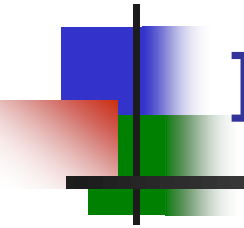


INF2220 – høsten 2016, 9. nov.

Sortering del I (kap. 7.)

Arne Maus,
Gruppen for Programmering og Software Engineering (PSE)
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 – 2015/2016?

- **627 sek = 10,5 min (optimalisert Boblesortering)**
- **95 sek = 1, 5 min (innstikksortering)**
- **0,073 sek. (Quick-sort)**
- **0,013 sek. (radix-sort)**
- **0,005 sek (parallell radix)**

Tidsmålinger – (1992 – 450 MHz)

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

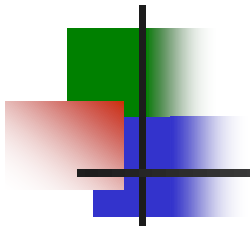
```
for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++)
        a[i] = a[n-j-1];
```

(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	?

```
import java.util.*;

class Tatid {
    long tid = 0;

    Tatid( int n) {
       Random r = new Random (1357);
       int kmax =7;
       int [] a = new int [n];

       for (int k =0;k<kmax;k++) {
            for (int i = 0; i < n; i++) {
                a[i] = r.nextInt();
             }
             tid = System.nanoTime();
             bruk(a);
             tid =  System.nanoTime() - tid;
             System.out.println(metode + "-sortering, n: " + n +", på: "
                       +( tid/1000000.0) + " millisek, k:"+k);
         }
    } // end kontruktor

     void bruk(int [] a) {
       // redefineres i subklasse
     } // end bruk

  } // end **** class Tatid *****
```

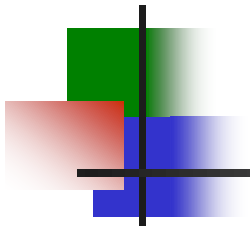
```
import java.util.*;

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: >java  Lokke  <n> ");
         } 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 – er antallet gitt)
- 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
    - Når lønner parallell løsning seg?

To klasser av sorteringsalgoritmer



- Sammenligning-baserte:
  Baserer seg på sammenligning av to elementer i a[ ]
  - Innstikk, boble
  - Tree
  - Quicksort

- Verdi-baserte :
  Direkte plassering basert på verdien av hvert element – ingen sammenligninger med nabo-elementer e.l.
  - Bøtte
  - Radix

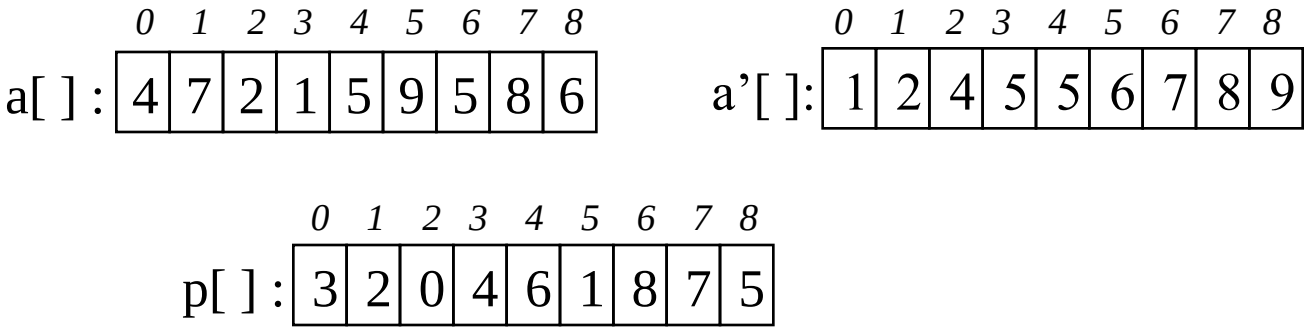
Sorteringsproblemet, definisjon.

- Kaller arrayen a[] før sorteringen og a'[ ] etter
  - og n er lengden av a: 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, og hvis a[i] er sortert inn på plass 'k' i a'[] og a[j] sortert inn på plass 'r' i a[], så skal k < r.
- 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 terstkjøringene 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
  - 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)



  - Hvis inndata er 1, 1, 0 så skal sortert rekkefølge være 0, 1, 1 (IKKE 0, 0, 1), men kanskje 0, 1 , 1 (dvs. ustabil)

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] ) {

                ................
          }
```

tider, millisek. – 2,7 GHz PC – ca 2013

n=	**100**
Boble-sort =	0,56
Innstikk-sort =	0,20
Tree - sort =	0,05
Quick-sort =	0,05

n =	**10 000**
Boble-sort =	57,7
Innstikk-sort =	10,7
Tree - sort =	1,68
Quick-sort t =	1,00

n =	**1 000 000**
Boble-sort =	627 800,
Innstikk-sort =	130 800,
Tree - sort =	130,8
Quick-sort =	97,3

n=	**1000**
Boble-sort =	0,67
Innstikk-sort =	0,89
Tree - sort =	0,17
Quick-sort =	0,10

n=	**100 000**
Boble-sort =	6 237,
Innstikk-sort =	1 064,
Tree - sort =	9,7
Quick-sort =	8,7

n =	**10 000 000**
Boble-sort =	---?? ----,
Innstikk-sort =	---?? ----,
Tree - sort =	2301
Quick-sort =	1136

