INF2220 – høsten 2007
15. okt. og 22. okt.

Noe om matematiske bevis (kap. 1)
Sortering del I og II (kap. 7.)

Arne Maus,
Gruppen for objektorientering, modellering og språk (OMS)
Inst. for informatikk, Univ i Oslo

Essensen av kurset

- Lære et sett av gode (og noen få dårlige) algoritmer for å løse kjente problemer
  - Gjør det mulig å vurdere effektiviteten av programmer
- Lære å lage effektive & velstrukturerte programsystemer/biblioteker
- Lære å løse ethvert "vanskelig" problem så effektivt som mulig.
  - Også lære noen klasser av problemer som **ikke** kan løses effektivt
- Eks: Hvor lang tid tar det å sortere 1 million tall ?
  - 16 timer og 10 min (optimalisert Boblesortering)
  - 1,4 sek. (Quick-sort)
  - 0,8 sek. (radix-sort)

Summering

Definisjon :

$$\sum_{i=l}^{k} a_i = a_l + a_{l+1} + ... + a_k$$

Tre kjente summer :

a) $$\sum_{i=0}^{n} 2^i = 1 + 2 + 4 + 8 + ... + 2^n = 2^{n+1} - 1$$

b) $$\sum_{i=1}^{n} i = 1 + 2 + 3 + 4 + ... + n = \frac{n(n+1)}{2}$$

Bevis :

Kall summen S :

$1 + \quad 2 \quad + ... + n = S$

$n + (n-1) + ... + 1 = S$

Summerer disse to :

$(n+1) + (n+1) + ... + (n+1) = 2S$

$$S = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Summering 2

c) Geometrisk rekke :

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + ... + a^n = \frac{1 - a^{n+1}}{1 - a}$$

Bevis (kaller $\sum_{i=0}^{n} a^i$ for S) :

$S = 1 + a + a^2 + ... + a^n \qquad (*)$

multipliserer (*) med a :

$aS = \quad a + a^2 + ... + a^n + a^{n+1} \quad (**)$

(*) - (**) :

$S - aS = 1 - a^{n+1}$

$S(1-a) = 1 - a^{n+1}$

Fire måter å bevise/motbevise et matematisk utsagn (teorem)

- Positivt bevis ved gradvis omforming
- Induksjonsbevis
- Motbevis med ett moteksempel
- Bevis ved selvmotsigelse

Positivt bevis

- Gitt teorem T vi skal bevise
- Start med et utsagn S vi vet er riktig
- Gjør lovlige operasjoner på S slik at vi i ett eller flere steg får omformet S til T

$T: \forall heltall\ n > 3 : n^2 > 2n+1$

Bevis:

Vet $S: n > 3$,

ganger så med n på begge sider av '>' med n og får:

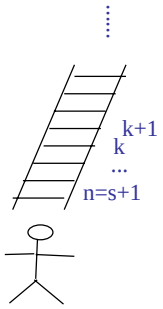
$n^2 > 3n = 2n + n$,

og siden vi vet at $n > 1$, kan vi erstatte den siste 'n'-en med 1 på høyre siden og får:

$n^2 > 2n + n > 2n + 1$ som er T

Induksjonsbevis

1. Gitt en 'matematisk' sats vi skal vise *for alle* heltall n > s
   (s konstant, typiske verdier : 0,1,2,3,..)
2. Viser først satsen for minste verdi (s + 1)
3. Antar så at satsen er riktig for n=k ,
   (k et vilkårlig heltall > s)
4. Bruker så denne antagelsen 3) til å vise at satsen også er riktig for n=k+1.
5. Da har vi vist satsen generelt



Stige-sammenligningen: Har vi vist at vi kan gå opp på nederste trinn, og så at vi *alltid kan gå ett trinn til*, kan vi gå så høyt vi vil .

induksjons-eks:

1) Skal vise: $2^n > n^2, \forall n > 4$

2) Basis, riktig for $n = 5$: $2^5 > 5^2$, $32 > 25$

3) Antar $2^k > k^2$ for vilkårlig $k > 4$

4) Vi har da (for $n = k+1$):

v.s: $2^{k+1} = 2^k 2 > k^2 2 = k^2 + k^2$

h.s: $(k+1)^2 = k^2 + 2k + 1$

Vi har v.s > h.s, fordi $k^2 > 2k+1$, $\forall k > 4$

(dette kan du evt. selv vise pr. induksjon, men vi har jo nettopp vist dette positivt for $\forall$ k > 3)

5) Vi har nå vist satsen fordi "vi kan gå opp på første trinnet på stigen" (n = 5) og alltid gå ett trinn til (fra k til k + 1 for alle verdier av k)

Moteksempel

- Vi skal motbevis et utsagn som hevder at noe gjelder for alle av en viss gruppe (tall,..)
- Metode: Finn et (enkelt) eksempel hvor påstanden er gal for ett element av denne gruppa
  - Eks
    - Påstand: $2k + 1 > k^2 \ \forall \ k > 0$
    - Motbevis: **Nei**, gjelder ikke for k=3
      fordi: **$2*3 + 1 < 3^2$**

Bevis ved selvmotsigelse

1. **Gitt en påstand som skal vises**
   Eks: Det finnes et uendelig antall primtall
   (tall som bare lar seg dele på 1 og seg selv uten rest)
2. **Anta det motsatte:**
   Det finnes bare et endelig antall primtall, og da også et største :
   $p_1, p_2, p_3, p_4, \ldots\ldots, p_m$ (hvor p_m er det største primtallet)
3. **Lag en selmotsigelse** ved å bruke antagelsen og ellers riktige omforminger av denne og andre kjente fakta.:
   lager: $s = p_1 * p_2 * p_3 * p_4 * \ldots\ldots * p_m + 1$
   men s er et primtall (fordi ingen av primtallene $p_1, p_2, p_3, p_4, \ldots\ldots, p_m$) går opp i 's'. Tallet 's' er også større enn p_m.Dette er en selvmotsigelse, fordi vi nå har et nytt primtall $s > p_m$
4. **Altså er påstanden riktig**
   Fordi antagelsen av det motsatte bar galt avsted . (når det motsatte er galt, er det dette er motsatt av, riktig)

$$(\neg T = false) \Leftrightarrow (T = true)$$

Tidsmålinger

Hvor lang tid bruker:

A) **Enkel for-løkke**

```
for (int i = 0; i < n; i++)
    a[i] = a[n-i-1];
```

B) Dobbel for-løkke

```
for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++)
        a[i] = a[n-j-1];
```

(Tid i millisek.)

n=	A) Enkel	B) Dobbel
10	1	1
100	1	1
1 000	1	56
10 000	2	5 856
100 000	13	640 110
1 000 000	134	?

