i)
1  return 2i + 1
```

O(1)

Max/Min-Heapify

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

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=175

Build-Max/Min-Heap

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

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=177

Heapsort

```
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)
```

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=180

Max/Min-Heap-Insert

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

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=185

Heap-Extract-Max/Min

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

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=184

Heap-Increase/Decrease-Key

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

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=185

Heap-Maximum/Minimum

```
HEAP-MAXIMUM(A)
1  return A[1]
```

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=184

Max-heap

Rotelementet er det største elementet. Alle barn er mindre enn sine foreldre og ancestors. For hvert sub-re er roten det største elementet.

Video: https://www.youtube.com/watch?v=HqPJF2L5h9U

Wikipedia: https://en.wikipedia.org/wiki/Heap_(data_structure)

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=172

Eks:

Analogi:

Egenskaper

- Max-heap egenskapen

Representasjon i minnet:
Som en array. Beskrevet i heap. Representasjon ellers er som et binært nesten komplett tre.

Brukes til:

- Heapsort
- Noen prioritetskøer

Algoritmer:

- Heapsort

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Metoder:

- **Parent**
- **Left**
- **Right**
- **Max-Heapify**
- **Build-max-heap**
- **Heapsort**
- **Max-heap-insert**
- **Heap-extract-max**
- **Heap-increase-key**
- **Heap-maximum**

Insert

Man legger til et element på slutten av arrayen. så “Upheaper” man den til den når en akseptabel plass.

Datastruktur

Den brukes på **heaps.**

Egenskaper

-

Det generelle problemet den løser

Legger til et element i en heap.

Spesifikke problemer den er best på:

Den er god til å legge til et element i en datastruktur. (Gjøres i god kompleksitet)

Kjøretid

Sortert	**O(?)**	
Best-case	**O(1)**	Hvis man legger til et tall som er strengt mindre enn alle andre så havner den på sin egen plass i starten av algoritmen. Dette vil også skje så lenge den er mindre enn forelderen som er på plass floor(n/2) av plassen den settes på.
Average-case	**O(log(n))**	
Worst-case	**O(log(n))**	Den må gjøre log(n) sammenligninger som kommer fra høyden til treet helt til den når toppen, aka den er størst.

Størrelse i minnet

In-place.

Pseudokode

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Må brukes på en heap. Hvis heapen ikke er sortert fra før kan det hende den ikke vil bli sortert etter heller.

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)
Plasseres på riktig plass i best-case O(1) og worst O(lg(n)) som er bedre enn noen typer datastrukturer.

Svakheter (+ i forhold til andre)
Plasseres ikke alltid riktig i O(1) tid, noe en hashtabell hadde gjort i average-case i hvertfall.

Versjoner

- Max-Heap-Insert
- Min-Heap-Insert

Delete

Man fjerner det øverste elementet. Tenk: epler som er stilt fram i en butikk i pyramideform. Man må ta eplet på toppen, det fineste eplet.