Effekten av JIT-kompilering

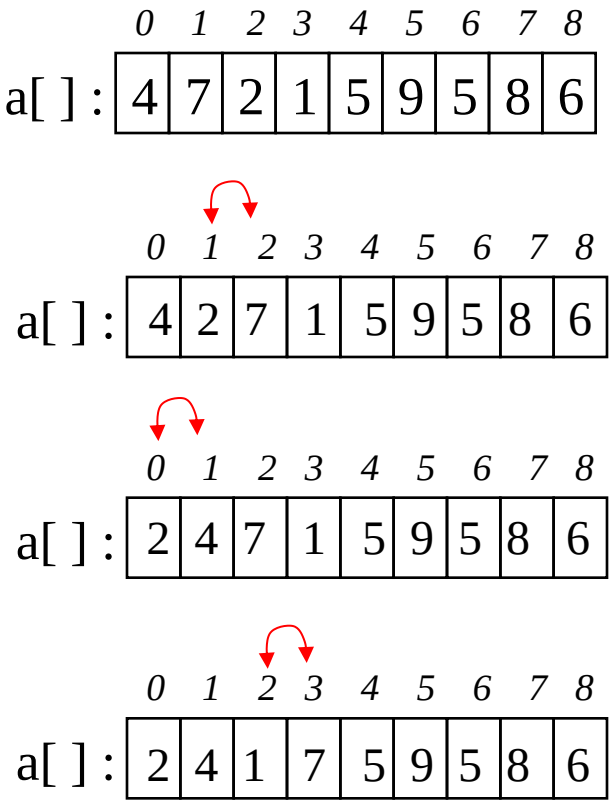
- Programmet optimaliseres (omkopmileres) under kjøring flere ganger av optimalisatoren i JVM (java) – stadig raskere:
  - Første gang en metode kjøres, oversettes den fra Byte-kode til maskinkode
  - Blir den brukt flere/mange ganger optimaliseres denne maskinkoden i en eller flere steg (minst 2 steg)
  - Denne prosessen kasse JIT (Just In Time)-kompilering
  - God idé: kode som brukes mye skal gå raskest mulig

```
Quick-sortering, n: 100 000, på: 56.645581 millisek,
Quick-sortering, n: 100 000, på: 14.32105 millisek,
Quick-sortering, n: 100 000, på:   9.64278 millisek,
Quick-sortering, n: 100 000, på:   8.456771 millisek
Quick-sortering, n: 100 000, på:   8.738068 millisek
```

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;

    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
```

***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 $x \neq y$). 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.

Dette er da en nedre grense for algoritmer som bruker naboombyttinger – de har alle kjøretid $O(n^2)$

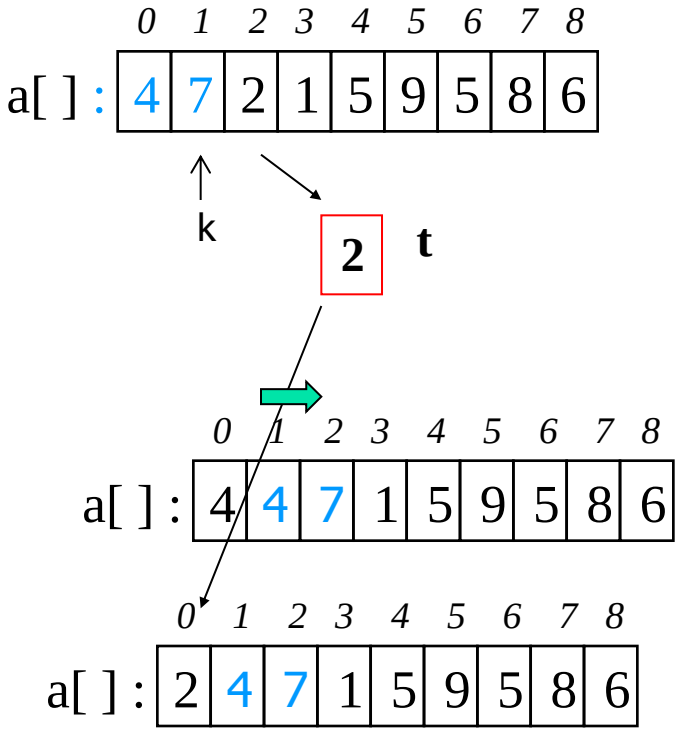
analyse av Boble-sortering

- 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 om disse (i snitt) halve arrayen ned mot begynnelsen.
  - $n/2 \times n/2 = n^2/4 = O(n^2)$, men mange operasjoner ved å boble (nabo-ombyttinger)

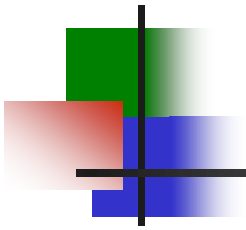
Innstikk-sortering – likevel best for n < 50

