INF2220 - Algoritmer og Datastrukturer - Kompendium

VEGARD BERGSVIK ØVSTEGÅRD

25-11-2017

Contents

1 Introduksjon, Analyse av algoritmer, rekursjon og binærtrær

1.1 Analyse av algoritme

- En algoritme er et sett med regler som gir en sekvens med operasjoner for å løse et problem.
  - Endelig
  - Bestemt
  - Input
  - Output
  - Effektivitet

1.1.1 Analyse av tidsforbruk

Typer analyse: * Average-case * Worst-case

Alternativ: * Ta tiden for ulike størrelser på input * Finne enkel funksjon som slik som kjøretiden

1.1.2 O-Notasjon

Definisjon:

La

$$T(n)$$

være kjøretiden til programmet.

$$T(n) = O(f(n))$$

hvis det finnes positive konstanter

$$c$$

og

$$n_0$$

slik at

$$T(n) \leq c * f(n) når n > n_0$$

$$O(f(n))$$

er den øvre grense for kjøretiden. Oppgaven er å finne

$$f(n)$$

som er minst mulig.

- En ligning som beskriver hvordan kjøretiden skaleres i forhold til input variabler.
- O-Notasjon er en forenklet måte å angi tidsforbruk på.
- Funksjoner:

Funksjon	Navn
1	Konstant
log n	Logaritmisk
n	Linær
n log n	
n^2	Kvadratisk
n^3	Kubisk
2^n	Eksponensiell
n!	

1.1.3 Rekursjon

- Metode som kaller seg selv
- Huskeregler:
  - Et basistilfelle som kan løses uten rekursjon må alltid foreligge
  - Kallene må gå i retning av basistilfellet.
  - Anta at de rekursive kallene fungerer.
  - Ikke løs samme instans av et problem i separate rekursive kall!

1.1.4 Trær

- Brukes på data som er organisert hiarkisk
- Traversering:
  - Leting
  - Innsetting
  - Fjerning
  - Beregning
  - Måter:
    * Prefiks; behandle noden før barna behandles
    * Postfiks; Behandle alle barna så noden
- Lengden av en sti er antall *kanter* på veien (k-1).
- Dybde er veien fra roten til noden.

- Høyde er lengste veien fra en node til en løvnode (node uten barn).
- Implementasjon:

```
class TreNode {
 Object element;
 TreNode forsteBarn;
 TreNode nesteSosken;
 }
```

```
class TreNode {
 Object element;
 List<TreNode> barn;
 }
```

Binærtrær

- Noder har aldri mer enn to barn
- Implementasjon:

```
class BinNode{
Object element;
BinNode venstre;
BinNode hoyre;
}
```

Binære søketrær

- Protokoll:
  - Verdien i **venstre** subtre er **mindre** en verdien i noden.
  - Verdien i **høyre** subtre er **større** en verdien i noden.

Søking: Rekursiv Metode

```
public BinNode finn(Comparable x, BinNode n) {
 if (n == null) {
 return null;
 } else if (x.compareTo(n.element) < 0) {
 return finn(x, n.venstre);
 } else if (x.compareTo(n.element) > 0) {
 return finn(x, n.hoyre);
 } else {
 return n;
 }
 }
```

Her antar vi at nodene er forskjellige

Søking: Ikke-Rekursiv Metode

```
public BinNode finn(Comparable x, BinNode n) {
 BinNode t = n;
 while (t != null && x.compareTo(t.element) != 0) {
 if (x.compareTo(t.element) < 0) {
 t = t.venstre;
 } else {
 t = t.hoyre;
 }
 }
 return t;
}
```

Innsetting

- Søk nedover treet og sett inn den nye noden hvis du kommer til en null-peker.

Sletting - Vanskeligere

- Hvis noden som skal fjernes er en løvnode, kan den fjernes direkte.
- Hvis noden har bare ett barn -> Foreldrenoden "hopper" over noden som fjernes.
- Noden har to barn:
  - Erstatt verdien i noden med den miste verdien i høyre *subtre*.
  - Slett noden som denne miste verdien var i.

Rekursiv metode for å fjerne tall:

```
public BinNode fjern(Comparable x, BinNode n) {
 if (n == null) { return null; }
 if (x.compareTo(n.element) < 0) {
 n.venstre = fjern(x, n.venstre);
 } else if (x.compareTo(n.element) > 0) {
 n.hoyre = fjern(x, n.hoyre);
 } else {
 if (n.venstre == null) {
 n = n.hoyre;
 } else if (n.hoyre == null {
 n = n.venstre;
 } else {
```

```
n.element = finnMinste(n.hoyre);
n.hoyre = fjern(n.element, n.hoyre);
}
}
return n;
}
```

2 Rød-svarte trær og B-trær

2.1 Rød-Svarte trær

- Hver node er rød eller svart

1. Roten = svart
2. Hvis noden er rød, er barna svart.
3. Alle veier fra en node til en NULL peker må inneholde samme antall svarte noder.