Putter så det siste elementet på toppen og så "nedheaper" det til riktig plass ved å sammenligne barna og så bytte med det største barnet om det er større.

Eks:

Datastruktur
Den brukes på **heaps/hauger.**

Egenskaper

-

Det generelle problemet den løser
Fjerner det øverste elementet i heapen og så sorterer opp et nytt element.

Spesifikke problemer den er best på:
Den er god på å fjerne et spesifikt element (det øverste). Spesielt når vi har en prioritetskø.

Kjøretid

Sortert	**O(log(n))**	
Best-case	**O(-)**	Den må uansett sortere opp de neste elementene./sortere ned den øverste
Average-case	**O(log(n))**	
Worst-case	**O(log(n))**	

Størrelse i minnet
In place. Kan ende opp med å store det fjernede elementet i den siste plassen i arrayen og bare definere at heapen stopper før. Da får man en liste i order etter der heapen stopper for hver gang man deleter elementer.

Pseudokode

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)
Fjerner et element i **O(lg(n))** tid som er bedre enn for en enkeltlenket liste (O(n)). Og hvis man tar med søketiden til dobbeltlenket liste er den også bedre.

Svakheter (+ i forhold til andre)
Verre tid enn for dobbeltlenket liste O(1), men dobbeltlenket liste må søke så totalt bedre som oftest.

Versjoner

-

Heapify (max-heapify)

Heapify(A,s). Sortering som i delete. Man antar at noden s er større enn barna og "nedheaper" (boken sier "float down") den til den når riktig plass ved å sammenlikne hvem av barna som er størst og om den er mindre enn det største barnet bytter den plass og det gjør den om og om igjen til den havner ned på riktig plass.

Video:

Wikipedia:

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=175

Eks:

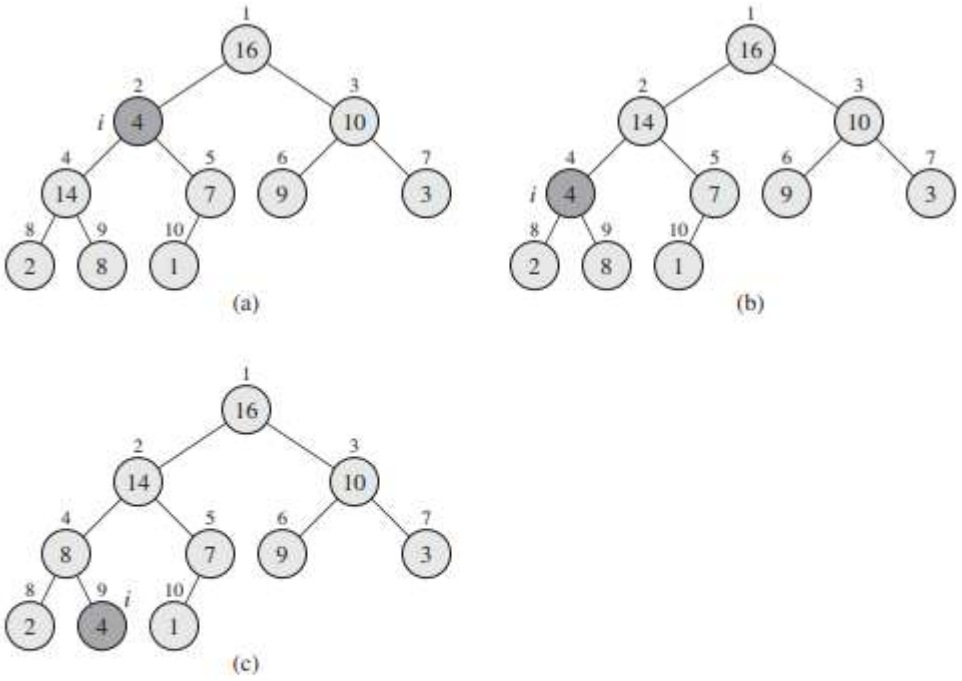


Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

Analogi:

Datastruktur

Den brukes på **heaps**

Egenskaper

- Rekursiv
- Inplace

Det generelle problemet den løser

Får et element som er på feil plass (for høyt) ned på riktig plass. Den kan være på riktig plass, i såfall blir det bare en sjekk?
For å opprettholde heap-egenskapen.

Spesifikke problemer den er best på:

Kjøretid
h er høyden der s er

Sortert	**O(**	
Best-case	**O(1)?**	hvis den er på riktig sted?
Average-case	**O(lg(n))** **O(h)**	
Worst-case	**O(lg(n))** **O(h)**	

Størrelse i minnet

In-place

Pseudokode

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

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Versjoner

- Max-Heapify
- Min-Heapify

Heapsort

Ved å delete alle elementene for så å putte de på den tomme plassen som oppstår, ender man opp med å sette de største elementene bakers i synkende rekkefølge bakover. AKA stigende rekkefølge forfra.

Video:

Wikipedia:

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=180

Eks:

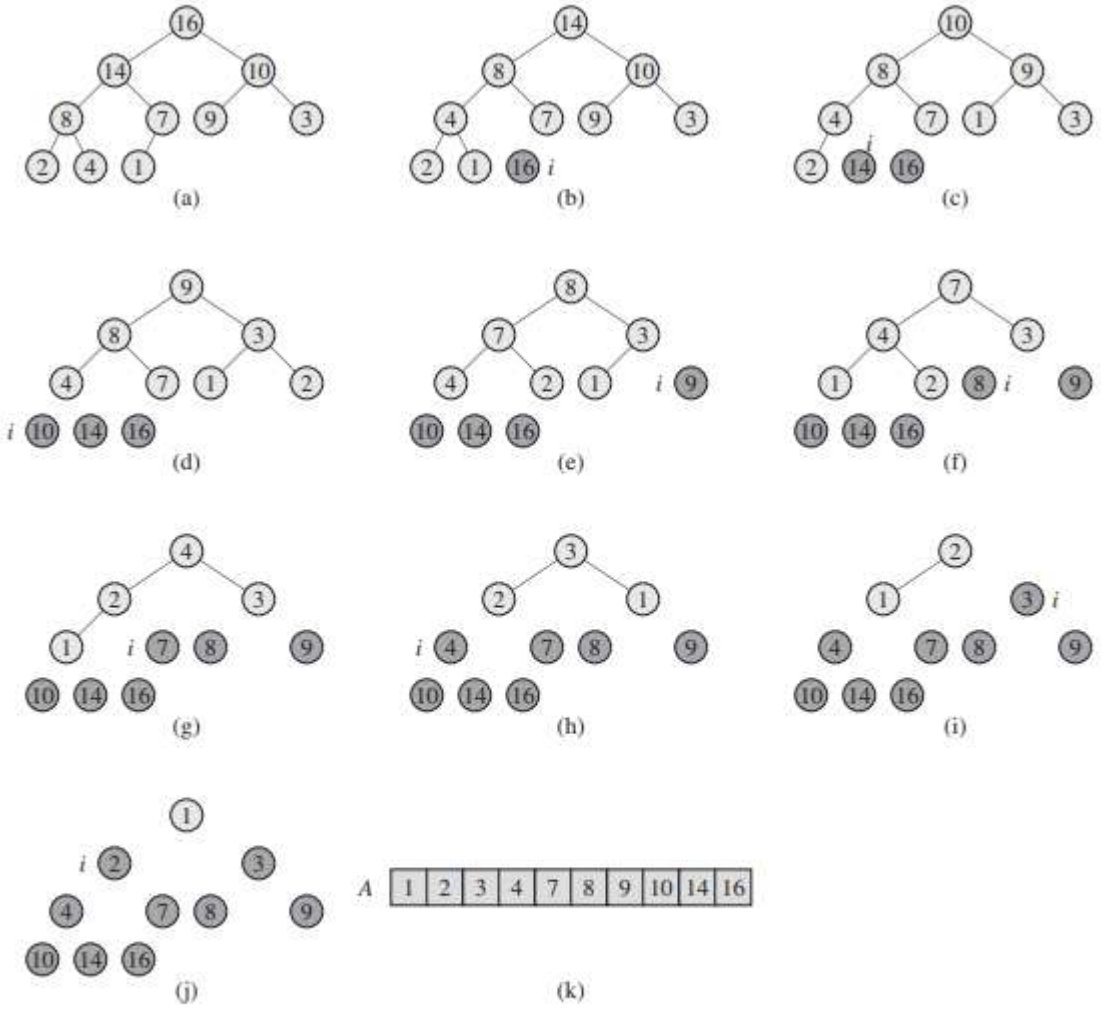


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

Analogi:

Datastruktur
Brukes på en usortert array. Starter med å gjøre den om til en heap.

Egenskaper
- Inplace

Det generelle problemet den løser
Sortering

Spesifikke problemer den er best på:
Veldig god på sortering ved at den har optimal worst-case for sammenlikningsbasert sortering og er inplace.

Kjøretid
O(n) + O(n-1)*O(lg(n))

Sortert	**O(n*lg(n))**	
Best-case	**O(n*lg(n))**	
Average-case	**O(n*lg(n))**	
Worst-case	**O(n*lg(n))**	

Størrelse i minnet
Inplace O(1) auxiliary

Pseudokode

