Oppsummering algoritmer, trær og grafer.

(alle logaritmer er $\log_2()$ hvis ikke annet er spesifisert)

Binærtrær

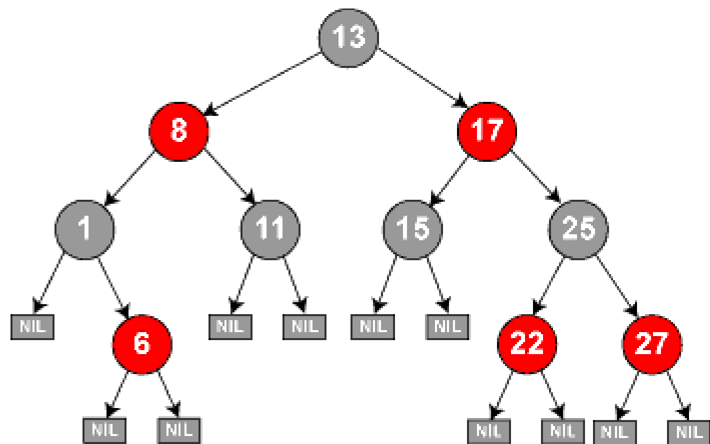
Like store på høyre og venstre side (perfekt balansert).

Høyde = log(n)

Red-Black-trees

Fra wikipedia.org

1. A node is either red or black.
2. The root is black.
3. All leaves are black.
4. Both children of every red node are black.
5. The paths from each leaf up to the root contain the same number of black nodes.



Demonstrasjon:

www.ece.uc.edu/~franco/C321/html/RedBlack/**redblack**.html

Rekurrenser og Rekurrenstrær

Kan sees på som hvordan en rekurrens utføres. Eventuelt hva rekurrensen gjør.

T(n) = T(n/2) + O(n) vil utføre en eller annen operasjon som koster O(n) hver gang den går.

Dette kan sees på som et perfekt balansert binærtre siden den «deler» operasjonen i n/2 hver gang. Dermed vil den ha en høyde på log(n). Kostnaden for hvert nivå i treet koster O(n). Dermed blir den sammelagte kjøretiden «n ganger log(n)» asymptotisk. Som medfører O(nlog(n)). Merk «stor-O» siden kjøretiden kan være lik nlog(n) altså ikke alltid mindre enn nlog(n), som ville vært «lille-o». Akkurat for denne rekurrensen vil kjøretiden også være Θ(nlog(n)) siden den er både øvre og nedre begrenset av nlog(n).

Løsning av rekurrenser

De tre forskjellige trivielle måtene å løse rekurrenser på er:

Induksjon

- Løse manuelt for å «se» et mønster slik at man kan «gjette» på en løsning

Substitusjon

- Bytte ut inputargumentet til noe bedre. For så å løse, og bytte tilbake igjen.

Masterteoremet

- Kan kun brukes på rekurrenser på formen:

  T(n) = aT(n/b) + f(n)

Induksjon

Gjetningen her kommer fra et erfarent øye. For mer dødelige, løs «for hånd» et par tre-fire-fem ganger og «se» mønsteret..

$$T(n) = 4\,T\left(\frac{n}{2}\right) + n^2, \qquad T(1) = 1 \tag{1}$$

Gjetningen

$$T(n) = O(n^2 \lg n) \tag{2}$$

skal bevises ut fra (1) ved induksjon på n. Da kan vi benytte (2) som induksjonshypotese for lavere argumenter til T enn n:

$$T(n) = 4\,T\left(\frac{n}{2}\right) + n^2 =$$

Her bruker vi induksjonshypotesen (2) på $T\left(\frac{n}{2}\right)$:

$$= 4\,O\left(\left(\frac{n}{2}\right)^2 \lg\frac{n}{2}\right) + n^2 = 4\,O\left(\frac{1}{4}\,n^2(\lg n - 1)\right) + n^2 =$$
$$4\,O\left(n^2 \lg n\right) + n^2 = O\left(n^2 \lg n\right)$$

Dermed er (2) bevist. Samme argument kan føres for Θ og Ω.