- Høyden er da maksimalt

$$2 * log_2(N + 1)$$

2.1.1 Innsetting

- Hvis foreldernoden er svart:
  - Den nye noden settes inn som rød, antall svarte noder på veien til null-pekerne blir som før.
- Hvis foreldernoden er rød:
  - Den nye noden kan ikke være rød (2)
  - Den nye noden kan ikke være svart (3)
  - Treet må endres ved hjelp av rotasjoner og omfarging.

Zig rotasjon

Zig-zag rotasjon

2.2 B-trær

- Brukes når vi behandler MYE data.
- Hver node har mange barn
- Er balansert
- Øverste nivå i internminnet, resten på disk.
- Brukes mye i databasesystemer.

2.2.1 Tidsforbruk

- Antar at M(indre) og L(løv) er omtrent like.
- Siden hver indre node unntatt roten har minst M/2 barn, er dybden til B-treet maksimalt

$$log_(M/2)N$$

- For hver node må vi utføre

$$O(logM)$$

  arbeid(binærsøk) for å avgjør hvilken gren vi skal gå til.
- Dermed tar søking

$$O(lovM * log_(M/2)N = O)$$

  tid.
- Ved innsetting og sletting kan det hende at vi må gjøre

3 Maps og hashing

3.1 Abstrakte datatyper

- Består av:
  - Et sett med objekter
  - Spesifikasjon av operasjoner på disse.
- Eksempler:
  - Binært søketre
    - Søking
    - Fjerning
  - Mengde
    - Snitt
    - Finn
  - Map
    - Get
    - Put
    - ContainsKey
- ADT kan gjenbrukes, de er generiske
- Lage modulære programmer

3.2 Map

- Samling nøkler-, verdi-par.
- Nøkler = unike

- Typiske operasjoner:
  - containsKey
  - get
  - put
  - keySet
  - values

3.3 Hashing

- Lagrer elementer i arrays.
- Nøkkelen -> indeksen
- Rask å beregne:
  - Bruker kalkuleringen for å finne en nøkkel istedenfor søking
- Hashtabellen bruker **konstant** gjennomsnittstid ved:
  - innsetting
  - søking
  - sletting
- NB! La alltid tabellstørrelsen være et *primtall*!

3.3.1 Hash-funksjoner

- Må kunne *gi alle mulige verdier* fra 0 til tableSize -1.
- Må gi en *god fordeling* utover tabellindeksene.

3.3.2 Kollisjonhåndtering

- *Åpen hashing/ Separate chaining* : Elementer med samme hashverdi samles i en liste.
  - Ulempe: Tar mere plass.
- *Lukket hashing - Åpen adressering*: Finner en annen indeks som er ledig dersom den ønskede var opptatt.
  - Brukes mest.
  - Trenger større tabell en for åpen hashing.
  - Lineær prøving
    - Dette gir primary-clustering
  - Kvadratisk prøving
    - Kan gi secondary-clustering
  - Dobbel hashing
    - Bruker en ny hash-funksjon for å løse kollisjonene.
    - Kan gi veldig god spredning hvis sekundær-funksjonen er passende.
- Load-faktoren

$$\lambda$$

er antall elementer i hash-tabellen i forhold til tabellstørrelsen.
  - Åpen hashing ønsker vi

$$\lambda = 1.0$$

  - Lukket hashing: Ønsker

$$\lambda < 0.5$$

    , litt sløsing med plass.
- Rehashing:
  - Metoden når tabellen blir for full.
  - Løsninger:
    * Lag ny tabell som er dobbelt så stor(Fortsatt primtall!).
  - Dyr operasjon, men opptrer sjeldent.

3.4 Prioritetskø & heap

3.4.1 Prioritetskø

- Lav tall = høy prioritet
- Må minst ha `insert(x,p)` og `deleteMin()`
- Mest brukte datastruktur er Heap grunnet gunstig O-notasjon.

3.5 Binær Heap (Heap)

- Et "fullt" binær tre med et *strukturkrav* og *ordningskrav*
  - Løv nivået er fylt fra venstre til høyre {% capture images %} /images/2220-4-1.jpg {% endcapture %} {% include gallery images=images caption="Binary heap" cols=1 %}
- Barn må alltid være større eller lik sine foreldre (NB! Når lavt tall = Høy prioritet)

3.5.1 Strukturkrav og egenskaper

- Treet må være i perfekt balanse
- Bladnoder vil ha høyde på maks 1
- Treet med høyden *h* har mellom

$$2^h$$

  og

$$2^{(h+1)} - 1$$

  noder
- Den maksimale høyden på treet vil være

$$log_2(n)$$

3.5.2 Representasjon

- Siden binærtreet er komplett kan vi legge elementene i en array
- Finn elementer:
  - Venstre barn:
    $$index * 2$$
  - Høyre barn:
    $$index * 2 + 1$$
  - Foreldre: (int)
    $$index/2$$

{% capture images %} /images/2220-4-2.jpg {% endcapture %} {% include gallery images=images caption="Binary heap" cols=1 %}