```
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)
```

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)
Den er in-place i motsetning til merge-sort. Bedre worst-case enn insertion-sort. Bedre worst-case enn quicksort.

Svakheter (+ i forhold til andre)
Større skjulte konstanter enn quick-sort i average case.
Litt usikker, men tror noen av de skjulte konstantene er store hvis man må starte med å lage en heap.

Versjoner

- Heapsort

Build-Max-Heap

Lager en heap fra en array. Brukes i starten av heapsort om man ikke har en heap allerede.

Video:

Wikipedia:

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=177

Eks:

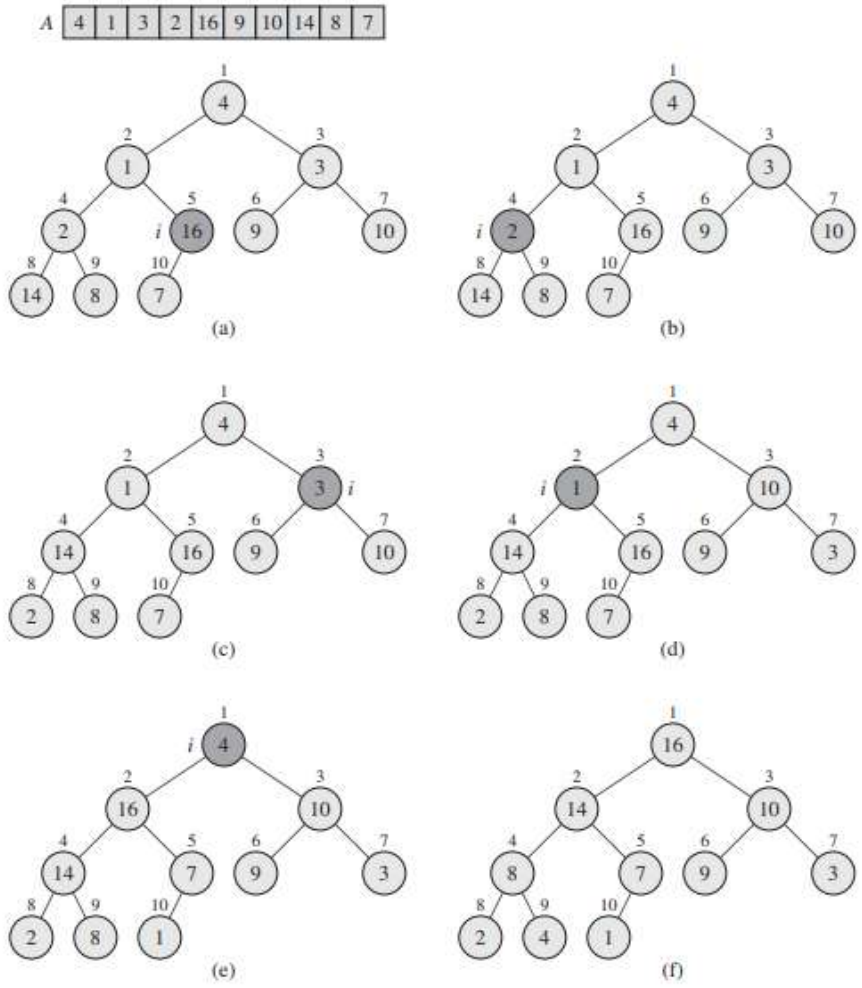


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

Analogi:

Datastruktur
Brukes på en array for å sette den opp i heap-struktur med heap-egenskapen

Egenskaper
- Inplace

Det generelle problemet den løser
Lager en heap av en vanlig array

Spesifikke problemer den er best på:

Kjøretid

Sortert	**O(n)**	
Best-case	**O(n)**	
Average-case	**O(n)**	
Worst-case	**O(n)**	

Størrelse i minnet
In-place O(1) auxiliary. O(n) for arrayen

Pseudokode

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

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Versjoner

- Build-Max-Heap
- Build-Min-Heap

Heap-Increase-Key

Man øker **keyen**/verdien til et element/node og plasserer den på riktig plass ved å "upheape"/"float up".

Video:

Wikipedia:

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=185

Eks:

Analogi:

Datastruktur

Brukes på heap. Increase-key brukes på max-heap.

Egenskaper

- In-place

Det generelle problemet den løser

Øker verdien av en node og plasserer den på riktig plass i heapen om den da er på feil.

Spesifikke problemer den er best på:

Kjøretid

Sortert	**O(**	
Best-case	**O(**	
Average-case	**O(**	
Worst-case	**O(lg(n))**	

Størrelse i minnet

Pseudokode

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

Forklaring på hvordan den fungerer, trinn for trinn

Krav for at den skal fungere (evt. tilleggskrav)

Korrekthetsbevis, hvordan og hvorfor den virker

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Versjoner

- Heap-Increase-Key
- Heap-Decrease-Key (Min-heap)

Min-heap

Rotelementet er det minste elementet. Alle barn er større enn sine foreldre. For hvert sub-tre er roten det minste elementet.

Video: https://www.youtube.com/watch?v=HqPJF2L5h9U

Wikipedia: Heap (data structure)

Boken: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=172

Eks:

Analogi:

Egenskaper

- Min-heap egenskapen

Representasjon i minnet:

Som en array. Beskrevet i heap. Representasjon ellers er som et binært nesten komplett tre.

Brukes til:

- Priorityqueue

Algoritmer:

- Heapsort (motsatt vei?)

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Metoder:

- **Parent**
- **Left**
- **Right**
- **Min-Heapify**
- **Build-min-heap**
- **Heapsort (blir motsatt sortert?)**
- **Min-heap-insert**
- **Heap-extract-min**
- **Heap-decrease-key**
- **Heap-minimum**

Fibonacci-heap

En fibonacci heap har gode amortiserte kjøretider på metodene sine.

bonacci heaps. Chapter 19 shows that if a Fibonacci heap holds $|V|$ elements, an EXTRACT-MIN operation takes $O(\lg V)$ amortized time and a DECREASE-KEY operation (to implement line 11) takes $O(1)$ amortized time. Therefore, if we use a

Navn

Video:

Wikipedia:

Boken:

Eks:

Analogi:

Egenskaper

-

Representasjon i minnet:

Brukes til:

Algoritmer:

Styrker (+ i forhold til andre)

Svakheter (+ i forhold til andre)

Metoder:

Nabomatriser

Shortest-Path-Matrix

Predecessor-Matrix

Nabolister

Konsepter

Her tar vi for oss deler av algoritmer og datastruktur pensum/tema som ikke er algoritmer, problemer eller datastrukturer.

Asymptotisk notasjon

Splitt-og-hersk/Divide-and-conquer

Splitt-og-hersk går ut på å gjøre ting **rekursivt** og på hvert nivå:

- **Divide:** Dele opp i enda mindre subproblemer
- **Conquer**: Løse delproblemene rekursivt. Hvis de er små nok, så skal de bare løses rett fram.
- **Combine:** Slå sammen delløsningene oppover til den totale løsningen.

Hvis det er subproblemer som ikke er som hovedproblemet, men litt annerledes. Så løses de som en del av combine steget.
Det er viktig at delproblemene er disjunkte.

Recursive case

Når et subproblem igjen kan deles i subproblemer rekursivt.

Base case