Substitusjon

Vi bytter ut inputargumentet til noe som vi vet blir mer «kjent». Siden vi vet at sqrt(n) = $n^{1/2}$ så er det lurt å sette

m=log(n)

siden det er $\log_2$ så er:

$n = 2^m$

Innsatt i den opprinnelige rekurrensen blir den greiere å løse. Når vi har løst med den innsatte er det bare å sette tilbake uttrykket for m.

$$T(n) = T(\sqrt{n}) + 1, \qquad T(1) = 1$$

Her kan vi sette $m = \lg n$, altså $n = 2^m$:

$$T(2^m) = T(\sqrt{2^m}) + 1 = T(2^{m/2}) + 1$$

Hvis vi nå setter $S(m) = T(2^m)$, får vi:

$$S(m) = S\left(\frac{m}{2}\right) + 1$$

som har løsningen

$$S(m) = \Theta(\lg m)$$

altså

$$T(2^m) = \Theta(\lg m)$$

eller

$$T(n) = \Theta(\lg \lg n)$$

Trikset her er å «se» hva vi skal substituere for.. Ingen god regel for dette enda..

Masterteoremet

Veldig greit på rekurrenser på formen:

T(n) = aT(n/b) + f(n)

Tre forskjellige caser. Her er det ofte ϵ – epsilonen som lager trøbbel. Husk også definisjonen på O() Θ() og Ω().

O() betyr at funksjonen er enn eller lik

Ω() betyr at funksjonen er større enn eller lik

Θ() betyr at funksjonen er både øvre og nedre grense, altså lik

HUSK: Dette er asymptotiske betegnelser.

La a ≥ 1 og b > 1

Case1:

Hvis

$$f(n) \in O\left(n^{\log_b a-\epsilon}\right)$$ **for ε > 0**

så

$$T(n) \in \Theta\left(n^{\log_b a}\right).$$

Case2

Hvis

$$f(n) \in \Theta\left(n^{\log_b a}\right)$$

så

$$T(n) \in \Theta\left(n^{\log_b a}\log(n)\right).$$

Case3

Hvis

$$f(n) \in \Omega\left(n^{\log_b a+\epsilon}\right)$$ **for a ε > 0**

og

$$af\left(\frac{n}{b}\right) \le cf(n)$$ **for ε > 0 og af(n/b) ≤ cf(n) når *c* < 1 og stor nok n**

så

$$T(n) \in \Theta(f(n)).$$

Case3 er farlig her. Det er lett å glemme tilleggsbetingelsene!

Grafer

Grafer er en datastruktur som ligner litt på trær, men med flere egenskaper.

Egenskaper

Retning på kantene (koblingen mellom nodene). Kantene kan enten være rettet den ene/andre veien, eller ingen retning.

Kantene i en graf kan ha verdier (som regel kostnader). Disse kan ha hvilken som helst verdi, også negative!

Sykler i grafen oppstår når man kan komme seg fra en node og tilbake igjen til samme noden ved å gå gjennom andre noder.

Hvis grafen har negative kanter og sykler kan det oppstå negative sykler. Disse oppstår når summen av kantene man går igjennom for å komme tilbake til utgangspunktet er negativt. Dette gir problemer siden man i teorien kan gå gjennom denne sykelen flere ganger og få mindre og mindre «vei».

Asykliske grafer er grafer uten sykler.

V er en betegnelse for noder. Engelsk: Vertices (noder)

En generell graf har maksimalt V^2 kanter.

Sortering

Små animasjoner av de forskjellige sorteringsalgoritmene:

http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html

Insertion-Sort

Sorterer på en relativt intuiv måte.

Tar ut det første elementet av listen og går nedover listen mens den sjekker om tallet «den står på» er større enn tallet som er tatt ut. Hvis det er mindre fortsetter den nedover listen. Når den går nedover listen bytter den plass med tallet som er mindre. Altså kreves det ingen ekstra plass til sorteringen.

Kjøretiden er **$O(n^2)$**

Insertion-Sort er en **stabil** sorteringsalgoritme

Den er altså effektiv på ganske små datamengder.

Brukes ofte som et tillegg til andre sorteringer, fks Quicksort.

Demonstrasjon: http://web.engr.oregonstate.edu/~minoura/cs162/javaProgs/sort/InsertSort.html

Mergesort

Deler opp listen i to sub-lister (omtrent halvparten hver).

Sorterer sublisten

Fletter sammen sublistene.

Mergesort er rekursiv og er et eksempel på splitt-og-hersk.