Program: se **hjemmsida til INF2220 for programeksempler/**


```
import java.util.*;

public class Tatid
// både mulig å bruke fra 'main' og via subklasse
{   long tid = 0;

    Tatid(int n)
    { tid = System.currentTimeMillis();
      bruk(n);
      tid =  System.currentTimeMillis() - tid;
      System.out.println("Tid brukt: " + tid + " millisekunder");
    }
     void bruk(int n)
     { // redefineres i subklasse
     } // end bruk

     public static void  main ( String[] args)
     {  if (args.length < 1){ System.out.println(" Bruk: \n >java  TaTid  <n>");
              } else {      int n  = new Integer(args[0]).intValue(); // få parameter fra linja

          Tatid t = new Tatid(n);
       }} // end main
 } // end **** class Tatid *****
```

```
import easyIO.*;
import java.util.*;

public class Lokke extends Tatid
{
    Lokke(int i)
    {   super(i);
     }

     void bruk(int n)
     {   for (int k = 0; k < n; k++)
              for (int j = 0; j < n; j++)
                   ;
     } // end bruk

     public static void  main ( String[] args)
     {  if (args.length < 1){ System.out.println(" Bruk: \n >java  Lokke  <n1> ");
                         } else {
          int n  = new Integer(args[0]).intValue(); // få parameter fra linja

          Lokke  l = new Lokke(n);
       }} // end main

 } // end **** class Lokke *****
```

Sortering

- Hva sorterer vi i praksis
  - Bare tall eller tekster – eller noe mer
- Hvordan definere problemet
  - Krav som må være oppfylt
- Empirisk testing av hastigheten til algoritmer
  - antall
  - Hvilke verdier (fordeling, max og min verdi – gitt antallet)
- Hva avjør tidsforbruket ved sortering
  - Sorteringsalgoritmen
  - N, antall elementer vi sorterer
  - Fordelingen av disse (Uniform, skjeve fordelinger, spredte,..)
  - Effekten av caching

To klasser av sorteringsalgoritmer

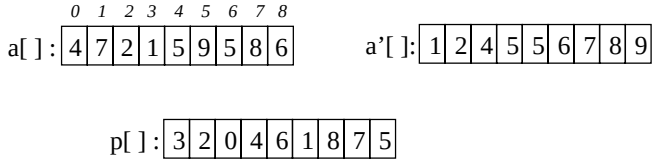
- Sammenligning-baserte:
  Baserer seg på sammenligning av elemntene i a[ ]
  - Instikk, boble
  - Merge, Heap, Shell, Tree
  - Quicksort
- Verdi-baserte :
  Direkte plassering basert på verdien av hvert element – ingen sammenligninger med nabo-elementer e.l.
  - Bøtte
  - Radix
  - PSort

Sorteringsproblemet, definisjon.

- Kaller arrayen a[] før sorteringen og a'[ ] etter
  - og n er lengden:dvs. a = new int [n];.
- Sorteringskravet:
  - $a'[i] \leq a'[i+1]$, i = 0,1,..,n-2
- Stabil sortering
  - Like elementer skal beholde sin innbyrdes rekkefølge etter sortering. Dvs. hvis a[i] = a[j] og i < j, så skal k < r, hvis a[i] er sortert inn på plass 'k' i a'[] og a[j] sortert inn på plass 'r' i a[],
- Sorteringsalgoritmene antar at det kan finnes like verdier i a[ ]
  - I bevisene antar vi alle a_i er forskjellige: $a[i] \neq a[j]$, når $i \neq j$.
- I terstkjøingene antar at innholdet i a[ ] er er en tilfeldig permutasjon av tallene 1..n – andre fordelinger av tallene kan gi helt andre kjøretider.
- Hvor mye ekstre plass bruker algoritmen
  - Et lite fast antall, et begrenset antall (eks. < 10^{12}) heltall, eller n ekstra ord
- **N.B** Ett krav til – hvilket ?

N.B. Husk bevaringskriteriet

- Bevaringskriteriet:
  - alle elementene vi hadde i a[ ], skal være i a'[ ]
  - Formelt : Skal eksistere en permutasjon, p, av tallene 0..n-1 slik at a'[i] = a[p[i]] , , i = 0,1,..n-1
    (kan også defineres 'den andre veien', men mindre nyttig)



- Vi skal senere se en sorteringsalgoritme basert på en slik array p (kallt Psort)

En litt enklere kode enn boka, antar vi sorterer heltall.

- Lar seg lett generalisere til bokas tilfelle, som antar at den sorterer en array av objekter som er av typen Comparable.

BOKA:

```
Comparable [ ] a  = new Comparable [n];

tmp = a[i];

if ( tmp.compareTo( a[j])   < 0 ) {
.....
}
```

HER:

```
int [ ] a = new int [n];

   tmp = a[i],

if ( tmp < a[j] ) {
     ...............
 }
```


tids-forbruk, millisek. – 450MHz PC

Lengde av a: 10

Boble-sort =	0,061
Innstikk-sort =	0,037
Heap - sort =	0,066
Shell-sort =	0,065
Tree - sort =	0,072

Lengde av a: 1 000

Boble-sort =	46,100
Innstikk-sort =	10,900
Heap - sort =	1,700
Shell-sort =	1,700 (2,268 for len=1024)
Tree - sort =	1,600

Lengde av a: 100

Boble-sort =	0,610
Innstikk-sort =	0,220
Heap - sort =	0,270
Shell-sort =	0,170
Tree - sort =	0,160

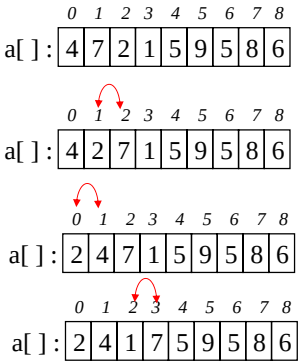
Lengde av a: 10 000

Boble-sort =	4735,000
Innstikk-sort =	1230,000
Heap - sort =	28,000
Shell-sort =	22,000
Tree - sort =	17,000

Boblesort – den aller langsomste !

```
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, max = a.length;

   while ( i < max )
          if (a[i] > a[i+1])
          {    bytt (a, i, i+1);
               if (i > 0 )  i = i-1;
           } else {
              i = i + 1;
           }
  } // end bobleSort
```

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



Theorem 7.1 – antall ombyttinger

En inversjon (’feil’) er per def: a[i] > a[j], men i < j.

Th 7.1

Det er gjennomsnittlig n(n-1)/4 inversjoner i en array av lengde n.

Bevis

Se på en liste L og den samme listen reversert L_r .Ser vi på to vilkårlige elementer x,y i begge disse listene. I en av liste står de opplagt i gal rekkefølge (hvis x ≠y). Det er n(n-1)/2 slike par i de to listene, og i snitt står halvparten ’feil’ sortert, dvs. n(n-1)/4 inversjoner i L.