Når subproblemet ikke kan løses rekursivt. Altså at man er på det dypeste i rekurrensen.

Algoritmer

- Merge-sort
- Quicksort

Problemer som kan løses med divide-and-conquer

- Sortering
- Maximum subarray
- Matrisemultiplikasjon (Strassen matrix multiplication)

Rekurrens

En rekurrens er en ulikhet eller likning for å beskrive verdien av en funksjon i form av mindre versjoner av seg selv.

Tre metoder for å løse rekurrenser: **substitusjon**, **rekurrenstrær** og **masterteoremet**

Substitusjon

Man gjetter grensen til rekurrensen og bruker matematisk induksjon for å regne ut om den stemmer.

Rekurrenstrær

Et tre der nodene representerer cost på nivået.

Master method/Masterteoremet

Gir grenser for rekurrenser på formen

$$T(n) = aT(n/b) + f(n)\,,$$

where $a \geq 1$, $b > 1$, and $f(n)$ is a given function.

Må huske tre ulike situasjoner

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Iterasjonsmetoden

En metode for å finne kjøretiden til en algoritme.
Man får oppgitt en kjøretid som T(n) = T(n-1) + x og T(0) = 0 for eksempel.
Bevises med induksjon.

Variabelskifte

Dynamisk programmering

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=380
Dynamisk programmering minner om splitt-og-hersk ved at man rekursivt deler opp problemer i mindre problemer, men her må problemene overlappe.
Vi har altså **overlappende delproblemer** som vil si at subploblemene deler subproblemer.
Kan bare brukes når problemet har **optimal substruktur/delstruktur**.

Dynamisk programmering løser bare et delproblem en gang og lagrer det i en tabell og henter ut svaret når man støter på problemet igjen.

Dynamisk programmering brukes typisk i **optimaliseringsproblemer**.

Når man skal lage en dynamisk programmerings algoritme er det typisk å følge 4 steg:

1. Karakterisere en optimal løsning
2. Rekursivt definere verdien på en optimal løsning
3. Komputere verdien på en optimal løsning, typisk nedenfra og opp
4. Konstruere/sette sammen en optimal løsning fra den komputerte dataen

Vi kan droppe steg 4 hvis vi kun trenger den optimale verdien, ikke selve løsningen. For å få den optimale løsningen må vi av og til legge til tilleggsinformasjon i steg 3.

“Steps 1–3 form the basis of a dynamic-programming solution to a problem. If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4. When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.”
https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=380

Overlappende delproblemer

Vi har altså **overlappende delproblemer** som vil si at subploblemene deler subproblemer.
https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=405

Optimal substruktur/optimal delstruktur

Et problem har optimal delstruktur hvis en optimal løsning inneholder optimale løsninger til delstrukturer.

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=400
Boken om å finne optimal delstruktur:

1. You show that a solution to the problem consists of making a choice, such as choosing an initial cut in a rod or choosing an index at which to split the matrix chain. Making this choice leaves one or more subproblems to be solved.
2. You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.
3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
4. You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a "cut-and-paste" technique. You do so by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction. In particular, by "cutting out" the nonoptimal solution to each subproblem and "pasting in" the optimal one, you show that you can get a better solution to the original problem, thus contradicting your supposition that you already had an optimal solution. If an optimal

solution gives rise to more than one subproblem, they are typically so similar that you can modify the cut-and-paste argument for one to apply to the others with little effort.

Memoisering/memoization

Memoisering er at vi lagrer verdien på en løsning av et delproblem, denne kan vi senere bruke hvis vi støter på problemet igjen.

Time-memory trade-off

Vi kan oppnå bedre kjøretid ved å bruke mer minne. Dette skjer ved memoisering, der man lagrer tidligere løste problemer så man slipper å regne de ut igjen, men det tar mer memory plass.
Kan endre problemer fra eksponensiell tid til polynomial tid. Hvis hvert distinkt delproblem er polynomisk i input og kan løses i polynomisk tid kan dette bli tilfelle.

Iteration