Kjøretiden er **O(n*log(n))**

Mergesort er en **stabil** sorteringsalgoritme

Demonstrasjon: http://haegar.fh-swf.de/AlgoDat/sorts/MergeSort/mergesort.html

Heapsort

Heaps er egentlig spesialserte trær. De bygger på prinsippet:

Hver node er større enn eller lik sine barn.

Binaryheaps er et perfekt eller nesten perfekt balansert binærtre.

Mer informasjon: http://en.wikipedia.org/wiki/Binary_heap

Det fine med heaps er at alle operasjoner tar O(log(n)) tid.

For at heapsort skal fungere må vi først legge alle elementene inn i en max- eller min-heap. Slik at vi kan til enhver tid kan ta ut enten det minste eller det største elementet som vi så legger inn i den sorterte listen.

Kjøretiden er **O(n*log(n))**

Heapsort er en **ikke stabil** sorteringsalgoritme

Demonstrasjon: http://www2.hawaii.edu/~copley/665/HSApplet.html

Quicksort

Velger en pivot

Flytter elementer større enn pivoten til høyre for den og de mindre enn til venstre. Hvis elementet er like stort som pivoten er det samme hvilken side det flyttes. Dette er ofte kalt partisjonering (partitioning)

Dette gjøres rekursivt til hele alle elementene er sortert.

Pivoten kan velges til å være på midten hele tiden. Men dette er ikke det mest effektive. Det er ekstremt mange muligheter for å velge pivoten, uten å si mer om det.

Kjøretiden varierer i forhold til hvordan listen ser ut. Det verste tenkelige er hvis listen allerede er sortert eller delvis sortert. Da blir kjøretiden **$O(n^2)$**. Til vanlig er kjøretiden **$O(n*log(n))$**.

Hvis det er få elementer i listen lønner det seg å bruke Insertion-sort.

Quicksort kan gjøres **Stabil** men kan også gjøres **Ustabil** avhengig av om man implementerer med in-place partisjon eller ikke.

Demonstrasjon: http://www.research.compaq.com/SRC/JCAT/jdk10/sorting/

Flere sorteringsalgoritmer

-RADIX-sort

-Bubble-sort

-Bucket-sort

-counting-sort

Minimalt Spenntre (MST)

Definisjon

Minimalt spenntre er den grafen som kobler sammen alle nodene hvor totalkostnaden er minst mulig, basert på den opprinnelige kostnadsgrafen. (kostnad = vekt) Grafen er **ikke rettet**.

Mer informasjon: http://en.wikipedia.org/wiki/Minimum_spanning_tree

<u>Prim</u>

Prim er en grådig algoritme som finner det minimale spenntreet til en vektet graf.

Startnoden er tilfeldig.

For hver node ser den nodens kanter og velger **grådig** den med minst kostnad. **Inkludert** kantene ut fra noder som allerede er besøkt! Det vil si hvis node D og E er besøkt så vil utvalget som Prim kan velge mellom være kanter ut fra D **og** E.

Prim bruker heaps som datastruktur. Ved bruk av enkel binaryheap blir kjøretiden **O(E*log(n))**

Dette betyr at prim kjører raskest på grafer med høy kanttetthet.

Kantene kan være **negative**, sykler er ikke aktuelt siden grafen ikke er rettet.

Animasjon: http://en.wikipedia.org/wiki/Prim%27s_algorithm

<u>Kruskal</u>

Kruskal er også en grådig algoritme som finner det minimale spenntreet i en vektet graf.

Kruskal ser på kantene mer enn nodene.

Først legger den alle kantene til i et set, og danner en **skog** hvor hver node er et tre. En skog er et set med flere trær.

Den velger **grådig** den kanten med minst kostnad, fjerner den fra kantsettet.

Hvis den kanten kombinerer to trær i skogen så legger den til kanten i skogen, de to trærene blir da til **ett tre.**

Kruskal stopper når skogen kun har ett tre, som er det minimale spenntreet.

Kjøretiden er **O(E*log(n))**

Animasjon: http://students.ceid.upatras.gr/~papagel/project/kruskal.htm

Korteste vei algoritmer

Prinsippet bak alle disse er å finne den korteste veien i en vektet graf.

DAG-shortest-path

DAG – Directed Asyclic Graph krever at det **ikke er sykler** i grafen, Derimot kan det være **negative kanter**.

Den finner også korteste vei fra en startnode til alle andre.