3.5.3 insert

- Plasser elementet "bakerst" i treet
- Sørg så for at ordningskravet opprettholdes og la elementet "boble" opp.
- Vi setter inn 14: {% capture images %} /images/2220-4-3.jpg /images/2220-4-4.jpg {% endcapture %} {% include gallery images=images caption="Inserting" cols=1 %}
- Dette tar
  $$O(log_2(n))$$

3.5.4 deleteMin

- Fjerner root og setter det siste elementet som root
- Opprettholder ordningskravet ved å la elementet "synke" nedover {% capture images %} /images/2220-4-5.jpg /images/2220-4-6.jpg /images/2220-4-7.jpg {% endcapture %} {% include gallery images=images caption="deleteMin" cols=1 %}
- Dette tar
  $$O(log_2(n))$$

3.5.5 Andre Operasjoner

- `findMin` kan gjøres i
  $$O(0)$$
- `delete`fjern vilkårlig element fra heapen
  - `decreaseKey`
    $$\infty +$$

    `deleteMin`

- Endring av prioritet på elementer:
  - Senking: `increaseKey`
  - Øking: `decreaseKey`
    * Lokaliser element
    * Øk eller senk
    * La elementet flyte/boble

3.6 Huffman Codes

- Taps-løs data komprimerings-algoritme
- Baserer seg på bit-lengde og frekvensen til bestemte koder eller tegn ### Algoritmen:

1. Finn de to elementene som forekommer minst
2. Lag et 2-blads tre av de
3. Finn det neste bladet som forekommer minst og legg det til

- Video {% capture images %} /images/2220-4-8.JPG /images/2220-4-9.JPG /images/2220-4-10.JPG /images/2220-4-11.JPG /images/2220-4-12.JPG /images/2220-4-13.JPG /images/2220-4-14.JPG {% endcapture %} {% include gallery images=images caption="Huffman's awesomeness" cols=1 %}

3.7 Venstreorientert heap - *Video*

- Variant av binær heap
- Samme ordningskrav
- Strukturkrav:
  - *Null path length**(npl(x))*** er den korteste veien fra x til en node uten to barn
  - **npl(l)**

    $$\geq$$

    npl(r) *l=left & r=right*
  - Forsøker å være ubalansert
- Fletting av to binære heaper `merge` tar

  $$O(N)$$

  for heaper med like størrelser
- **Venstreorientert Heap** støtter merg i

  $$O(logn)$$

3.8 Heapsort - *Video*

- Kort fortalt:
  - `buildMaxHeap()`

- `deleteMax()`

  $$n$$

  times

4 Grafer

4.1 Definisjoner

- **graf G=(V,E)**
  - V = noder (**Vertises**)
  - E = kanter (**Edges**)
- |V| og |E| er antall noder og kanter i grafen
- En kant **E** er et node-par:

  $$(u, v)$$

  slik at

  $$u, v \in V$$

- **Rettet** graf:
  - Rekkefølge
- **Urettet** graf:
  - Ingen rekkefølge {% capture images %} /images/2220-5-1.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}
- **Alt kan moduleres med graf!** {% capture images %} /images/2220-5-2.jpg {% endcapture %} {% include gallery images=images caption="" cols=1 %}
- **Nabo-node / etterfølger:** {% capture images %} /images/2220-5-3.jpg {% endcapture %} {% include gallery images=images caption="" cols=1 %}
- **Vektet** graf har en tredje komponent kalt vekt {% capture images %} /images/2220-5-4.jpg {% endcapture %} {% include gallery images=images caption="" cols=1 %}
- En **vei/sti** er en sekvens av noder *v1,v2,v3,..,vi*
- **Lengden** er antall kanter på veien (n-1) {% capture images %} /images/2220-5-5.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}
- **Kosten** til en vei er summen av vektene langs veien {% capture images %} /images/2220-5-6.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}
- Hvis alle nodene på veien er forskjellige er veien **enkel**
- En **løkke/sykel** er en vei slik at v1=vn

  - Løkken er **enkel** dersom stien er enkel
- I en urettet graf må også alle kanter i løkken være forskjellige {% capture images %} /images/2220-5-7.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}
- **Asyklisk** graf har ingen løkker
- En **DAG(Directed, Acyclic graf)** er en rettet, asyklisk graf
- En urettet **sammenhengende** graf har en vei fra en node til alle andre noder {% capture images %} /images/2220-5-8.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}
- **Sterkt sammenhengende** rettet graf har en vei fra hver node til alle andre noder
- **Svakt sammenhengende** rettet har ikke dette
- Urettet graf:
  - **Graden** til en node i en urettet graf er antall kanter mot noden
- Rettet graf:
  - **Inngraden** til til en node er antall kanter inn til noden
  - **Utgraden** til en node er antall kanter ut fra noden
- Nabo-matrise: {% capture images %} /images/2220-5-9.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}
- Nabo-liste: {% capture images %} /images/2220-5-10.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