```
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
```

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.*

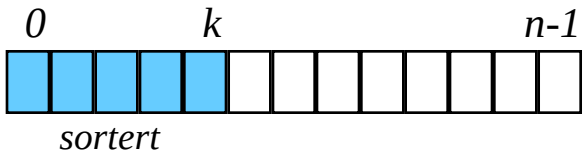


Generell design-metodikk



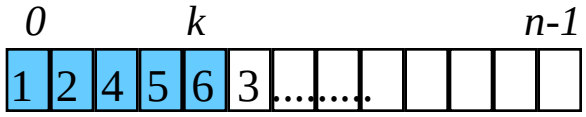
- Gitt en klar spesifikasjon med flere ledd
- En av delene i spesifikasjonen nyttes som invariant i programmets (ytterste) hovedløkke i en litt endret form.
  (løkke-invariant = noe som er sant i begynnelsen av løkka)
- Dette kravet svekkes litt (gjøres litt enklere); gjelder da typisk bare for en del 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

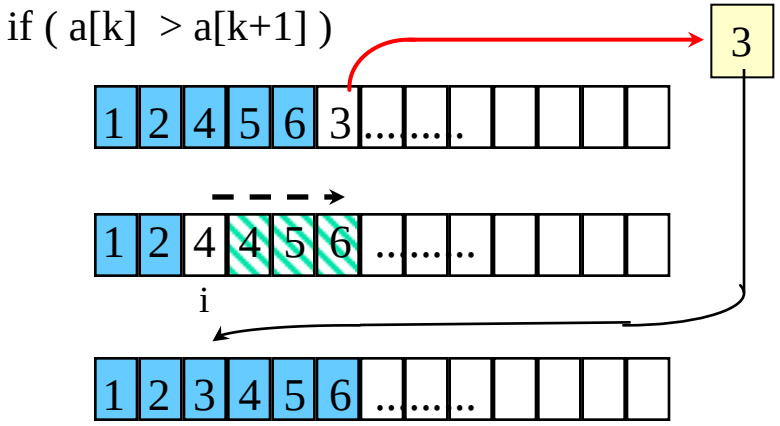
Design, innstikksortering



Svekker Sortert-kravet til bare å gjelde a[0..k-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]

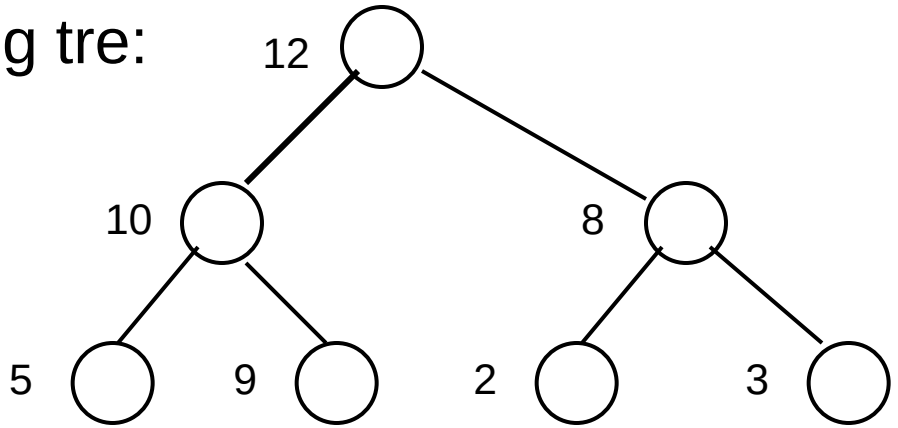
2) Finn *i* hvor 'gamle a[k+1]' skal plasseres og skyv a[i..k] ett-hakk-til- høyre (*ødelegger Bevaringskravet)*

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

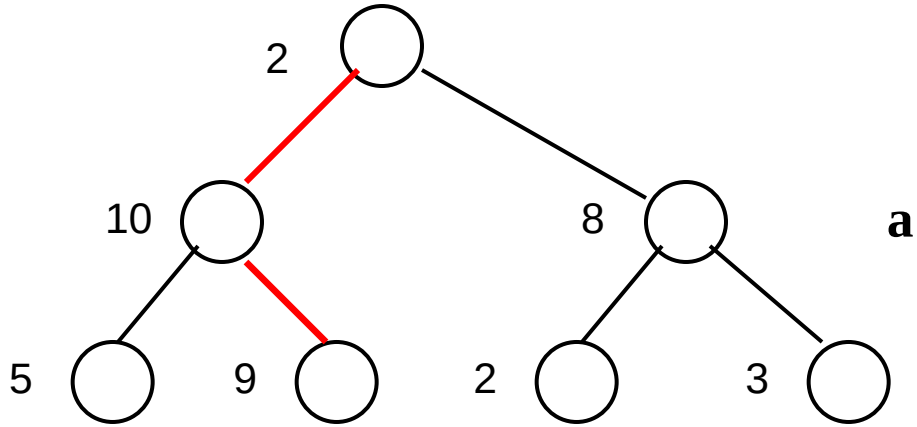
Rotrettet tre (heap)

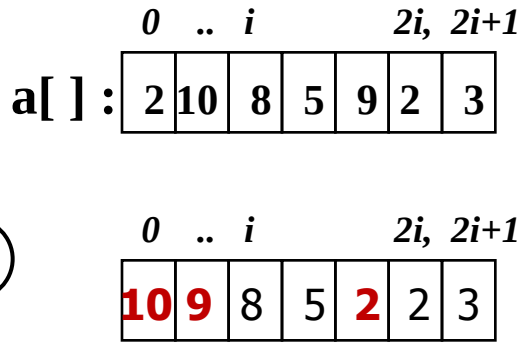
- Idé for (Heap &) Tre sortering – rotrettet tre i arrayen:
  1. Rota er største element i treet (også i rota i alle subtrær – rekursivt)
  2. Det er ingen ordning mellom vsub og hsub (hvem som er størst)
  3. Vi betrakter innholdet av en array a[0:n-1] slik at vsub og hsub til element ’i’ er i: ’2i+1’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

Eks på riktig tre:



Feil i rota, ’2’ er ikke størst:

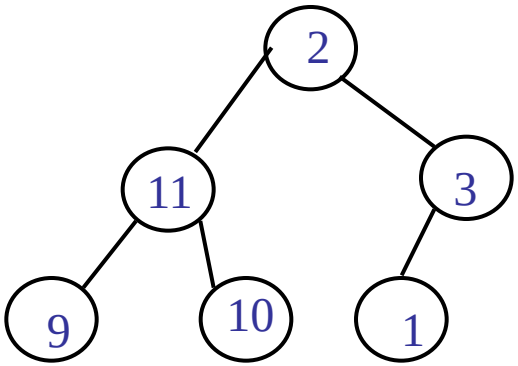




Hjelpemetode – roten i et (sub)tre muligens feil :

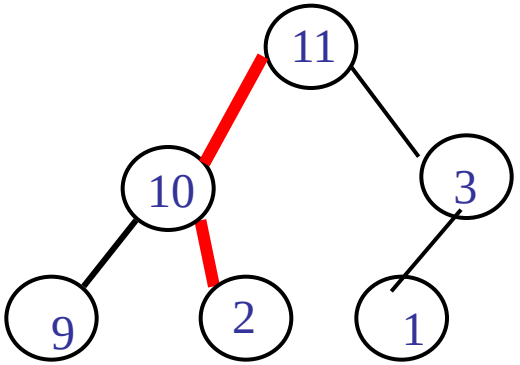
```
static void dyttNed (int i, int n) {
    // Rota er (muligens) feilplassert – dytt 'gammel og liten' rot nedover
    //  få ny, større oppover
    int j = 2*i+1, temp = a[i];