Først så ordnes nodene i topologisk rekkefølge, så settes kostnaden til alle noder til uendelig, unntatt startnoden, som får kostnaden 0. Deretter traverserer den grafen ved å ta hver enkelt kant ut fra den noden den står på og kjører RELAX på denne. Som med andre ord betyr at den tilbyr noden kanten går til en ny kostnad, hvis denne kostnaden er bedre enn den noden allerede har oppdateres kostnaden.

Dette fører til kjøretiden **O(n + E)**

Se Cormen (s. 592) for visualisering.

Djikstra (dejkstra)

Djikstra finner den korteste veien fra **en** startnode **til alle** andre noder.

Til traverseringen bruker den bredde-først-søk (BFS).

Først settes kostnaden til alle noder til **uendelig**. Når den begynner traverseringen **«tilbyr»** den noden den er på en kostnad. Den tilbydte verdien kommer fra den forrige noden sin kostnad. Hvis den nye (tilbudte) kostnaden er bedre enn den kostnaden som noden har fra før, så oppdateres noden med den beste kostnaden. Denne prosedyren kalles **RELAX**.

Algoritmen kjører til den ikke kan tilby noe bedre kostnad til noen noder. Da vil **sluttnoden** ha den **minste kostnaden** (korteste veien).

Djikstra benytter en prioriteringskø for å velge ut «neste» node. Prioriteringskøen kan implementeres på forskjellig vis, og forandrer kjøretiden drastisk.

Ved bruk av **array** medfører dette kjøretiden **$O(n^2)$**

Ved bruk av **heaps** kan kjøretiden fåes ned i **O(n*log(n) + E*log(n))** som i noen tilfeller kan være bedre. Spesielt når grafen er «sparse» altså at den har få kanter i forhold til noder. (lav kanttetthet)

Djikstra klarer ikke grafer med **negative kanter** og derfor ingen negative sykler.

For et alle-til-alle problem kan vi bare kjøre djikstra en gang for hver eneste node altså n ganger. Som medfører en kjøretid på $O(n^3)$

Animasjon

http://www.cs.sunysb.edu/~skiena/combinatorica/animations/dijkstra.html

Bellman-Ford

Bellman-Ford finner korteste vei fra **en-til-alle**. Den kan også si ifra om det finnes en negativ sykel eller ikke.

Først settes kostnaden til alle nodene **til uendelig,** unntatt startnoden, som **får kostnad 0.**

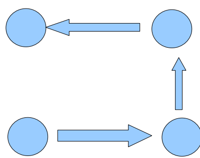
For hver node sjekkes nodens utgående kanter, og det kjøres **RELAX** på hver kant.

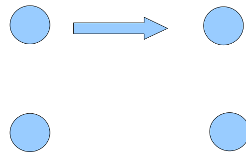
Deretter kjøres en sjekk på hver kant om den kanten den står på er større enn det som kan tilbys av foregående kant. Dette sjekker om det finnes en negativ sykel.

Siden den kjører over **hver kant** for **hver node** får den kjøretiden: **O(N*E)**

En worst-case er hvis grafen har n-1 utgående kanter som fører til $O(n^2)$, noe den i praksis alltid vil ha. Hvis ikke vil ikke alle nodene være koblet til startnoden, og de ukoblete nodene vil få kostnad lik uendelig. (se fig.)

Realistisk($e \geq n-1$): Urealistisk ($e \leq n-1$)





Den store styrken i Bellman-Ford er at den tåler negative kanter og at den kan si om det finnes negative sykler. Kjøretiden vil i praksis bli større enn Djikstra. Derfor brukes den som oftest kun når det finnes negative kanter.

Animasjon: side 589 i Cormen

Floyd-Warshall

Floyd-Warshall er en **korteste vei alle-til-alle** algoritme. Den baserer seg på en **rettet og vektet** graf representert ved en **nabomatrise.** Grafen kan **ha negative kanter** men **ikke negative sykler**.

Delproblem til Floyd-Warshall:

«Finne korteste vei fra node i til j som kun går igjennom node [1 .. k]»

Den er **ikke grådig** dvs. at den kan endre på valgene den tar.

Kjøretid: **$O(n^3)$**

Ikke så bra animasjon:

http://www.pms.informatik.uni-muenchen.de/lehre/compgeometry/Gosper/shortest_path/shortest_path.html#visualization