Top-down/ovenfra og ned

Løses fortsatt rekursivt. Fra toppen: Vi ser om vi har løst "delproblemet" før og returnerer det hvis vi har det, hvis ikke løser den problemet som vanlig. -> når man ikke har løst det øverste nivået vil man havne et nivå ned, men man har ikke løst det heller så man havner videre ett nivå ned osv. til man når bunnen også sendes resultatene oppover igjen.

Lik asymptotisk kjøretid som bottom-up, bortsett fra i spesielle tilfeller der top-down er tregere fordi den gjør samme delproblemer flere ganger.
Top-down har ofte verre konstante faktorer fordi den har mye mer overhead.

Bottom-up/nedenfra og opp

Når man vet at hvert problem kan løses fra mindre deler av problemet. Og man vet de relevante størrelsene på delproblemene. Starter med å sette opp alle delproblemene i stigende rekkefølge og løser de i orden. Da kan man bruke alle de tidligere resultatene i de kommende problemene.

Lik asymtotisk kjøretid som top-down, bortsett fra visse tilfeller der bottom-up har bedre kjøretid fordi den faktisk bare gjør delproblemene en gang.
Bottom-up har bedre konstante faktorer fordi den har mye mindre overhead.

Sub-problem graph/delproblem graf

En delproblemgraf viser hvordan delproblemer hører til/"avhenger" høyere problemer.
"**reversed topological sort**" ingen delproblemer kalles før "barna" er løst.
Vanligvis er tiden det tar å konstruere svaret på et delproblem proporsjonal til antall kanter den har ut.
https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=388

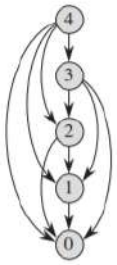


Figure 15.4 The subproblem graph for the rod-cutting problem with $n = 4$. The vertex labels give the sizes of the corresponding subproblems. A directed edge (x, y) indicates that we need a solution to subproblem y when solving subproblem x. This graph is a reduced version of the tree of Figure 15.3, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

Algoritmer

Problemer

- Knapsack

- Stavkutting

Greedy/Grådige algoritmer

Optimaliseringsalgoritmer vil typisk gå igjennom steg med mange valg per steg. Dynamisk programmering kan være “overkill”. Grådige algoritmer vil gjøre det som ser ut som det beste steget ved hvert steg.
Håp om at lokale optimale valg vil lede til et globalt optimalt valg.

Det er ikke garantert at det blir riktig, men ved flere problemer vil det føre til optimale løsninger.

Greedy-choice property/Grådighetsegenskapen

At hvis vi gjør grådige valg vil vi allikevel ende opp med en optimal løsning.

Huffman

Amortisert analyse

Amortisert analyse er en gjennomsnittsanalyse av hver operasjon i worst-case. Det handler ikke om statistisk gjennomsnittskjøretid til en algoritme, men gjennomsnittet av alle operasjoner. Da vil en lang operasjon gjevnes ut av mange små.

Innledning: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=472
Datastrukturutvidelse: https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=484

Load factor/Lastfaktor

I en ikketom datastruktur T, er lastfaktoren **alfa** antall elementer delt på antall plasser.
Eksempel:

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2}, \\ 1 & \text{otherwise}. \end{cases}$$

The total cost of n TABLE-INSERT operations is therefore

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n, \end{aligned}$$

Amortisert kostnad per operasjon er 3

Det brukes 3 teknikker men ikke særlig del av pensum:

Aggregate analysis

Accounting method

Potential method

Traversering av grafer

Vet ikke hvilken del av pensum som krever dette, men man kan bruke SSSP algoritmer e.l.
Dijkstra, Bellman-Ford og DAG-Shortest path

Hvis man skal traversere trær har man 3 kjente traverseringer:

Preorder tree walk

Inorder tree walk

Postorder tree walk

Grafrepresentasjon

Flyt

Flyt omhandler forflytting av tall i en graf. Man kan intuitivt tenke på det som for eksempel vann, av navnet flyt, som skal bevege seg i gjennom rør(grafen).
https://en.wikipedia.org/wiki/Flow_network
https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=729

Relaterte algoritmer:

- Ford-Fulkerson
- Edmond Karp

Relaterte datastrukturer:

- Flytnettverk/Flytgraf

Relaterte problemer:

- Maksimum flyt

Problemer

Mal

Video

Wikipedia

Boka

Egenskaper

-

Eksempel

Analogi

Versjoner

-

Løsning

Algoritmer som løser problemet

-

Maximum subarray