      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
```

Før:



dyttNed (0, 5)

Etter:



a

2	11	3	9	10	1
0	*1*	*2*	*3*	*4*	*5*

a

11	10	3	9	2	1
0	*1*	*2*	*3*	*4*	*5*

Eksekveringstider for dyttNed

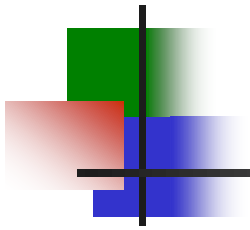
```
void dyttNed (int i, int n) {
  //  Rota er (muligens) feilplassert
 //   Dytt gammel nedover
 //   få ny og større oppover
 int j = 2*i+1, temp = a[i];
    while(j <= n )
    {   if ( j < n && a[j+1] > a[j] ) j++;
         if (a[j] > temp) {
                 a[i] = a[j];
                  i = j;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

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))

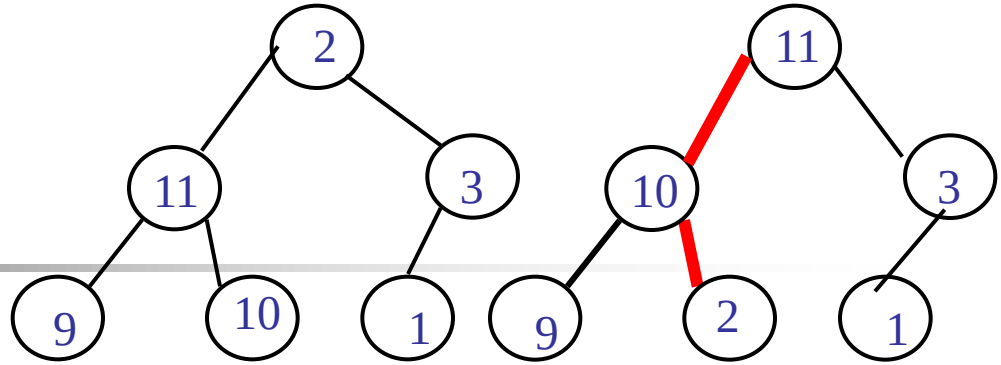
Ideen bak Tre (& Heap)-sortering



- Tre – sortering:
  - Vi starter med røttene, i først de minste subtrærne, og dytter de ned (får evt, ny større rotverdi oppover)
  - Etter denne første ordningen, er nå største element i a[0]

- (Heap-sortering:
  - Vi starter med bladnodene, og lar de stige oppover i sitt (sub)-tre, hvis de er større enn rota.)

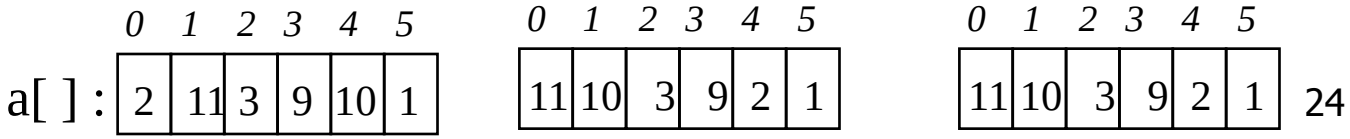
Tre sortering




```
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,..)*



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 , dvs. = 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)

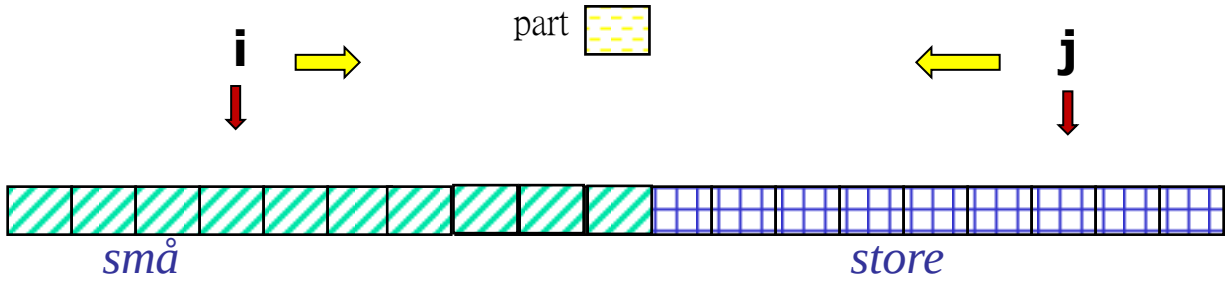
Quicksort – generell idé

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 to 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

små *store*

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.

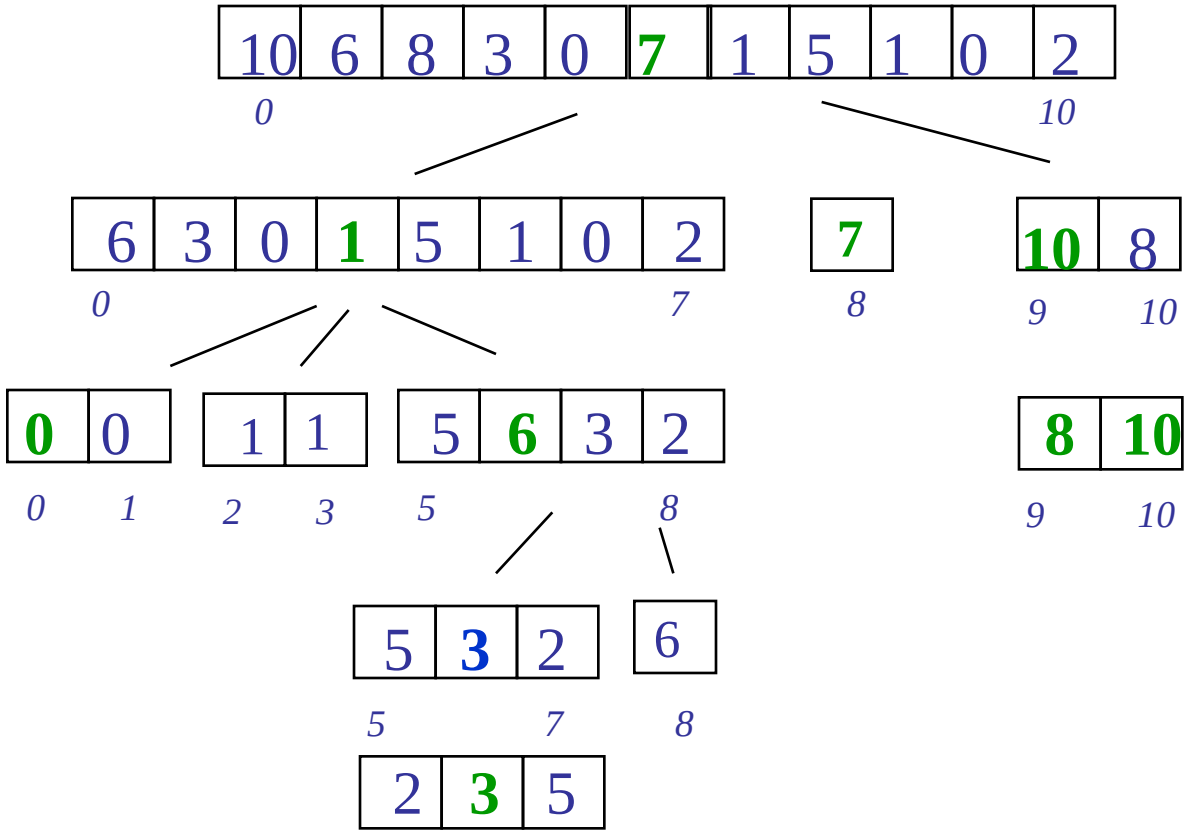