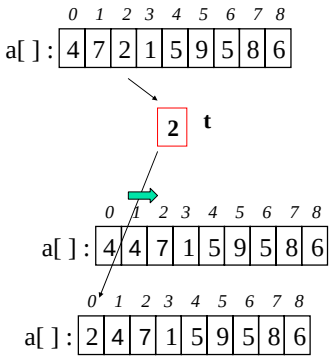
analyse av Boble-sortering

- Boble er opplagt $O(n^2)$ fordi:
  - Th. 7.1 sier at det er $O(n^2)$ inversjoner, og en naboombytting fjerner bare en slik inversjon.
- Kunne også argumentert som flg.:
  - Vi går gjennom hele arrayen, og for hver som er i gal rekkefølge (halvparten i snitt) – bytter vi disse (i snitt) halve arrayen ned mot begynnelsen.
  - n/2 x n/2 = $n^2/4 = O(n^2)$, men mange operasjoner ved å boble (nabo-ombyttinger)

Innstikk-sortering

```
void insertSort(int [] a )
  {int i, t, max = a.length -1;

   for (int k = 0 ; k < max; k++) {
   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
```

***Ide**: 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.*



’Formelt’ bevis av insertSort, Spesifikasjon av Sort

Spesifikasjon_: ***void*** Sort (a,n);
//a’[..] er utdata, *a[..]* er inndata -verdiene

Inndata: n tall a[0..n-1] i vilkårlig rekkefølge
Utdata: $a'[i-1] \le a'[i],\ 0 < i < n$ **og**
$\exists$ permutasjon P: $\forall i\ 0 \le i < n$: $a'[P[i]] = a[i]$

⇐ ***Sortert-kravet***
⇐ ***Bevar innholdet av a[0..n-1]***

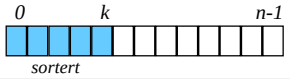
Bevaringskravet leses:

Det finnes en rekkefølge P av tallene 0..n-1, slik at ved å lese utdata *a’[..]*i den rekkefølgen, er den lik inndata *a[..]*

Generell design-metodikk

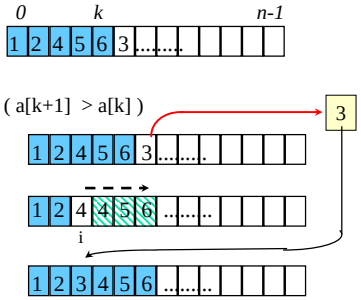
- Gitt en klar spesifikasjon med flere ledd
- En av delene i spesifikasjonen nyttes som invariant i programmets (ytterste) hovedløkke i en litt endret form.
  (løkke-invariant = noe som er sant i begynnelsen av løkka)
- Dette kravet svekkes litt (gjøres litt enklere); gjelder da typisk bare for en del av datastukturen
- I denne hoved-løkka, gjelder så resten av spesifikasjonene:
  - i begynnelsen av hoved-løkka
  - ødelegges ofte i løpet av løkka
  - gjenskapes før avslutning av løkka

Design, innstikksortering



Svekker Sortert-kravet til bare å gjelde a[0..i-1]
Bevaringskravet beholdes (for hele a[0..n-1])

Innstikk-sortering (for k = 0, 1,2,...,n-1:):



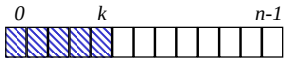
1) Ta ut det 'galt plasserte' elementet a[i]

2) Finn *i* hvor 'gamle a[k+1]' skal plasseres og skyv a[i..k] ett-hakk-til- høyre (*ødelegger Bevaringskravet)*

3) Sett 'gamle a[k+1]' inn på plass ***i*** (*gjenskaper Bevaringskravet*)

Kode - innstikksortering

```
void  insertSort ( int [ ]a )
    {  int i,  t, max = a.length –1;

1       for(int k=0; k < max; k++)
2       if (a[k] > a[k+1] )
3       {     t = a[k+1];
4             i = k;
5             do
6               { a[i+1] = a[i]; i - -;
7               } while ( i >=0 && a[i] > t)
8             a[i+1] = t;
9       }
10    }
```



Her gjelder Sortert: a[0.. k]
og Bevart a[0..n-1]

1) Ta ut det 'galt plasserte elementet a[k+1]

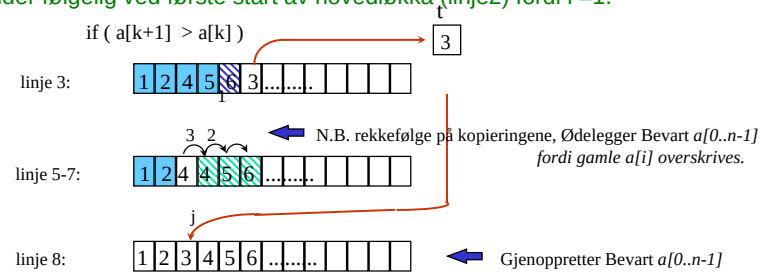
2) Finn *i* hvor 'gamle a[k+1]' skal plasseres og skyv a[i..k] ett-hakk-til- høyre (*ødelegger Bevaringskravet)*

3) Sett 'gamle a[k+1]' inn på plass *i* (*gjenskaper Bevaringskravet*)

Verifiseres iflg. spesifikasjonene + resonnement for *terminering* av prosedyren

argument for at insertSort er riktig

- En array med ett element er sortert - dvs. *a[0]* er sortert.
- Løkke-invarianten (*a[0..k] er sortert* og *Bevart a[0..n-1])* gjelder følgelig ved første start av hovedløkka (linje2) fordi i =1.



- Ett gjenomløp av hovedløkka gjør den sekvensen som er sortert ett element lenger og øker i med 1 - dvs. løkkeinvarianten gjelder på toppen ved neste gjennomløp (*a[0..k] er sortert* og *Bevart a[0..n-1])*.
- Prosedyren terminerer opplagt når k=n-1 og for (..) øker *k* med 1 hver gang.

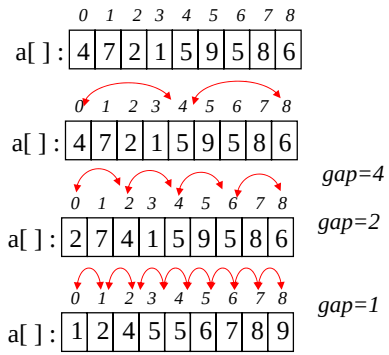
O analyse av innstikk-sortering

- Samme analyse som Boble, men langt færre operasjoner per forflytning – $O(n^2)$

Shell,

```
void ShellSort(int [] a)
 {
   for (int gap = a.length/2 ; gap > 0 ; gap =gap/2)
     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;
          }

} // end ShellSort
```

Ide**: Gjør essensielt innstikksortering langs a[i], a[i-gap] a[i-2gap] ... for gap = n/2, n/4,..,1 og i = gap, gap+1 ,..,n-1. Dvs. alle sekvenser i a[] av lengde ..n/2, n/4, ...,2 **og til sist 1



analyse av Shell-sortering

- Hvorfor virker den – hvorfor sorterer den ?
  - Fordi når gap = 1 er dette innstikksortering av arrayen
- Hvorfor er dette vanligvis raskere enn innstikksortering
  - Fordi vi på en 'billig' måte har nesten sortert a[ ] før siste gjennomgang med gap=1, og når a[ ] er delvis sortert, blir innstiksortering meget rask.
- Worst case, som 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
- Husk: En slik sekvens begynner på 1

tider i millisek

Lengde av a: 100 000

Heap - sort = 330

Shell-sort = 390

Tree - sort = 270

Lengde av a: 1 000 000

Heap - sort = 4290

Shell-sort = 5500

Tree - sort = 4070

Lengde av a: 1 048 576 = 220**

Heap - sort = 4535

Shell-sort = 35700

Tree - sort = 4230

Hvorfor er Shell-sort så dårlig når n = 2^k ?