4.2 Topologisk sortering

- En rekkefølge av noder i en **DAG**.
  - Hvis det er en vei fra *vi* til *vj* så kommer *vj* **etter** *vi* {% capture images %} /images/topo.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

4.3 Korteste vei en-til-alle uvektet graf

- Den veien fra *s* til *t* som bruker færrest kanter
  - Alle kanter har en "vekt" = 1
- Korteste vei:
  - Velg alle nodene med avstand 1 fra *s*, så alle med 2,. . . , så alle med *n* osv.
  - Mer generelt: Velger hele tiden en ukjent node blant dem med minst avstand fra *s*

4.4 Korteste vei en-til-alle vektet graf (Djikstra<3)

Video {% capture images %} /images/djikstra.jpg {% endcapture %} {% include gallery images=images caption="" cols=1 %}

4.5 Dijkstra fortsetter

{% capture images %} /images/dijkstraalgo.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

Algoritmen fungerer fordi: * Ingen kjent node har større avstand til **s** enn en ukjent node * Alle kjente noder har riktig korteste vei satt

4.5.1 Tidsforbruk

Hvis vi leter sekvensielt etter den ukjente node med minst avstand tar dette

$$O(V)$$

tid, noe som gjøres

$$V$$

ganger, så total tid for å finne minste avstand blir

$$O(V^2)$$

. I tillegg oppdateres avstandene, maksimalt en oppdatering per kant, dvs. til sammen

$$O(|E|)$$

Raskere implementasjon(for tynne grafer):

- Bruker en prioritetskø til å ta vare på ukjente noder med avstand mindre enn uendelig.
- Prioriteten til ukjent node forandres hvis vi finner kortere vei til noden

4.6 Minimale spenntrær

Et minimalt spenntre for en urettet graf **G** er et tre med kanter fra grafen, slik at alle nodene i **G** er forbundet til lavest mulig kostnad {% capture images %} /images/spenntre.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

4.7 Grådige Algoritmer

Disse algoritmene prøver i hvert trinn å gjøre det som *ser best ut der og da* i håp om at det er den *totalt beste løsningen*. Eksempelvis hvis en rik person med bare 5, 10 og 20 kroninger skal gi vekk 35 kr. Er den den grådige løsningen å begynne med å gi den høyeste høyeste mynten 20, så 10 så 5. **Grådig** kommer av å bruke minst mulig energi der og da. Dette er svært raske algoritmer, men de kan ikke løse alle problemer.

4.7.1 Prims algoritme

Denne algoritmen er essensielt lik Dijkstras algoritme for korteste vei. Den minste vekten til en kant som forbinder **v** med en **kjent** node er avstanden til en ukjent node. Hver kan befinner seg i **2** nabolister fordi vi har urettede grafer. {% capture images %} /images/prim1.JPG /images/prim2.JPG {% endcapture %} {% include gallery images=images caption="" cols=2 %}

4.7.2 Kruskals algoritme

Kruskals algoritme ser på kantene en etter en, og sorterer de etter minst vekt. Kanten aksepteres hvis og bare hvis den ikke fører til noen løkke. Typisk implementers algoritmen med en prioritetskø og disjunkte mengder.

4.7.3 Prim vs. Kruskals

- Prims algoritme er noe mer effektiv for spesielt tette grafer.
- Prims algoritme virker bare i sammenhengende grafer.
- Kruskals algoritme gir et minimalt spenn-tre i hver sammenhengskomponent i grafen.

4.8 Dybde-først søk

Dette er en klassisk graf traversering og en generalisering av **prefiks** traversering for trær.

4.8.1 Urettet graf & DFS

En **urettet** graf er sammenhengende hvis og bare hvis et dybde-først søk som starter i en tilfeldig node, besøker alle nodene i grafen

4.8.2 Rettet graf & DFS

En **rettet** graf er sterkt sammenhengende hvis og bare hvis vi fra hver eneste node **v** klarer å besøke alle de andre nodene i grafen ved et dybde-først søk fra **v**

Hvis grafen ikke er sammenhengende, kan vi foreta nye dybde-først søk fra noder som ikke er besøkte, inntil alle nodene er behandlet

4.8.3 Urettet grafer

Dybde-først-spenntre

- For urettet sammenhengende
- ikke sammenhengende grafer: dfs spanning forest
- huske "back-pointers"

Ulike typer kanter

- Tree edges -> tilhører treet
- **Back** edges -> tilhører IKKE treet {% capture images %} /images/dfstre.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

4.9 Biconnectivity

4.9.1 Definisjon

En sammenhengende urettet graf er bi-connected hvis det ikke er noen noder som ved fjerning gjør at grafen blir ikke sammenhengende. Slik node heter cut-vertices eller articulation point.

4.9.2 Egenskap

- Det er to disjunkte stier mellom hvilke to noder som helst.
- F.eks. Nettverk-feiltoleranse: dersom en ruter går ned, finnes det alltid en alternativ vei å rute datapakkene på.
- Viktig å identifisere disse nodene. Single point to failure!