```
void  quicksort ( int [] a, int l, int r)
  { int i= l, j=r;
     int t, part = a[(l+r)/2];

      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 ( l < j ) {   quicksort (a,l,j); }
      if ( i < r ) {   quicksort (a,i,r);  }
} // end quickSort
```

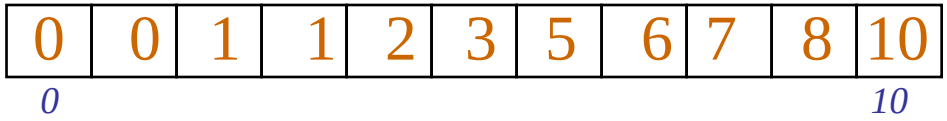
Kall:

```
void quick (int [] a) {
   quicksort(a, 0 , a.length-1);
}
```

QuickSort - eksempel



Sortert :



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 kan velge 'part' som medianen av a[l], a[(l+r)/2 og a[r] og vi får 'alltid' O (n log n)

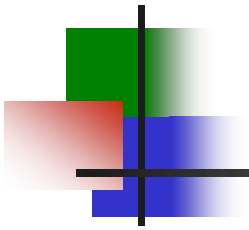
Quicksort i praksis

- Valg av partisjoneringselement 'part' er vesentlig
- Bokas versjon av Quicksort OK, men tidligere versjoner i Weiss var langt dårligere.
- 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)

En helt annen koding av Quicksort (ganske rask):

```
void sekvQuick( int[] a, int low, int high) {
            // only sort arrayseggments > len =1
            int  ind =(low+high)/2,
                  piv = a[ind];
            int  større=low+1,  // hvor lagre neste 'større enn piv'
            mindre=low+1;     // hvor lagre neste 'mindre enn piv'
            bytt (a,ind,low);     // flytt 'piv' til a[lav] , sortér resten

            while (større <= high) {
                        // test iom vi har et 'mindre enn piv' element
                        if (a[større] < piv) {
                                    // bytt om  a[større] og a[mindre], få en liten ned
                                    bytt(a,større,mindre);
                                    ++mindre;
                        } // end if - fant mindre enn 'piv'
                        ++større;
            } // end gå gjennom a[i+1..j]

            bytt(a,low,mindre-1);        // Plassert 'piv' mellom store og små

             if ( mindre-low > 2)  sekvQuick (a, low,mindre-2); // sortér alle <= piv
                                                                //   (untatt piv)
             if ( high-mindre > 0) sekvQuick (a, mindre, high);  // sortér alle > piv
} // end sekvensiell Quick
```

Flette - sortering (merge)



Velegnet for sortering av filer og data i hukommelsen.

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. 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

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.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;}

}
```

II) 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' (kan ikke nyttes til å sortere f.eks to relaterte data som: deltagere og tider i et skirenn, alder og navn i en liste,.. osv):

```
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--;
    }       }
```

To Radix-algoritmer:

- RR: fra høyre og venstre-over (vanligst - iterativ)
- LR: fra venstre og høyreover (raskest - rekursiv)

- Sorterer en array a [] på hvert siffer
  - Et siffer et 'bare' ett visst antall bit
  - RR sorterer *på **siste** (mest høyre) siffer først*
  - LR sorterer *på **første** (mest venstre) siffer først*
- Algoritmene:

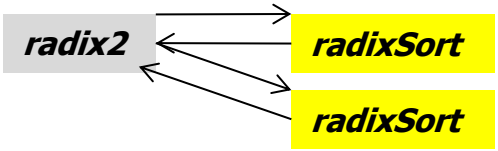
  Begge: Finner først max verdi i a [ ]
  - = bestem største 'siffer' i alle tallene

  a) Tell opp hvor mange elementer det er av hver verdi på det sifferet (hvor mange 0-er, 1-ere, 2-.....) man sorterer på

  b) 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.

  c) Flytter så elementene i a[] direkte over til riktig plass i b[]

To metoder
Radix2 finner konstanter og kaller to ganger :
RadixSort– en gang for hvert siffer vi sorterer på.