Shell2 – en annensekvens 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;
                                }
            }
} // end
```

tider i millisek
Shell = originale 'gap' = 1,2, ,n/8, n/4, n/2–
Shell2 med 'gap'= 1,2,5,11,..n/11, n/5, n/2

Lengde av a:	**100 000**
Heap - sort =	**209**
Shell-sort =	**253**
Shell 2 -sort =	**185**
Tree - sort =	**189**

Lengde av a:	**1048576 = 2 ** 20**
Heap - sort =	**3 235**
Shell-sort =	**33 281 !!**
Shell 2 -sort =	**2 938**
Tree - sort =	**3 078**

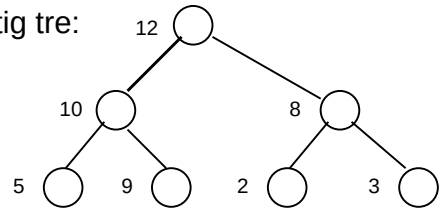
Lengde av a:	**1 000 000**
Heap - sort =	**3 079**
Shell-sort =	**4 032**
Shell 2 -sort =	**2 750**
Tree - sort =	**2 875**

Lengde av a:	8 192 = 2**13
Heap - sort =	13
Shell-sort =	22
Shell 2 -sort =	11
Tree - sort =	11

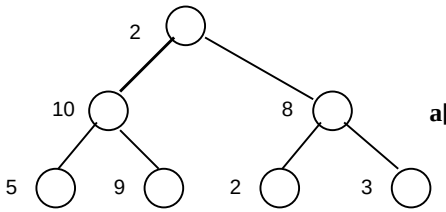
Rotrettet tre

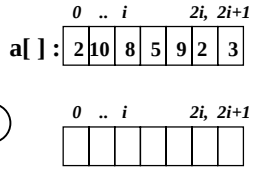
- Ide for Heap &Tre sortering – rotrettet tre i arrayen:
  1. Rota er største element i treet (også i rota i alle subtrær – rekursivt)
  2. Det er ingen ordning mellom vsub og hsub (hvem som er størst)
  3. Vi betrakter innholdet av en array a[0:n-1] slik at vsub og hsub til element 'i' er i: '2i+1' og '2i+2' (Hvis vi ikke går ut over arrayen)

Eks på riktig tre:

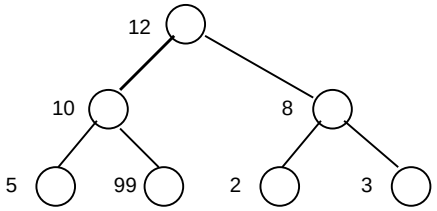


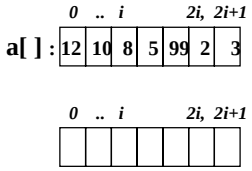
Feil i rota, ’2’ er ikke størst:





Feil i bladnode, ’99 ’ er større enn sin rot:

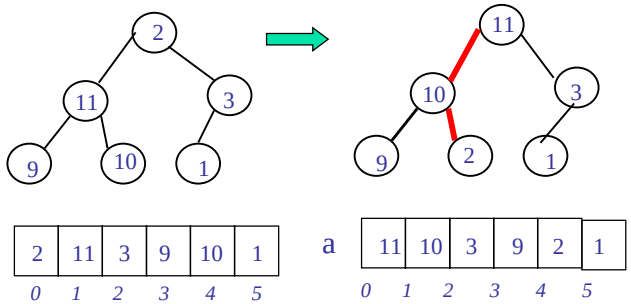




Hjelpemetode 1 – roten i et (sub)tre muligens feil :

```
static 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;
 }
```

Før: *dyttNed (0, 5)* *Etter:*



Eksekveringstider for dyttNed

```
static void dyttNed (int i, int n)
 { 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;
 }
```

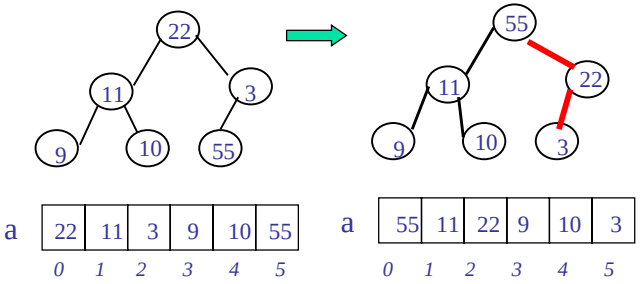
Vi ser at metoden starter på subtreet med rot i a[i] og i verste tilfelle må flytte det elementet helt til ned til en bladnode – ca. til a[n] ,

Avstanden er (n-i) i arrayen og hver gang **dobler** vi j inntil j <= n:
dvs. while-løkka går maks. log(n-i) ganger = **O(log n)**

(dette er det samme som at høyden i et binærtre er log(n))

```
static void dyttOpp (int i)
// Bladnoden på plass i er (muligens) feilplassert
// dytt den oppover mot rota til hele treet
{ 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;
 }
```

Før: *dyttOpp (5)* *Etter:*



Eksekveringstiden for dyttOpp

```
static void dyttOpp (int i)
// Bladnoden på plass i er (muligens) feilplassert
// dytt den oppover mot rota til hele treet
{ 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;
 }
```

Vi ser at metoden starter på det treet med rot i a[0] som går til og med a[i], og i verste må flytte gamle a[i] helt opp til rota a[0]

Avstanden er 'i+1' i arrayen og hver gang **halverer** vi i inntil i verste fall i = 0: dvs. while-løkka går maks. log(i+1) ganger = **O(log n)** fordi 'i' maksimalt er lik 'n' og gjennomsnitlig 'n/2'

(dette er også et samme som at høyden i et binærtre er log(n))

Ideen bak Tre & 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]

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);
   }
 }
```

***Ide**: Vi har et binært ordningstre i a[0..k] med største i rota. Ordne først alle subtrær..Få største element opp i a[0] og Bytt det med det k'te elementet (k= n, n-1,..)*

	0	1	2	3	4	5	6	7	8
a[ ] :	4	7	2	1	5	9	5	8	6

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(n log n)
  - Først ordner vi n/2 subtrær med gjennomstitshøyde = (log n) / 2 = n*logn/4
  - Så setter vi inn en ny node 'n' ganger i toppen av det treet som er i a[0..k], k = n, n-1,..,2,1
    I snitt er høyden på dette treet (nesten) log n – dvs n logn
  - Summen er klart O(n logn)

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

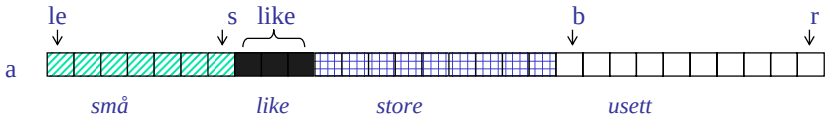
- Som Tre-sortering: Vi jobber med binære trær (hauger) , og 'innsetter' i prinsippet n verdier, alle med vei log $_2$ til rota = O(n log n)