4.10 Strongly Connected Components

4.10.1 Rettet graf & dfs

En rettet graf er sterkt sammenhengende hvis og bare hvis vi fra hver eneste node v klarer å besøke alle de andre nodene i grafen ved et dybde-først søk fra v

4.10.2 Definisjon

Gitt en rettet graf G = (V, E). En strongly connected component av G er en maksimal sett av noder

$$U \subseteq V$$

s.t.: for alle

$$u_1, u_2 \in U$$

vi har at

$$u1 \rightarrow *u2$$

and

$$u2 \rightarrow *u1$$

.

Tanken er:

Hvis v kan nå alle noder og alle noder kan nå v, så kan alle nå alle ved å gå gjennom v.

5 Sortering

5.1 Effekten av JIT-kompilering

- Programmet optimaliseres (omkopmileres) under kjøring flere ganger av optimalisatoren i JVM (java) stadig raskere.
  - Første gang en metode kjøres, oversettes den fra Byte-kode til maskinkode
  - Blir den brukt flere/mange ganger optimaliseres denne maskinkoden i en eller flere steg (minst 2 steg)
  - Denne prosessen kalles JIT (Just In Time)-kompilering
  - God idé: kode som brukes mye skal gå raskest mulig
  - Det er nå over 100 programmeringsspråk (også Python, som Jython) som bruker JVM dette gjelder da alle disse språkene, ikke bare Java.

{% capture images %} /images/jitcomp.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

5.1.1 Tidtaking med JIT-kompilatoren

```
double [] tider = new double[numIter];
//nHigh =10mill, nLow=100, nstep =10, numIter = 3 eller helst 5
for (n = nHigh; n >= nLow; n= n/nStep) {
    for (med = 0; med < numIter; med++) {
        long t = System.nanoTime(); // start tidtagning i nanosek.
        < kode du tar tider på >;
        tider[med] = (System.nanoTime()-t)/1000000.0; // millisek.
}}
```

5.2 Sortering

- Sorteres: Tall, tekster eller noe mer sammensatt.
- Hva avgjør tidsforbruket ved sortering
  - Sorteringsalgoritmen
  - N, antall elementer vi sorterer
  - Fordelingen av disse (Uniform, skjeve fordelinger, spredte,..)
  - Effekten av caching
  - Optimalisering i jvm (Java virtual machine) også kalt >java
  - Parallell sortering eller sekvensiell

5.2.1 Krav:

- Riktig - opplagt
- Rask
  - For alle typer fordelinger
  - For alle datatyper (int, double, tekst, ..)
  - Vanligste test er sortering av n heltall, trukket U(0:n)
- Stabil
- Bruker lite plass - ikke så viktig lengre

5.2.2 Sorterings klasser.

Sammenligning-baserte:

- Baserer seg på sammenligning av to elementer
  - Innstikk, boble
  - Shellsort
  - TreeSort
  - Quicksort

Verdi-baserte:

- Direkte plassering basert på verdien av hvert element ingen sammenligninger med nabo-elementer e.l.
  - Bøtte
  - VenstreRadix og HøyreRadix

5.2.3 Sorterings problemet:

- Kaller arrayen a[] før sorteringen og a[ ] etter
- Sorteringskravet
- Stabil sortering
- Sorteringsalgoritmene må virke hvis er to (eller fler) like verdier i a[ ]
- I testkjøringene antar vi at innholdet i a[] er en tilfeldig trukne tall mellom 0 og n-1. Dette betyr at også etter all sannsynlighet er dubletter (to eller flere like tall) som skal sorteres , men ikke så veldig mange.
- Hvor mye ekstra plass bruker algoritmen
- Bevaringskriteriet

5.2.4 Bubblesort - Aldri bruk denne

Ide: Bytt om naboer hvis den som står til venstre er størst, lar den minste boble venstreover

```
void bytt(int[] a, int i, int j)
    { int t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
void bobleSort (int [] a)
    {int i = 0;
        while ( i < a.length-1)
            if (a[i] > a[i+1]) {
            bytt (a, i, i+1);
            if (i > 0 ) i = i-1;
        } else {
            i = i + 1;
        }
} // end bobleSort
```

5.2.5 Innstikk sortering - best for n < 50

- Idé: Ta ut ut element a[k+1] som er mindre enn a[k]. Skyv elementer k, k-1,… ett hakk til høyre til a[k+1] kan settes ned foran et mindre element.

```
void insertSort(int [] a )
    {int i, t, max = a.length -1;
    for (int k = 0 ; k < max; k++) {
    // Invariant: a[0..k] er sortert, skal
    // nå sortere a[k+1] inn på riktig plass
        if (a[k] > a[k+1]) {
        t = a[k+1];
        i = k;
        do{ // gå bakover, skyv de andre
            // og finn riktig plass for t
            a[i+1] = a[i];
            i--;
        } while (i >= 0 && a[i] > t);
        a[i+1] = t;
        }
    }
} // end insertSort
```