1<<numBit

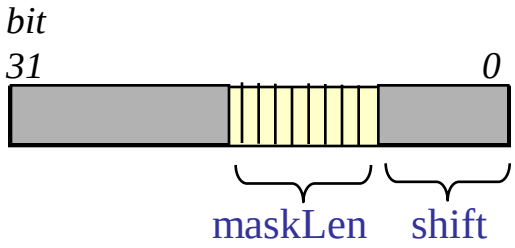
Betyr at vi skifter tallet 1 '*numbit*' plasser oppover i en int.

Eks: 1 << 0 = 1
1 << 2 = 4
1 << 7 = 128

Innledende kode for å radix-sortere på to siffer:

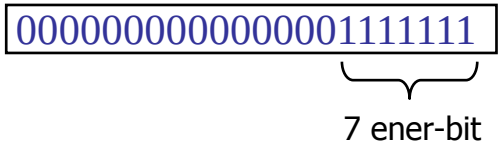
```
static void radix2(int [] a) {
    // 2 digit radixSort: a[ ]
    int max = 0, numBit = 2;

    // finn max = største verdi I a[]
    for (int i = 0 ; i <= a.length; i++)
        if (a[i] > max) max = a[i];

    // bestemme antall bit i max
    while (max >= (1<<numBit)) numBit++;

    int bit1 = numBit/2,    // antall bit i første siffer
        bit2 = numBit-bit1; // antall bit i andre siffer

    int[] b = new int [a.length];
    radixSort( a,b, bit1, 0);
    radixSort( b,a, bit2, bit1);
}
```

a) Hvordan finne sifferverdien i a[i] (antar maskLen=7):



a.1) Lager en maske (= bare ener-bit så langt som vårt siffer er, her 7 bit)

mask = (1<<maskLen) -1;



a.2) Tar AND mellom *a[i]* skiftet ned *shift* plasser og mask.
Da får vi ut de bit i *a[i]* som er 1 og 0, og bare 0 på resten av bit-ene i *a[i].*

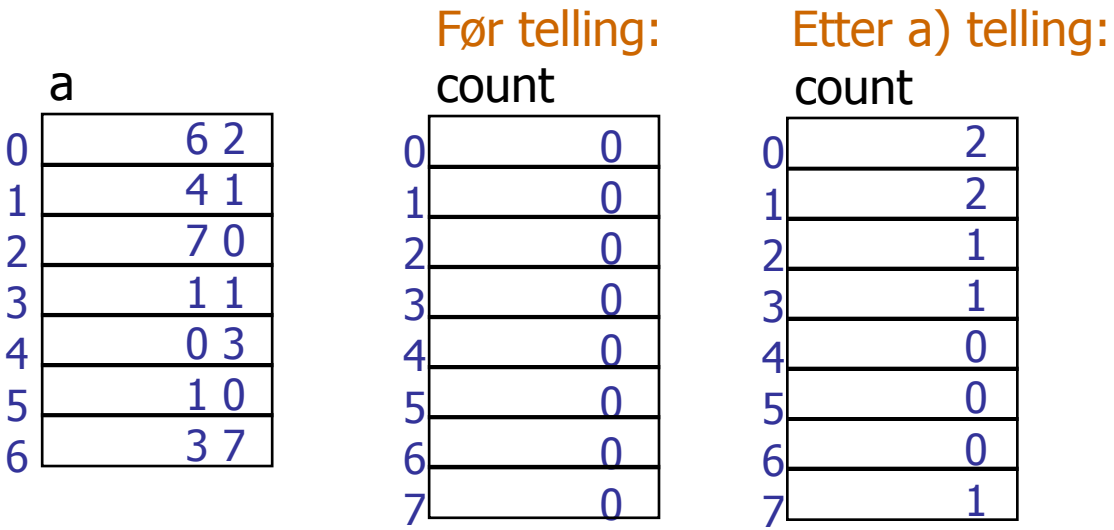
(a[i]>> shift) & mask
gir ett tall mellom 0 - 127

```
static void radixSort (int [] a, int [] b, int maskLen, int shift){
      int  acumVal = 0, j;
      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 < a.length; i++)
        b[count[(a[i]>>shift) & mask]++] = a[i];

  } /* end radixSort */
```

Radix-sortering – steg a) første, bakerste siffer

Vi skal sorterere på siste siffer med 3 bit sifferlengde (tallene 0-7)

a) Tell opp sifferverdier i count[]:



Radix-sortering – steg b) finne ut hvor sifferverdien skal plasseres

De a[i] ene som inneholder 'j' – hvor skal de flyttes sortert inn i b[] ?
- Hvor skal 0-erne stare å flyttes, 1-erne, ….osv

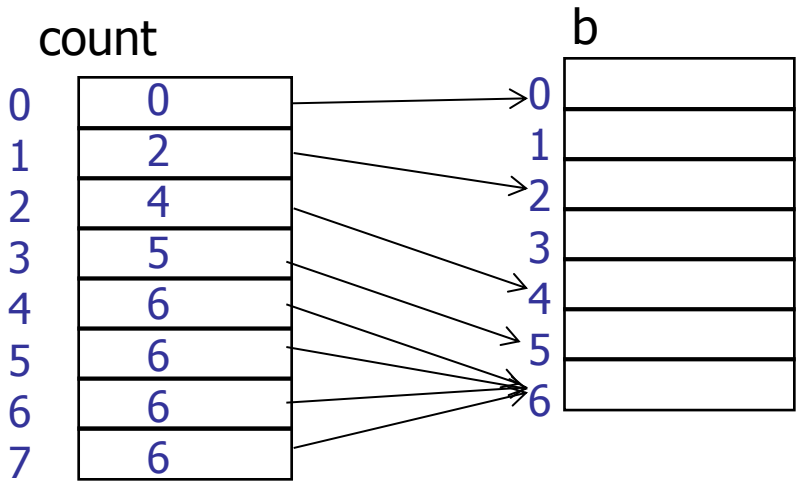
b) Adder opp sifferverdier i count[]: count[i] = $\sum_{k=0}^{i-1} count[k]$