Quicksort – generell idé

1. Finn ett element i (den delen av) arrayen du skal sortere som er omtrent 'middels stort' blant disse elementene – kall det ***'part'***

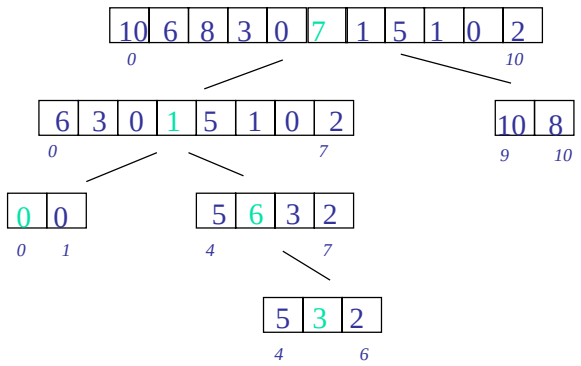
2. Del opp arrayen i tre deler og flytt elementer slik at:
   a) *små* - de som er mindre enn 'part' er til venstre
   b) *like* - de som har samme verdi som 'part' er i midten
   c) *store* - de som er større, til høyre

3. Gjennta pkt. 1 og 2 rekursivt for de *små* og *store* områdene hver for seg inntil lengden av dem er < 2, og dermed sortert.




```
void  quickSort ( int [] a, int le, int r)
{ int s = le-1, like = 0, ind;
  int t, part = a[(le+r)/2];

 for ( int b = le; b <= r; b++)
   if ( a[b] = = part )
   {    like++;              // bytt om den venstre store
        t = a[s+like];       // og den nye (a[b])
       a[s+like] = a[b];
       a[b] = t;
   } else
   if (a[b] < part) {
       s++;                 // bytt om syklisk den venstre
        ind = s+like;       // store , den venstre like og
        t = a[b];           // den nye a [b]
        a[b] = a[ind];
        a[ind]= a[s];
        a[s] = t;
    }
  if ( le < s )             quickSort (a,le,s);
  if ( s+1+like < r )  quickSort (a,s+1+like,r);
}
```

QuickSort - eksempel



Sortert :

0	0	1	1	2	3	5	6	7	8	10

Quick – sort, 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 (n log n)

(log n ledd i addisjonen)

Gjennomsnitt

I praksis vil verste tilfellet ikke opptre – men velger ofte 'part' som medianen av a[l], a[(l+r)/2 og a[r] og vi får O (n log n)

Quicksort i praksis I

- Bruker en annen implementasjon enn den som er vist tidligere (med færre ombyttinger)
- Kaller 'innstikkSort' når lengden av det som skal sorteres er mindre enn ca. 10

En slik QuickSort går ca dobbelt så fort som den som er demonstrert tidligere (men vanskelig å få riktig):

```
void  quickSort ( int [] a,int l,int r)
 { int i=l, j=r;
   int t, part = a[(l+r)/2];

    while ( i <= j)
    { while (a[i] < part ) i++;
      while (part < a[j] ) j--;

      if (i <= j)
      {   t    = a[j];
          a[j]= a[i];
          a[i]= t;
          i++;
           j--;
      }
    }
```

```
if ( l < j ) {
      if ( j-l < 10) innstikkSort (a,l,j);
                else quicksort (a,l,j);}
     if ( i < r ) {
           if ( r-i < 10) innstikkSort (a,i,r);
            else quicksort (a,i,r); }
   }

} // end quickSort
```

Quicksort i praksis II

- Valg av pertisjoneringselement 'part' er vesentlig
- Bokas versjon av Quicksort OK, men flytter partisjonerings elementet ut på sidelinje, og tar ikke høyde for flere like elementer
- Velger derfor ofte medianen (det midterste i verdi) av:
  - det første
  - det midterste
  - det siste

  elementet i det området vi skal sortere
- Quicksort er ikke den raskeste algoritmen (f.eks er Radix minst dobbelt så rask), men nyttes mye – f.eks i java.util.Arrays.sort();

Finne det k' største elementet

Generell idé:

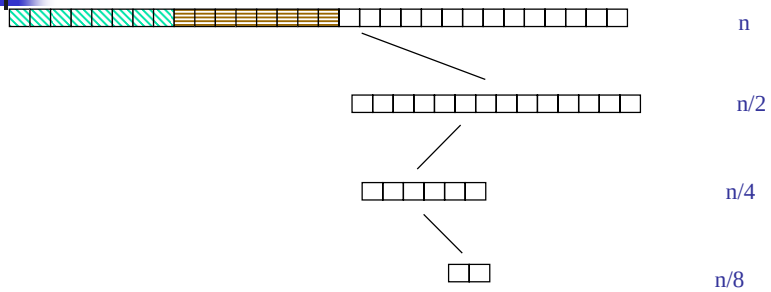
1. Bruk oppdelingsmetoden fra QuickSort og del 'a' opp i 'liten' 'like' og 'stor' del
2. Let videre (rekursivt) i riktig del:

if (k <= (lengden av 'liten')) let i 'liten' **else**
if (k <= (lengden av lite + like) funnet i 'like' **else**
let videre i 'stor'

Eksekveringstid - se QuickSort (men hvordan ?)


```
int  kvikkValg ( int [] a,int l, int r, int k)
// deler  a[l,r] i 'liten' , 'lik' og 'stor'
// velger ut det k-største elementet i a  (k = 1,..., a.length)
{ int s = l-1, like = 0, ind;
  int t, part = a[(l+r)/2];

  if (l == r) return a[r];
  else {
    for ( int b = l; b <= r; b++)
    if ( a[b] == part )
    {  like++;
       t = a[s+like];
       a[s+like] = a[b];
       a[b] = t;
    } else
    if (a[b] < part)
    {  s++;
       ind = s+like;
       t = a[b];
       a[b] = a[ind];
       a[ind]= a[s];
       a[s] = t;
     }

     if     ( k -1<= s )   return  kvikkValg (a,l,s,k);
     else if ( k -1<= s + like) return part;
     else  return  kvikkValg (a,s+1+like,r,k );
   }
}
```

Eksekveringstid - kvikkValg



dvs: n + n/2 + n/4 + n/8 + ...+1
= n + n (½ + ¼ +1/8.....) = 2 n = O(n)

Flette - sortering (merge)

Velegnet for sortering av filer.