- Fungerer ofte som en hjelpe algoritme

5.2.6 Radix algoritmer

- To typer:
  - RR: fra høyre og venstre-over (vanligst iterativ og rask) RR: 1234**5**
  - LR: fra venstre og høyre-over (rask - rekursiv) LR: **1**2345 #### Algoritmene:
- Begge: Finner først max verdi i a [ ]
  - = bestem største *siffer* i alle tallene

1. Tell opp hvor mange elementer det er av hver verdi på det sifferet (hvor mange 0-er, 1-ere, 2-..) man sorterer på
2. Da vet vi hvor 0-erne skal være, 1-erne skal være, etter sortering på dette sifferet ved å addere disse antallene fra 0 og oppover.
3. Flytter så elementene i a[] direkte over til riktig plass i b[]

5.3 Shell-Sortering, forbedring av Innstikksort

- Video ### Analyse:
- Den virker fordi når gap = 1 fungere den som en Innstikksort.
- Den er vanligvis raskere fordi vi har nesten sortert a[] før gap=1. Når a[] er delvis sortert, sorterer innstikksortering meget kjapt.
- Worst case som med innstikk:

$$O(n^2)$$

- Mye raskere med andre, lure valg av verdier for gap

$$O(n^{3/2})$$

  eller bedre:
  - Velger primtall i stigende rekkefølge som er minst dobbelt så store som forgjengeren + n/(på de samme primtallene):

$$(1, 2, 5, 11, 23...., n/23, n/11, n/5, n/2)$$

- Meget lett å lage sekvenser som er betydelig langsommere enn Shells originale valg, f.eks bare primtallene.
  - En slik sekvens begynner på 1

5.3.1 Shell2 – en annen sekvens for gap

```
void Shell2Sort(int [] a)
    { int [] gapVal = {1,2,5,11,23, 47, 101, 291, n/291, n/101,n/47,n/23,n/11,n/5,n/2
        int gap;

    for (int gapInd = gapVal.length -1; gapInd >= 0; gapInd --) {
        gap = gapVal[gapInd];
```

```
        for (int i = gap ; i < a.length ; i++)
            if (a[i] < a[i-gap] ) {
                int tmp = a[i],
                    j = i;
                do
                { a[j] = a[j-gap];
                    j = j- gap;
                } while (j >= gap && a[j-gap] > tmp);
                a[j] = tmp;
                }
            }
    }
```

5.4 MaxSort - enkel og langsom

$$(O(n2))$$

```
void maxSort ( int [] a) {
    int max, maxi;
    for ( int k = a.length-1; k >= 0; k--){
        max = a[0]; maxi=0;
        for (int i = 1; i <=k; i++) {
            if (a[i] > max) {
                max = a[i];
                maxi =i;}
            }
        bytt(k, maxi);
    } // end for k
} // end maxSort

void bytt (int k, int m){
    int temp = a[k];
    a[k] = a[m];
    a[m] = temp;
}
```

5.5 Tre og Heap-sortering

- Tre – sortering:
  - Vi starter med røttene, i først de minste subtrærne, og dytter de ned (får evt, ny større rotverdi oppover)
- Heap-sortering:

  - Vi starter med bladnodene, og lar de stige oppover i sitt (sub)-tre, hvis de er større enn rota.
- Felles:
  - Etter denne første ordningen, er nå største element i a[0]:w

5.5.1 Tre sortering

```
void dyttNed (int i, int n) {
// Rota er (muligens) feilplassert
// Dytt gammel nedover
// få ny større oppover
int j = 2*i+1, temp = a[i];
    while(j <= n )
    { if ( j < n && a[j+1] > a[j] ) j++;
        if (a[j] > temp) {
            a[i] = a[j];
            i = j;
            j = j*2+1;
        }
        else break;
    }
    a[i] = temp;
} // end dyttNed

void treeSort( int [] a)
    { int n = a.length-1;
    for (int k = n/2 ; k > 0 ; k--) dyttNed(k,n);
    for (int k = n ; k > 0 ; k--) {
        dyttNed(0,k); bytt (0,k);
    }
}
```

Analyse av tree-sortering

- Den store begrunnelsen: Vi jobber med binære trær, og 'innsetter' i prinsippet n verdier, alle med vei

$$log_2 n$$

til

$$rota = O(nlogn)$$

5.5.2 Heap-Sortering

```
void dyttOpp(int i)
// Bladnoden på plass i er
// (muligens) feilplassert
// Dytt den oppover mot rota
{ int j = (i-1) / 2,
    temp = a[i];
    while( temp > a[j] && i > 0 ) {
        a[i] = a[j];
        i = j;
        j = (i-1)/2;
    }
    a[i] = temp;
} // end dytt OPP

void heapSort( int [] a) {
    int n = a.length -1;
    for (int k = 1; k <= n ; k++)
        dyttOpp(k);

    bytt(0,n);
    for (int k = n-1; k > 0 ; k--) {
        dyttNed(0,k);
        bytt (0,k);
    }
}
```