Man skal finne den største sammenhengende arrayen i en array. Altså arrayen med størst sum. Det er kun interessant å finne når det er både positive og negative verdier. Med positive verdier ville det bare vært hele arrayen.

Kan løses ved å sammenlikne alle par av “sluttverdier”/”totalverdier”, men den løses bedre med divide-and-conquer.

Video

Wikipedia

Boka

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=89

Egenskaper

-

Eksempel

Analogi

Versjoner

Løsning

Algoritmer som løser problemet

Rod-Cutting/Stavkutting

Hvis vi har en stav som vi kan dele opp i ulike størrelser og vi har forskjellige verdier for en stav av størrelse l så får vi et optimalsiering problem for hvilken kombinasjon av oppkuttinger som gir størst verdi for en stor stav med størrelse L.

Antall kombinasjoner av oppkutting vi kan gjøre på en stav hvis hver lengde vi kan ha er en "integral"/heltallsstørrelse er **2^(n-1).** Der **n** er lengden på hele staven i et helt tall.
Dette kan vi intuitivt tenke oss ved å se på staven som en binær streng der hvert kutt er en bit. Vi kan for eksempel da tenke oss at 1 er hvis vi kutter og 0 er hvis vi ikke kutter. Det vil være 1 mindre bit en lengden av hele staven.

Hvis verdien av en hel stav er stor nok, kan løsningen være å ikke kutte noe. Og hvis verdien av den minste stavstørrelsen er delvis større eller lik alle andre størrelser vil det være en optimal løsning å kutte staven maks antall ganger.

Stavkuttingproblemet inneholder **optimal delstruktur.** Som vil si at om man regner ut løsningen for en delstruktur også skalerer det opp vil løsningen forbi optimal hvis den er satt sammen av delstrukturer.

Egenskaper

-

Video

Wikipedia

Boka

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=381

Eksempel

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Figure 15.1 A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.

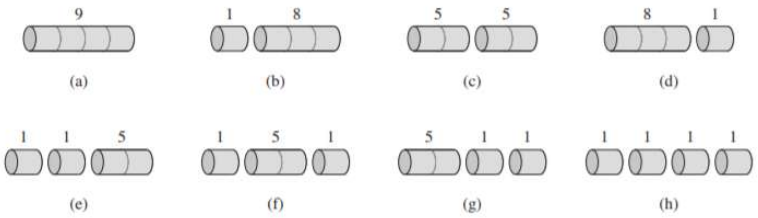


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is

$$
\begin{aligned}
r_1 &= 1 \quad \text{from solution } 1 = 1 \quad \text{(no cuts)}, \\
r_2 &= 5 \quad \text{from solution } 2 = 2 \quad \text{(no cuts)}, \\
r_3 &= 8 \quad \text{from solution } 3 = 3 \quad \text{(no cuts)}, \\
r_4 &= 10 \quad \text{from solution } 4 = 2 + 2, \\
r_5 &= 13 \quad \text{from solution } 5 = 2 + 3, \\
r_6 &= 17 \quad \text{from solution } 6 = 6 \quad \text{(no cuts)}, \\
r_7 &= 18 \quad \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3, \\
r_8 &= 22 \quad \text{from solution } 8 = 2 + 6, \\
r_9 &= 25 \quad \text{from solution } 9 = 3 + 6, \\
r_{10} &= 30 \quad \text{from solution } 10 = 10 \quad \text{(no cuts)}.
\end{aligned}
$$

Analogi

Versjoner

- Det finnes en versjon der man ikke kan ha synkende størrelser i stavkuttene, men det utgjør minimal forskjell, det blir bare noen færre muligheter enn **2^(n-1)**.

Løsning

Generell løsning:

More generally, we can frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) . \tag{15.1}$$

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) .$$

Likning nr 2 tar bare hensyn til 1 av 2 oppdelinger som gir samme resultat.

Algoritmer som løser problemet

- Bruk dynamisk programmering

Top-Down recursive:

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

Procedure CUT-ROD takes as input an array $p[1..n]$ of prices and an integer n, and it returns the maximum revenue possible for a rod of length n. If $n = 0$, no revenue is possible, and so CUT-ROD returns 0 in line 2. Line 3 initializes the maximum revenue q to $-\infty$, so that the **for** loop in lines 4–5 correctly computes $q = \max_{1 \le i \le n}(p_i + \text{CUT-ROD}(p, n-i))$; line 6 then returns this value. A simple induction on n proves that this answer is equal to the desired answer r_n, using equation (15.2).

IKKE BRA

2^n tid

Med memoizering top-down:

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

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

Bottom-up:

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

Theta(n^2) kjøretid for begge memoriserte løsninger.
Men begge disse gir kun verdien på den beste løsningen, ikke hva som inngår i den.

Denne gir også lengdene:

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] and s[0..n] be new arrays
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          if q < p[i] + r[j - i]
7              q = p[i] + r[j - i]
8              s[j] = i
9      r[j] = q
10 return r and s
```