Generell idé:

1. Vi har to sorterte sekvenser 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.

I praksis skal det meget store filer til, før du bruker flette-sortering. 1024 MB 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

skisse av Flette-kode

```
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.first;}
     else        {  C.wite (b); b = B.first;}

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

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

}
```


Verdi-baserte sorteringsmetoder

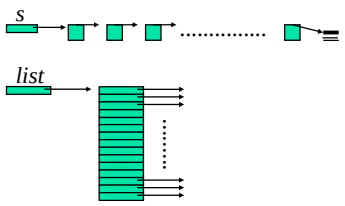
- Direkte plassering basert på verdien av hvert element – ingen sammenligninger med nabo-elementer e.l.
- Telle-sortering, en metode som ***ikke*** er brukbar i praksis (hvorfor ?)
- Er klart av O(n) , men 'svindel':

```
void telleSort(int [] a) {
  int max = 0, i,m, ind = 0;
  for (i = 1 ; i < n; i++)   if (a[i] > max) max = a[i];

  int [] telle = new int[max+1];

  for(  i = 0; i < n; i++)  telle[a[i]] ++;

   for(  i = 0; i <= max; i++) {
         m = telle[i];
         while ( m > 0 ) {
           a[ind++] = i;
           m--;
}      }
```


Bøtte-sortering og sortering av objekter

- Inndata: Usortert liste s, med n objekter hver med en (int) nøkkel 'value'
- Sortering via direkte innplassering i n stk. Stakker (= LIFO –kø)
- Utdata: Sortert liste s
- Stabil sortering
- Kjøretid O(max + n), max er største verdi i nøkkelen = O(n) når max er 'omlag lik' n (ikke spredd, tynn fordeling.) , men gjerne mange like elementer.
- Ulemper:
  - Bruker ekstra plass: 2*n pekere (liste + neste-pekere i objektene) + overhead av objekter, tilsammen = **ca. 3 * n**
  - Antar uniform eller tett fordeling.
- Brukes bl.a av FAST søkemotor


```
class BNode
  { BNode neste; int verdi;
    BNode (BNode n, int v)
    { neste= n; verdi= v; }
  }
..
 for(int i = 0; i<n; i++)
    s = new BNode( s, a[i]);

 s = bucketSort( s );
...
```



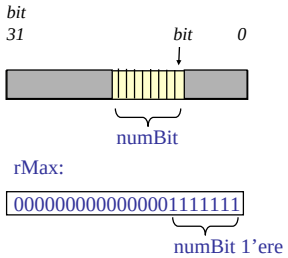

```
BNode bucketSort ( BNode s )
  {int max =0;
   BNode t =s;
   while (t!= null) {
      if (t.value >max ) max = t.value;
      t = t.next;
   }
   BNode [] list = new BNode [max+1];
   BNode t;
    while (s != null) {
      t = s;
      s = s.neste;
      t.neste = list[t.verdi];
      list[t.verdi] = t;
    }
   // lag liste FIFO from LIFO +LIFO
   for (int i = max; i >= 0; i--)
    while (list[i] != null ) {
      t = list[i];
      list[i] = t.neste;
      t.neste = s;
      s = t;
    }
    return s;
  }
```

Radix-sortering

- Sorterer en array a [] på hvert siffer
- Et siffer et 'bare' ett visst antall bit
- Algoritme:
  - Finner først max verdi i a [ ]
  - Kopierer data ved hver slik gjennomgang fra en array (i utgangspunktet a []) til en annen, b[] av samme lengde
  - Neste gang (for neste siffer) kopieres tilbake fra b[] til a[]


```
// Radix – konstanter og kode n < 1024 elementer
int [] b;
static int numBit = 10, rMax = 1024-1, max =0;

void radixSort(int [] a )
 { //  finn max i a[]
    for (int i = 0 ; i < a.length; i++)
       if (a[i] > max) max = a[i];

     if ( max < rMax)
     { // for første gang, hvis små/få verdier
       // Trenger nå bare ett siffer som også er mindre
       while ( (1<<numBit)  >  max )  numBit --;
           numBit ++; // went one too far
       rMax = (1<< numBit) -1;
     }

     b = radixSort( a,b, 0, max);

     // kopier tilbake til a hvis svaret nå er i b []
     if ( a != b)
       for (int i = 0; i < n; i++)
        a[i] = b[i];
 }
```

Ett element i fra[]:




```
int [] radixSort ( int [] fra, int [] til , int bit, int max )
 {  int [] ant = new int [rMax+1];
    int  acumVal = 0, j;

     // tell opp i ant hvor mange av hver verdi
     for (int i = 0; i < n; i++)
        ant[((fra[i]>> bit) & rMax)]++;

    // Adder opp i 'ant' – akkumulerte verdier
    for (int i = 0; i <= rMax; i++)  {
       j = ant[i];
       ant[i] = acumVal;
       acumVal += j;
    }
     // flytt  tallene i sortert (på dette feltet) til til.
    for (int i = 0; i < n; i++)
       til [ ant [ ((fra[i]>>bit) & rMax) ]++  ] = fra[i];

    // Hvis mer igjen å sortere – sorter på neste 'bit' biter
    if ( (1 << (bit + numBit)) < max )
        return  radixSort ( til, fra, bit + numBit, max);
    else return  til;
   }
```

Tidsforbruket til Radix

1. Først går vi gjennom a[] for å finne max = O(n)
2. Legg merke til at ant[] har lengde rMax = 1023 – en konstant
3. Følgende operasjoner gjøres $\lceil (\log max)/(\log rMax) \rceil$ ganger:
   1. Tell opp i ant[] hvor mange det er av hver siffer-verdi = O(n)
   2. Juster pekere i ant[] = O(log rMax)
   3. Flytte alle n data (fra: fra[] til: til[]) = O(n)
4. Dvs totalt O(n log max), og siden max ofte er en funksjon av n, blir dette ofte O(n logn) – men koeffisienten er meget liten
5. Radix er vel dobbelt så rask som Quicksort i praksis, men krever dobbelt så stor plass
6. Hovegrunnen til at Radix er så rask at den aksesserer hvert element langt færre ganger enn Quicksort (når n = 1 mill, gjør Quicksort ca. 20 les/skriv per element, Radix gjør ca. 7)

Sortere ved å lage sorterings-permutasjonen
Psort I

- Hvorfor sortere ?
  - Ikke egentlig interessert i å sortere en **int array**
  - Man sorterer sammenhengende datamengder:
    - Bank : Konto #., navn, adresse, saldo sortert på Konto #.
    - Student: Navn, adresse, institutt #, eksamen, sortert på Inst # og navn
- Antar at data er i **a[ ],b[ ],...,d[ ]**
  og at vi vil presentere **a, b,...d** sortert på **a**
- Tre løsninger:
  1. Flytte data i b,c,...d sammen med og tilsvarende man sorterer **a**
     - ganske langsomt !
  2. Legg data(a_i,b_i,...,d_i) i objekt$_i$. Sorter objektene på nøkkelen a_i
     - raskt, men plassforbrukende (og derfor langsomt for store datasett)
  3. Generere sorteringspermutasjonen – **int [ ] p** fra a – slik at:
     - a[p[i]], b[p[i]], ...,d[p[i]] er det sorterte datasettet.