Analyse av Heap -sortering

- Som Tre-sortering: Vi jobber med binære trær (hauger) , og innsetter i prinsippet n verdier, alle med vei

$$log_2$$

til

$$rota = O(nlogn)$$

5.5.3 Quicksort

- Video

```
void quicksort ( int [] a, int left, int right)
    { int i= l, j=r;
```

```
    int t, part = a[(left+right)/2];
    while ( i <= j) {
        while ( a[i] < part ) i++; //hopp forbi små
        while (part < a[j] ) j--; // hopp forbi store

        if (i <= j) {
            // swap en for liten a[j] med for stor a[i]
            t = a[j];
            a[j]= a[i];
            a[i]= t;
            i++;
            j--;
        }
    }
    if ( left < j ) { quicksort (a,left,j); }
    if ( i < right ) { quicksort (a,i,right); }
} // end quickSort
```

Quicksort - tidsforbruk

- Vi ser at ett gjennomløp av quickSort tar

  $$O(r-l)$$

  tid, og første gjennomløp

  $$O(n)$$

  tid fordi

  $$r-l=n$$

  første gang

Verste tilfellet

Vi velger 'part' slik at det f.eks. er det største elementet hver gang. Da får vi totalt n kall på quickSort , som hver tar

$$O(n/2)$$

tid i gj.snitt – dvs

$$O(n^2)$$

totalt

Beste tilfellet

Vi velger part slik at den deler arrayen i to like store deler hver gang. Treet av rekursjons-kall får dybde log n. På hvert av disse nivåene gjennomløper vi alle elementene (høyst) en gang dvs:

$$O(n) + O(n) + ... + O(n) = O(nlogn)$$

(log n ledd i addisjonen)

Gjennomsnitt

I praksis vil verste tilfellet ikke opptre – men kan velge 'part' som medianen av

$$a[l], a[(l+r)/2$$

og

$$a[r]$$

og vi får "alltid"

$$O(n * log(n))$$

Quicksort i praksis

- Valg av partisjoneringselement 'part' er vesentlig
- Quicksort er ikke den raskeste algoritmen (f.eks er Radix minst dobbelt så rask), men Quicksort nyttes mye – f.eks i java.util.Arrays.sort();
- Quicksort er ikke stabil (dvs. to like elementer i inndata kan bli byttet om i utdata)

5.5.4 Lamoto Quicksort

```
void lamotoQuick( int[] a, int low, int high) {
    // only sort arrayseggments > len =1
    int ind =(low+high)/2, piv = a[ind];
    int storre=low+1, // hvor lagre neste 'storre enn piv'
    mindre=low+1; // hvor lagre neste 'mindre enn piv'
    bytt (a,ind,low); // flytt 'piv' til a[lav] , sortér resten
    while (storre <= high) {
        // test iom vi har et 'mindre enn piv' element
        if (a[storre] < piv) {
            // bytt om a[storre] og a[mindre], få en liten ned
            bytt(a,storre,mindre);
            ++mindre;
        } // end if - fant mindre enn 'piv'
        ++storre;
    } // end gå gjennom a[i+1..j]
```

```
    bytt(a,low,mindre-1); // Plassert 'piv' mellom store og små
    if ( mindre-low > 2) lamotoQuick (a, low,mindre-2); // sortér alle <= piv
        // (untatt piv)
    if ( high-mindre > 0) lamotoQuick (a, mindre, high); // sortér alle > piv
} // end sekvensiell Quick
```

Fordeler:

- Meget enklere å få riktig
- Litt langsommere, men lett optimaliseres ved å legge inn flg. linjer hvis det er flere like elementer i a[]:

```
int piv2 = mindre-1;
while (piv2 > low && a[piv2] == piv) {
    piv2--; // skip like elementer i midten
}
```

5.5.5 Flette - sortering (merge)

- Velegnet for sortering av filer og data i primær minnet. #### Generell idé:

1. Vi har to sorterte sekvenser (eller arrayer) A og B (f.eks på hver sin fil)
2. Vi ønsker å få en stor sortert fil C av de to.
3. Vi leser da det minste elementet på 'toppen av' A eller B og skriver det ut til C, ut-fila
4. Forsett med pkt. 3. til vi er ferdig med alt.
5. Rask, men krever ekstra plass.
6. Kan kodes rekursivt med fletting på tilbaketrekking.