LCS (Longest Common Subsequence)

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=411

Optimal binary search tree

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=418

Knapsack

0-1 KsP

0/1 Knapsack problemet går ut på at man har n objekter med n_w vekt og n_p verdi. Så har man en plass m. Dette er et optimaliseringsproblem der man ønsker samlet så høy verdi som mulig uten å overstige vekten m.
Video https://www.youtube.com/watch?v=nLmhmB6NzcM

Wikipedia https://en.wikipedia.org/wiki/Knapsack_problem

Boka https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=446

Egenskaper

- Optimal substruktur
- NPC/NPH

Eksempel

Analogi
Tenk at en tyv skal i en butikk og har en **m** stor sekk og det er **n** objekter å stjele, hvilke bør han stjele for å få mest verdi?

Versjoner

- 0/1 knapsack
- fractional knapsack (man kan dele opp objektene)

Løsning

Algoritmer som løser problemet
Bruk dynamisk programmering

Fractional KsP

0/1 Knapsack problemet går ut på at man har n objekter med n_w vekt og n_p verdi. Så har man en plass m. Dette er et optimaliseringsproblem der man ønsker samlet så høy verdi som mulig uten å overstige vekten m.

Video

Wikipedia https://en.wikipedia.org/wiki/Continuous_knapsack_problem

Boka https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=447

Egenskaper

- Optimal substruktur

Eksempel

Analogi

Som i 0/1:

Tenk at en tyv skal i en butikk og har en **m** stor sekk og det er **n** objekter å stjele men han kan dele opp objektene (det er støv), hva burde han ta for å maksimere verdien?

Versjoner

- Fractional knapsack
- 0/1 knapsack (man kan ikke dele opp objektene)

Løsning

Algoritmer som løser problemet

-

Minimum Spanning Trees (MST)

Problemet er at man skal lage et tre av en vektet graf der summen av vektene i treet er lavest mulig. Treet vil ende opp med |V|-1 kanter.

Video

Wikipedia

https://en.wikipedia.org/wiki/Minimum_spanning_tree

Boka

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=645

Egenskaper

-

Eksempel

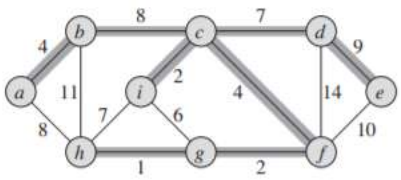


Figure 23.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

Analogi

Man skal kun ha en sti fra hver node til en annen og den samlede vekten til treet skal være så lav som mulig.

Versjoner

Løsning

Algoritmer som løser problemet

- Kruskal's algorithm
- Prim

SSSP (Single-source shortest path) (en-til-mange)

I SSSP skal vi finne korteste veien fra en node til alle andre noder i en vektet rette graf. Man kan både ha kun positive vekter eller både positive vekter og negative.

Video

Wikipedia

https://en.wikipedia.org/wiki/Shortest_path_problem#Single-source_shortest_paths

Boka

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=664

Egenskaper

-

Triangle inequality (Lemma 24.10)
For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property (Lemma 24.11)
We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 24.12)
If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.

Convergence property (Lemma 24.14)
If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

Path-relaxation property (Lemma 24.15)
If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Predecessor-subgraph property (Lemma 24.17)
Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Eksempel

Analogi

Versjoner

- med/uten negative vekter
- med/uten sykler

Løsning

Algoritmer som løser problemet

- Dijkstra (kun positive vekter)
- Bellman-Ford (positive og negative vekter)

APSP (All-pairs shortest path)

Vi ønsker å finne den korteste veien fra en node til alle andre for alle noder.

Video

Wikipedia

https://en.wikipedia.org/wiki/Shortest_path_problem#All-pairs_shortest_paths

Boka

https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=705

Egenskaper

-

Eksempel

Analogi

Versjoner

-

Løsning

Vi kan bruke en SSSP algoritme |V| ganger, for eks Dijkstra hvis alle vekter er positive. Hvis vi bruker Dijkstra med en linear-array versjon av en min-priority queue blir kjøretiden O(V^3+VE) = O(V^3). Med binary min-heap som priority-køen blir kjøretiden O(VE*lg(V)) som er bedre hvis det er få kanter ("sparse"). Med en fibonacci-heap blir kjøretiden O(V^2*lg(V)+VE).

Hvis det er negative kanter kan vi bruke Bellman-Ford som gir O(V^2E) som i en tett ("dense") graf blir O(V^4).

Dette er ikke veldig gode kjøretider, derfor kan vi bruke andre egnede algoritmer som Floyd-Warshall og Johnson.

APSP med SSSP algoritmer:

Dijkstra linear-array min-priority queue	O(V^3)
Dijkstra binary min-heap priority queue	O(VE*lg(V))
Dijkstra fibonacci-heap	O(V^2*lg(V)+VE)
Bellman-Ford	O(V^2E) som er O(V^4) i en tett graf

Løsningen pleier å bli skrevet i en distanse-matrise hvor d_ij er distansen under enhver iterasjon og ved terminering er det den faktiske korteste distansen. Man har også en **predecessor matrix**.

Algoritmer som løser problemet

- Floyd-Warshall
- Johnson

SDSP (Single-Destination Shortest-Paths) (Ikke pensum?)

SPSP (Single-pair Shortest-Path) (Ikke pensum?)