Psort, lager sorterings-permutasjonen

1. (som Radix: Finn max verdi i a).
2. Som Radix: Tell i array count hvor mange det er av hver verdi i a.
3. Som Radix: Adder antallene til logiske pekere i count
   count '[i] = count [i-1] + count [i-2]
4. Lag sorterings- permutasjonen: **p[count[a[i]]++] = i.**

Eks: **a**

0	1	2	3	4	5
3	3	0	5	1	6

1. max = 6

2 . Count

0	1
1	1
2	0
3	2
4	0
5	1
6	1

3 Adder til pekere

0	0
1	1
2	2
3	2
4	4
5	4
6	5

4. Lag p :

0	2
1	4
2	0
3	1
4	3
5	5

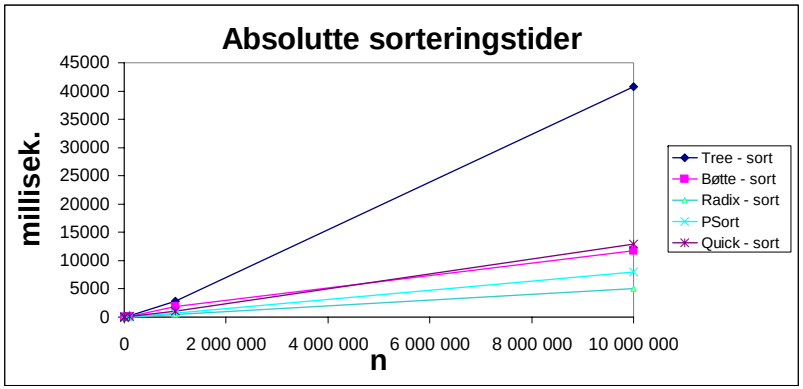

```
int [] psort ( int [] a )
 { int n= a.length;
   int [] p = new int [n];
   int [] count ;
   int localMax = 0;
   int accumVal = 0, j,

   for (int i = 0 ; i < n ; i++)
         if( localMax < a[i]) localMax = a[i];

   count = new int[localMax++];

   for (int i = 0; i < n; i++)
          count[a[i]]++;

   for (int i = 0; i < localMax; i++) {
       j = count[i];
       count[i] = accumVal;
       accumVal += j;
    }

    for (int i = 0; i < n; i++)
         p[count[a[i]]++] = i;

    return p;
 }
```

Sammenligning av algoritmer

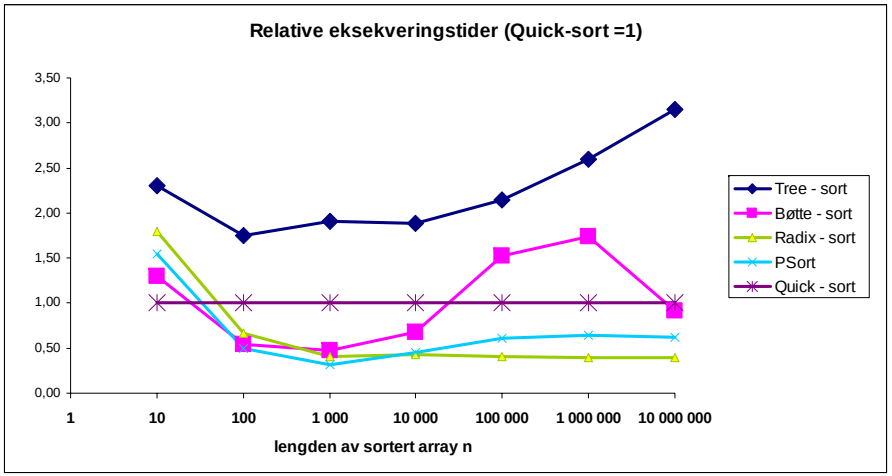
- Tree-sort
- Quick-sort
  - effektivisert med Innstikksortering for subdeler < 10
- Bøtte-sortering
  - Data plassert i objekter. Disse plasseres i lister – en per mulig verdi – generering av objekter utenfor tidtagingen
- P-sort
- Radix
  - Minst signifikant siffer først, 2 pass (hvis nødvendig)
  - 10 bit fast siffer

Noen tider

n	10000000	1000000	100000	10000	1000	100	10
Tree - sort	40765	2875	187,5	12,97	0,954	0,0657	0,0046
Bøtte - sort	11781	1922	132,8	4,68	0,235	0,0203	0,0026
Radix - sort	5093	438	36	2,98	0,203	0,025	0,0036
PSort	8046	719	53,1	3,12	0,156	0,0188	0,0031
Quick - sort	12938	1109	87,4	6,88	0,501	0,0375	0,002

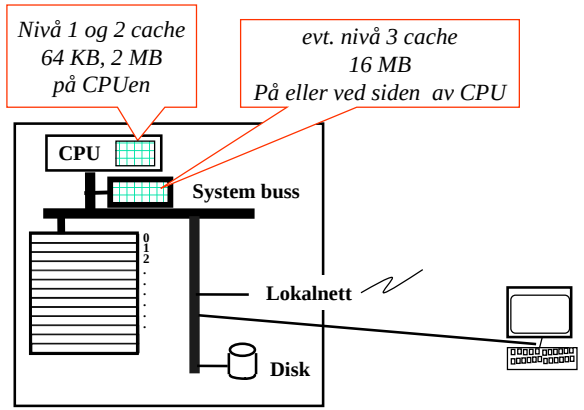


Relativt til Quicksort (logaritmisk x-akse)



Hva er caching ?

Når vi ikke finner data i nærmeste cache, overføres en cachelinje = 32/64 Byte fra saktere til raskere hukommelse.



Effekten av caching – hvor stor ?

- Ser på uttrykk som nyttes i Radix og Psort
- Radix:

```
// flytt  tallene
for (int i = 0; i < n; i++)
   til[ant[((fra[i]>>bit) & rMax)]++] = fra[i];
```

- Psort

```
// make p[]
for (int i = 0; i < n; i++)
   p[count[a[i]]++] = i;
```

- Disse gjennomløpene er ganske cache-uvennlige
  - **De 'hopper' relativt tilfeldig rundt i 'count' og 'p'**

Cache-test

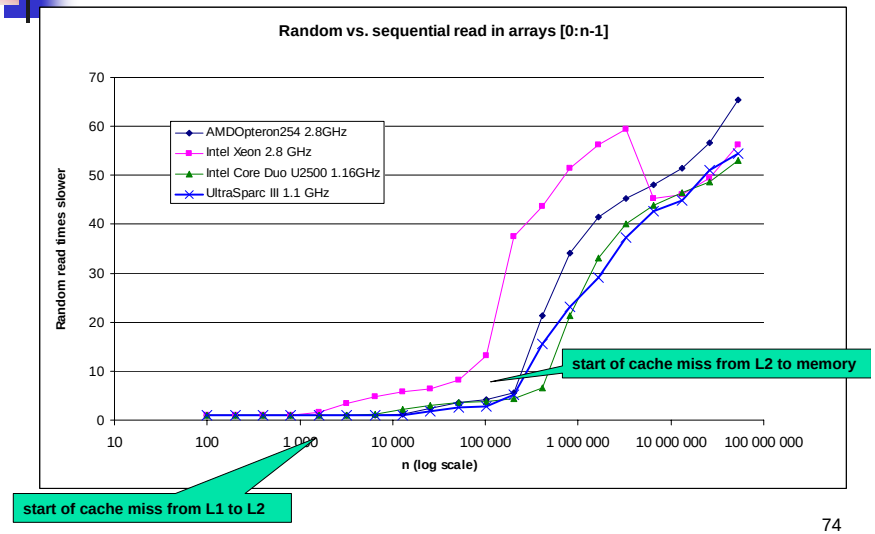
Tester uttrykk av typen med 1 skriv og k les:

```
for(int i= 0; i < n; i++)
        a[b[b[…b[i]…]]] = i ;
```

To innhold i p:

- Sekvensiell: b[i] = i
  - gir neste ingen cache-feil
- Tilfeldig innhold b[i] = tilfeldig int (0..n-1)
  - gir 'nesten' k stk. cache-feil ved økende n, først nivå 1 cache feil, så også nivå 2, og evt. også til nivå 3.