Før addering:

count

0	2
1	2
2	1
3	1
4	0
5	0
6	0
7	1

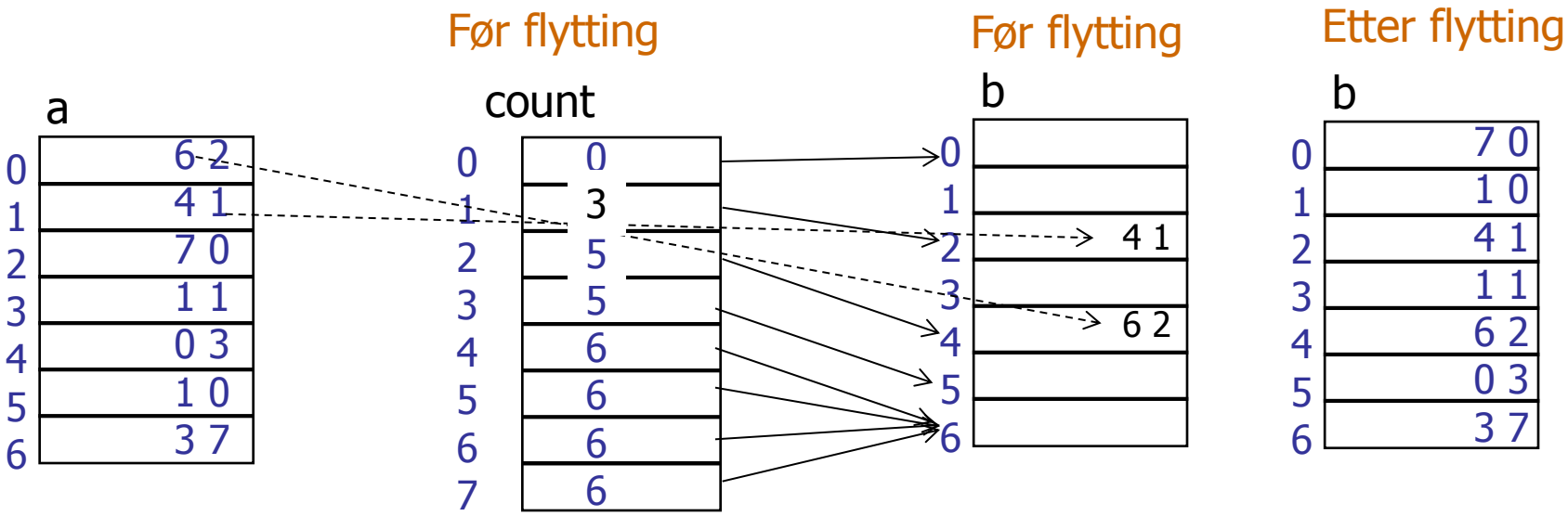
Etter addering :



Kan også sies sånn: Første 0-er vi finner, plasserer vi b[0], første 1-er i b[2] fordi det er 2 stk 0-ere og de må først. 2-erne starter vi å plassere i b[4] fordi 2 stk 0-ere og 2 stk 1-ere og de må før 2-erne,….osv.

Radix-sortering – steg c) flytt a[k] til b[] der count[s] 'peker', hvor s= sifferverdien i a[k]

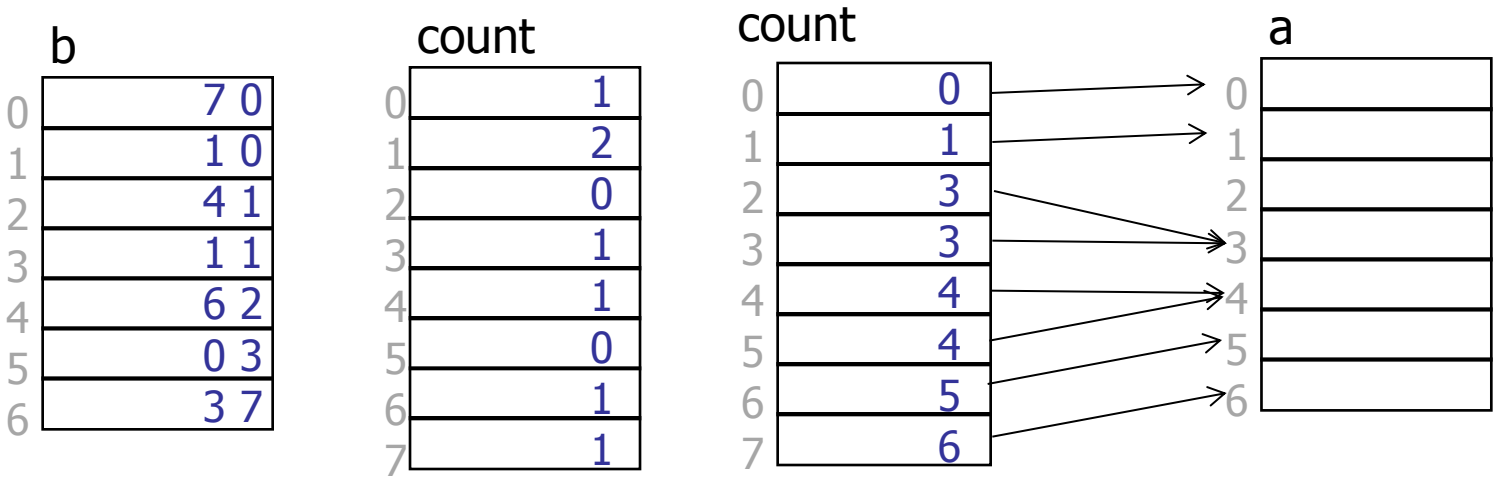
c) flytt a[k] til b[] der count[s] 'peker', hvor s= sifferverdien i a[k], øk count[s] med 1.