I praksis skal det meget store filer til, før du bruker flettesortering. 16 GB intern hukommelse er i dag meget billig (noen få tusen kroner). Før vi begynner å flette, vil vi sortere filene stykkevis med f.eks Radix, Kvikk- eller Bøtte-sortering

5.5.6 Skisse:

```
Algoritme fletteSort ( innFil A, innFil B, utFil C)
{
    a = A.first;
    b = B. first;

    while ( a!= null && b != null)
        if ( a < b) { C.write (a); a = A.next;}
        else { C.wite (b); b = B.next;}

    while (a!= null) { C.write (a); a = A.next;}
```

```
    while ( b!= null) { C.write (b); b = B.next;}
}
```

5.6 Stabilitet

- Innstikk: stabil
- Quick: ikke stabil
- HRadix: stabil
- VRadix: ikke stabil, men kan sette stabil
- TreSort: ikke stabil
- MaxSort: ikke stabil, avhengig av: `if (a[i] > max)` eller `if(a[i] >= max)`
- Flette: ikke nødvendigvis

5.7 Oppsummering:

- Mange sorteringsmetoder med ulike egenskaper (raske for visse verdier av n, krever mer plass, stabile eller ikke, spesielt egnet for store datamengder,...)
- Algoritmer:
  - Boblesort : bare dårlig (langsomst)
  - Innstikksort: raskest for
    $$n < 0\breve{}50$$
  - Maxsort – langsom, men er et grunnlag for Heap og Tre
  - Tre-sortering: Interessant og ganske rask :
    $$O(n * log(n))$$
  - Quick: rask på middelstore datamengder (ca. n = 50 -5000)
  - Radix-sortering: Klart raskest når n > 500 , men HøyreRadix trenger mer plass (mer enn n ekstra plasser – flytter fra a[] til b[] + count[])

6 Tekstalgoritme

6.1 Pattern matching algorithms

Algoritmer for lokalisering av substrenger * Brute force * Enkleste tenkelige algoritme for a løse problemet * Boyer Moore (Horspool) * Relativt komplisert algoritme, med rask **worst case**

6.1.1 Brute force

Brute force er typisk: * unødvendig tung * dårlig * treg * lite gjennomtenkt * nødløsning dog noen ganger nødvendig

Video

6.1.2 Bouyer Moore

- Skal vi øke hastigheten må vi minske antall sammenligninger
  - Vi kan preprosessere informasjonen i nål og høystakk
  - Vi kan gjøre rimelige antagelser om input
- Boyer Moore antar at vi bare har 1-byte characters
- 1-byte characters gir oss 256 muligheter (28 = 256)
- Vi kan bruke den informasjonen til å preprosessere nålen

Gjennomgang:

- Vi matcher baklengs med Boyer Moore
  - Begynner på slutten og går bakover
- Merk at elementet z ikke finnes i nalen
- Dvs. etter første match kan nalen flyttes 3 hakk frem

Video

Enkel sudo kode:

```
BouyerMoore(String H, String N) //H has n characters, N has m characters
    i = m-1
    j = m-1
    repeat
        if N[j] = H[j]
            if j = 0
                return i //a match
            else
                //last() char returns -1 if not found
                i = i + m - min(j, 1 + last(H[i]))
                j = m-1
            until i > n-1
            return -1;
```

Bad Character Shift

- Vi ma raskt kunne svare pa om en bokstav er med i nalen

- Bokstaver er 1-byte lange dvs. (int) bokstav [0, 255]
- badCharShift er en array int[256]
- Vi fyller denne med shift verdier ut i fra hva som er i nalen

Sudo kode

```
int[] badCharShift = new int[256]; // assume 1-byte characters

for(int i = 0; i < badCharShift.length; i++){
    badCharShift[i] = needle.length;
}

/* shift size = 1 for characters inside needle */

for(int i = 0; i < needle.length; i++){
    badCharShift[ (int) needle[i] ] = 1;
}
```

{% capture images %} /images/bcs.JPG {% endcapture %} {% include gallery images=images caption="" cols=1 %}

Boyer Moore Horspool

```
public int boyer moore horspool(char[] needle, char[] haystack){

    if ( needle.length > haystack.length ){ return -1; }

    int[] bad shift = new int[CHAR MAX]; // 256

    for(int i = 0; i < CHAR MAX; i++){
        bad shift[i] = needle.length;
    }

    int offset = 0, scan = 0;
    int last = needle.length - 1;
    int maxoffset = haystack.length - needle.length;

    for(int i = 0; i < last; i++){
        bad shift[needle[i]] = last - i;
    }

    while(offset <= maxoffset){
        for(scan = last; needle[scan] == haystack[scan+offset]; scan--){
```

```
            if(scan == 0){ // match found!
                return offset;
            }
        }
        offset += bad shift[haystack[offset + last]];
    }
    return -1;
}
```

Horspool er en forenkling av Boyer-Moore-streng-søkealgoritmen

- Bad character shift er effektiv til for eksempel a søke i tekst i naturlige sprak fordi mismatches er sannsynlige
- Med fa alfabeter er det sannsynlig a fa matching nær slutten av nalen. I dette tilfellet kan vi ha nytte av a vurdere vellykket match-suffikser av nalen.

Mer Boyer Moore

Boyer-Moore-algoritmen er basert pa: 1. a analysere nalen baklengs 2. bad character Shift 3. good suffix shift

- bad character shift unngar a gjenta mislykkede sammenligninger mot et tegn i høystakken
- good suffix shift beregner hvor langt vi kan flytte nalen, basert pa antall matchende bokstaver før mismatch
- justerer bare matchende nal-tegn mot høystakk-tegn som allerede har fatt match
- good suffix shift er en array som er like lang som nalen

Video