Maximum flow

SAT

Circut-SAT

SAT

CNF SAT

2-CNF SAT

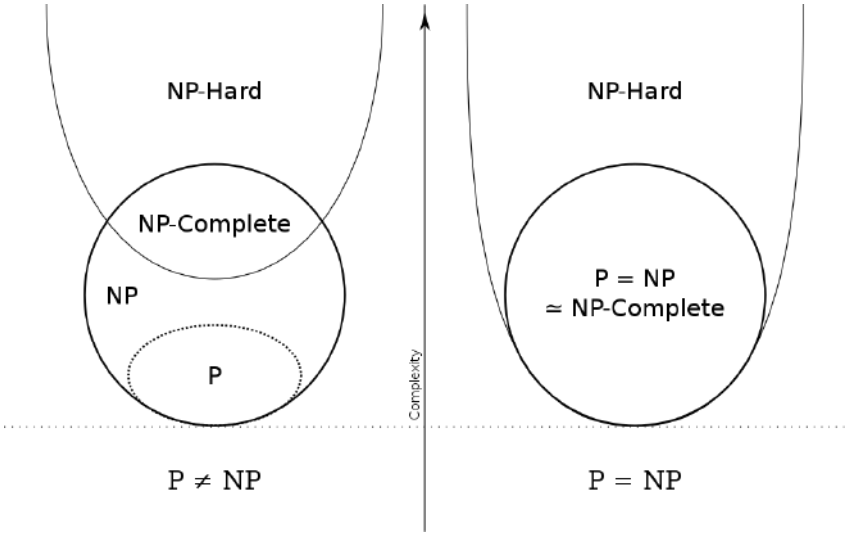
3-CNF SAT

Clique problem

Graph coloring

Sum of subsets

NP/NP-complete/NP-hard



I computer science complexity har vi forskjellige grupper der P, NP, NPC og NPH er spesielt interessante (og pensum). De omhandler **beslutningsproblem**, og NPH kan inneholde **optimaliseringsproblemer** også.
Disse mengdene er **ikke exhaustive/uttømmende**. Altså omfatter de ikke alle problemer.

Boken om hvordan vi kommer oss unna beslutningsproblemproblematikken:
Although NP-complete problems are confined to the realm of **decision problems**, we can take advantage of a convenient relationship between **optimization problems** and **decision problems**. We usually can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized. For example, a decision problem related to SHORTEST-PATH is PATH: given a directed graph G, vertices u and v, and an integer k, does a path exist from u to v consisting of at most k edges?

Decision problem/Beslutningsproblem

Et problem som kan besvares med ja eller nei.

Optimaliseringsproblem

Et problem man ønsker å finne et minimum eller maksimum på og den tilhørende løsningen. (Ikke ja/nei)

NP

NP står for Non-deterministic-polynomial og er en gruppe med problemer vi kan bekrefte en løsning på i polynomisk tid, men vi kan ikke nødvendigvis løse de i polynomisk tid.
Non-deterministic delen står for at vi kan skrive en **ikke-deterministisk algoritme** på

problemet som vil si at det er deler av algoritmen vi ikke kan forklare/kode i detalj. NP inneholder P og NPC der NPC er snittet av NP og NPH. Det er uvisst om P=NP (et av de største problemene i CS 2000-problemene). ALT i NP er **beslutningsspørsmål**.
NP gir ja i polynomisk tid.

CoNP

Nesten som NP, CoNP er gruppen med problemer som gir nei i polynomisk tid.

P

P er en delmengde av (eller hele) NP. Beslutningsproblemer vi kan både løse og bekrefte i polynomisk tid.

P-problemer:

- Finne ut om en boolsk 2-CNF-formel er oppfyllbar
- Finne ut om en graf har en sti mellom to noder av lengde <= k

NPC (NP-Komplett)

En delmengde av NPH og NP (snittet av dem). NPC problemer kan enda ikke løses i polynomisk tid, men det er ikke bevist at de ikke kan. Siden de er en del av NP kan vi bekrefte løsningene på problemene i polynomisk tid. NPC problemer er beslutningsproblemer.

NPC-problemer:

- Finne ut om en graf har en hamilton-sykel
- Finne ut om en boolsk 3-CNF-formel er oppfyllbar
- Finne ut om en digital krets kan gi 1 som output
- Finne ut om en graf har en sti mellom to noder av lengde >= k

NPH (NP-Hardness)

En mengde med problemer som vi enda ikke kan løse i polynomisk tid. Det er også forventet at ikke alle kan løses i polynomisk tid. (tar pensum med Halting?) NPH inneholder både **beslutningsproblemer** og **optimaliseringsproblemer**. NPC er i NPH. NPH er minst like vanskelig som NPC og verre.

NPH-problemer:

- Finne ut om en graf har en hamilton-sykel
- Finne ut om en boolsk 3-CNF-formel er oppfyllbar
- Finne ut om en digital krets kan gi 1 som output
- Finne ut om en graf har en sti mellom to noder av lengde >= k

- Finne den største mulige klikken i en graf (clique)

Reduksjon

Reduksjon handler om å gjøre om en instans av et problem til et annet type problem som gjerne kan løses raskere. For **polynomisk-tid reduksjonsalgoritme** kreves det at:

- Transformasjonen tar polynomisk tid
- Svarene er de samme. Svaret i reduksjonen er 'ja' hvis og bare hvis svare er 'ja' i den reduksjonerte

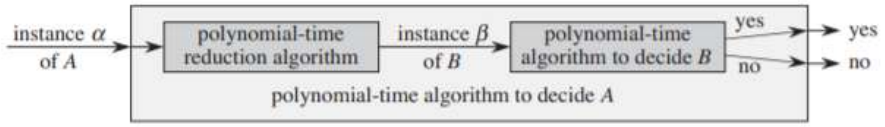


Figure 34.1 How to use a polynomial-time reduction algorithm to solve a decision problem A in polynomial time, given a polynomial-time decision algorithm for another problem B. In polynomial time, we transform an instance α of A into an instance β of B, we solve B in polynomial time, and we use the answer for β as the answer for α.

Video https://youtu.be/e2cF8a5aAhE

Wikipedia **NP (complexity)**
NP-hardness

Boka **https://sjarit.no/tjenester/indok-data/boker/algdat.pdf#page=1069**

Eksempel

Analogi

Versjoner

Løsning
NP=P vet vi ikke enda. Dette er et åpent spørsmål i computer-science. 2000-talls questions.