Random vs. sequential access times, the same number of instructions performed. Cache-misses slowing random access down a factor: 50 – 60 (4 CPUs)



Caching – konklusjoner

- Opp til 70x forsinkelse med 'ekstreme' cache-feil
- Opp til over 5x forsinkelse ved normaltilfellet:
  1 skriv 2-3 les

- Vi kan kanskje ha 2-3x lenger kode (= utførte instruksjoner)
  enn en cache-uvennlig kode og ***enda være raskere***
- Test Radix-sortering med 1, 2 og 3 sifre

UNIVERSITY OF OSLO

Applying the model to Radix sorting – the test

- Three Radix algorithms
  - radix1, sorting the array in **one** pass with one 'large' digit
  - radix2, sorting the array in **two** passes with two half sized digits
  - radix3, sorting the array in **three** passes with three 'small' digits
- radix3 performs almost three times as many instructions as radix1
  - should be almost 3 times as slow as radix1?
- radix2 performs almost twice as many instructions as radix1
  - should be almost 2 times as slow as radix1?

OMS 2007

Base:

Right Radix sorting algorithm :

One pass of array a with one sorting digit of width: maskLen (shifted shift bits up)

```
static void radixSort
( int [] a, int [] b ,int left, int right, int maskLen, int shift) {

    int  acumVal = 0, j, n = right-left+1;
    int mask = (1<<maskLen) -1;
    int [] count = new int [mask+1];

    // a) count=the frequency of each radix value in a
    for (int i = left; i <=right; i++)
      count[(a[i]>> shift) & mask]++;

    // b) Add up in 'count' - accumulated values
    for (int i = 0; i <= mask; i++) {
        j = count[i];
        count[i] = acumVal;
        acumVal += j;
     }

     // c) move numbers in sorted order a to b
    for (int i = 0; i < n; i++)
      b[count[(a[i+left]>>shift) & mask]++] =  a[i+left];

     // d) copy back b to a
    for (int i = 0; i < n; i++)
      a[i+left] = b[i] ;
   }
```


Radix sort with 1, 2 and 3 digits = 1,2 and 3 passes

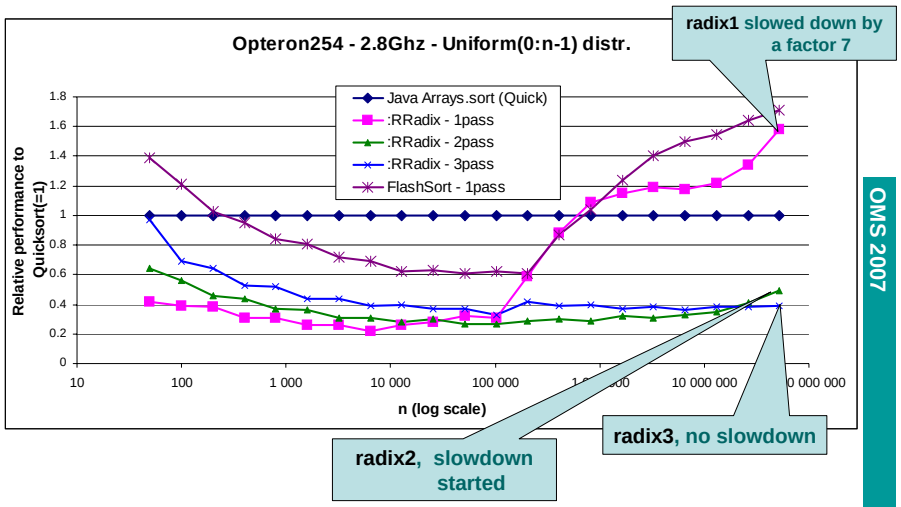
```
static void radix1 (int [] a, int left, int right) {
    // 1 digit radixSort: a[left..right]
    int max = 0, numBit = 1, n = right-left+1;

    for (int i = left ; i <= right ; i++)
        if (a[i] > max) max = a[i];

    while (max >= (1<<numBit)) numBit++;

    int [] b = new int [n];

    radixSort( a,b, left, right, numBit, 0);
}
```

1

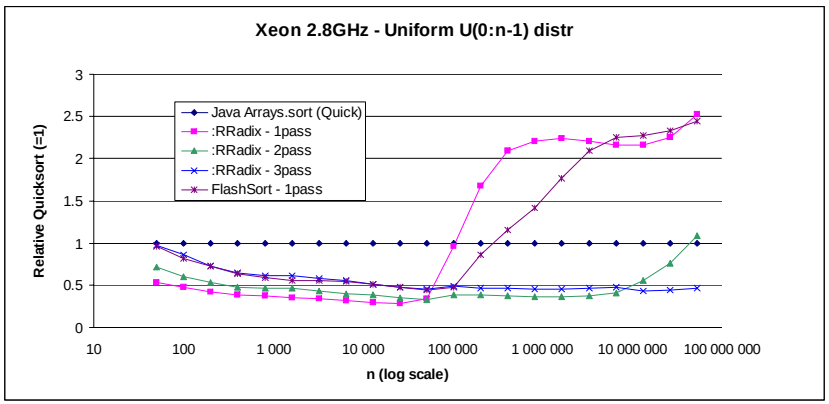
3

```
static void radix3(int [] a, int left, int right) {
    // 3 digit radixSort: a[left..right]
    int max = 0,numBit = 3, n = right-left+1;

    for (int i = left ; i <= right ; i++)
        if (a[i] > max) max = a[i];

    while (max >= (1<<numBit)) numBit++;

    int bit1 = numBit/3,
        bit2 = bit1,
        bit3 = numBit-(bit1+bit2);

    int [] b = new int [n];

    radixSort( a,b, left, right, bit1, 0);
    radixSort( a,b, left, right, bit2, bit1);
    radixSort( a,b, left, right, bit3, bit1+bit2);
  }
```


Random /sequential test (AMD Opteron) , Radix 1, 2 and 3 compared withQuicksort and Flashsort



Random /sequential test (Intel Xeon) , Radix 1, 2 and 3 compared withQuicksort and Flashsort



Conclusions

- The effects of cache-misses are real and show up in ordinary user algorithms when doing random access in large arrays.
- We have demonstrated that radix3, that performs almost 3 times as many instructions as radix1, is 4-5 times as fast as radix1 for large n.
- i.e. radix1 experiences a slowdown of factor 7-10 because of cache-misses

1. *The number of instructions executed is no longer a good measure for the performance of an algorithm.*
2. *Algorithms should be rewritten such that **random access in large** data structures is removed.*