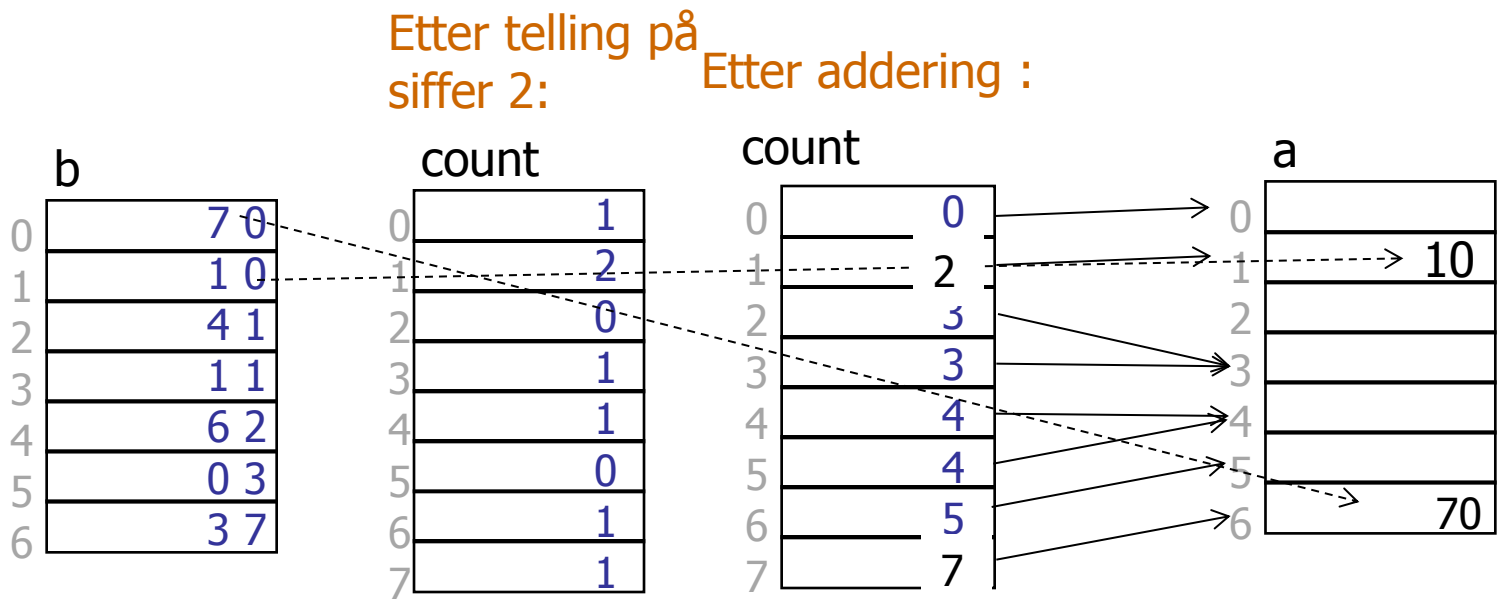
Så sortering på siffer 2 – fra b[] til a[] trinn a) og b)

Etter telling på siffer 2:

Etter addering :

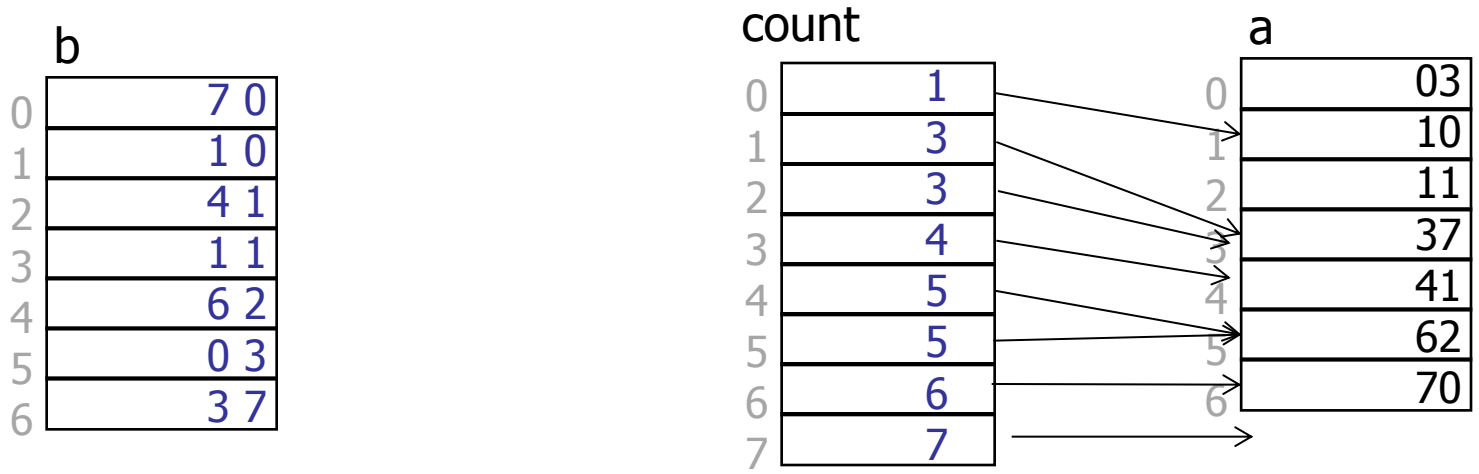


Så sortering på siffer 2 – fra b[] til a[] trinn c)



Situasjonen etter sortering fra b[] til a[] på siffer 2

Etter flytting



a[] er sortert !

Litt oppsummering

- Mange sorteringsmetoder med ulike egenskaper (raske for visse verdier av n, krever mer plass, stabile eller ikke, spesielt egnet for store datamengder,…)
- Vi har gjennomgått
  - Boblesortering : bare dårlig (langsomst)
  - Innstikksorteing: raskest for $n < 0 - 50$
  - Tre-sortering: Interessant og ganske rask : O(n logn)
  - Quick: rask på middelstore datamengder (ca. $n = 50 - 1000$)
  - Radix-sortering: Klart raskest når $n > 500$, men HøyreRadix trenger mer plass (mer enn n ekstra plasser – flytter fra a[] til b[])
  - Hvilken av følgende sorterings-algoritmer er stabile?
    - Innstikk
    - Quick
    - Radix
    - Tre

  Svar: Radix og Innstikk er stabile, Quick og Tre er ikke stabile.