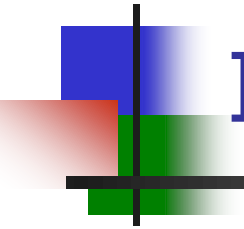


INF2220 – høsten 2017, 12. okt.

Sortering (kap. 7.) – sekvensiell sortering I

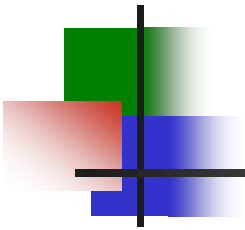
Arne Maus,
Gruppen for Programmering og Software Engineering (PSE)
Inst. for informatikk, Univ i Oslo

Essensen av INF2220

- 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 løsninger og
  - måle eksekveringstider og kunne vurder ulike algoritmers forventet **O(n)**
- 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. (HøyreRadix-sort)**
- **0,005 sek (parallell HøyreRadix)**



Sortering – oversikt idag

- Definisjon av Sortering
  - Ulike kriterier (hastighet, plassbehov, stabil sortering,..)
- Maskinene vi sorterer, utfordringer med tidtaking
  - JIT-kompilering, optimalisering i flere trinn
  - Cache-systemet
  - Hvordan få riktige tider
  - Hvorfor tidtaking når vi har kompleksitets-analyse O()?
    - Svar: Hvis du har 2 eller 10 algoritmer som er O(n*logn) – hvilken er best ?
- To ulike prinsipper for sortering
  - Sammenligning eller verdibasert
- I dag se på 5 algoritmer, 3 i dybden– flere algoritmer neste uke.
  - Innstikk-sortering (sammenligning)
  - HøyreRadix (verdibasert – raskest)
  - Oblig 3 = Venstre Radix (verdibasert – nesten like rask)

Tidsmålinger – (1992 – 450 MHz)

Hvor lang tid bruker (da fantes ikke JIT-kompilering):

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	?

Tabell 17.1 *Gjennomsnittlige kjøretider i µs med Java 1.8.0 for ulike Java-konstruksjoner og et enkelt sorteringsprogram som funksjon av antall utførte ganger, Intel i7-7600 @3,4 GHz, 8(4) kjerner.*

2017-målinger

n, ant. ganger	1	2	10	100	10000	100000	x bedre
for-løkke	0,3	0,15	0,03	0,018	0,009	0,007	42
metode-kall	2697	0,45	0,06	0,054	0,026	0,026	103730
new int[100]	1,2	0,6	0,24	0,195	0,151	0,136	33
array copy med for-loop, n=100	1,8	1,5	2,64	2,500	1,177	0,188	9
System.arraycopy, n=100	5,7	0,3	0,15	0,126	0,072	0,064	89
new Thread med start & join()	3015	336	66,6	61,68	61,87	61,86	48
new C(int) og metodekall	2697	0,45	0,15	0,21	0,035	0,035	77 057
array read	0,3	0,3	0,06	0,036	0,012	0,012	25
Innstikk-sortering (n=100)	46,6	42,8	42,42	21,27	19,60	1,45	32

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.

```
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
```

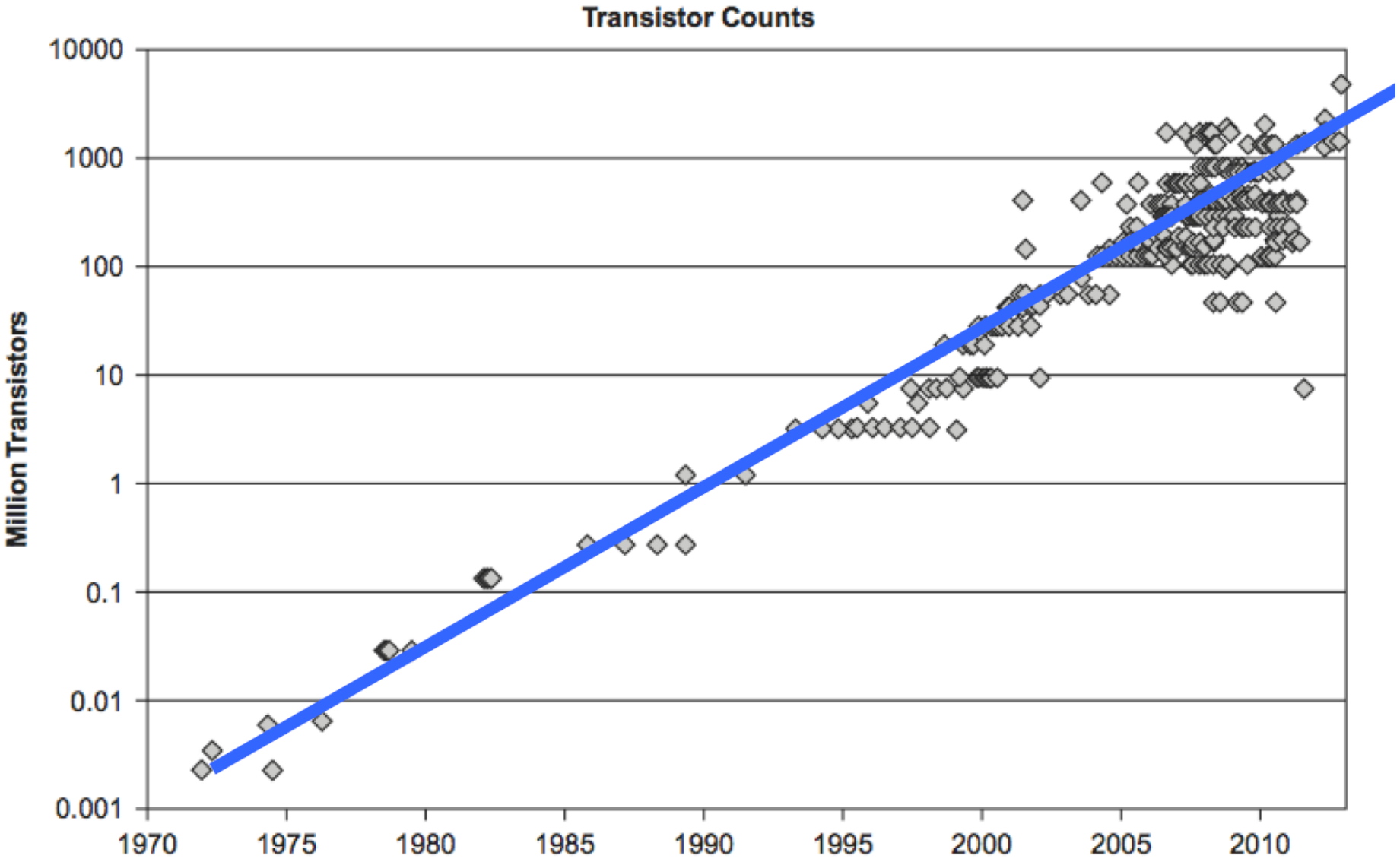
Løsning på tidtaking med JIT-kompilatoren

1. Kjør selve programmet som tar tid flere ganger, minst 3 ganger, og ta medianen av tidene
   - Dette løser du med en ekstra for-løkke rundt koden for det du vil ta tiden på. Alle tidene tas inn en double-array for hver algoritme du tester. Så innstikksorterer du disse arrayene og tar ut midt-elementet.
2. Kjør løkka med **n** fra største verdi og nedover mot minste n.
   - Bare da får du fornuftige verdier for små verdier av n:

```
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.
 }}
```

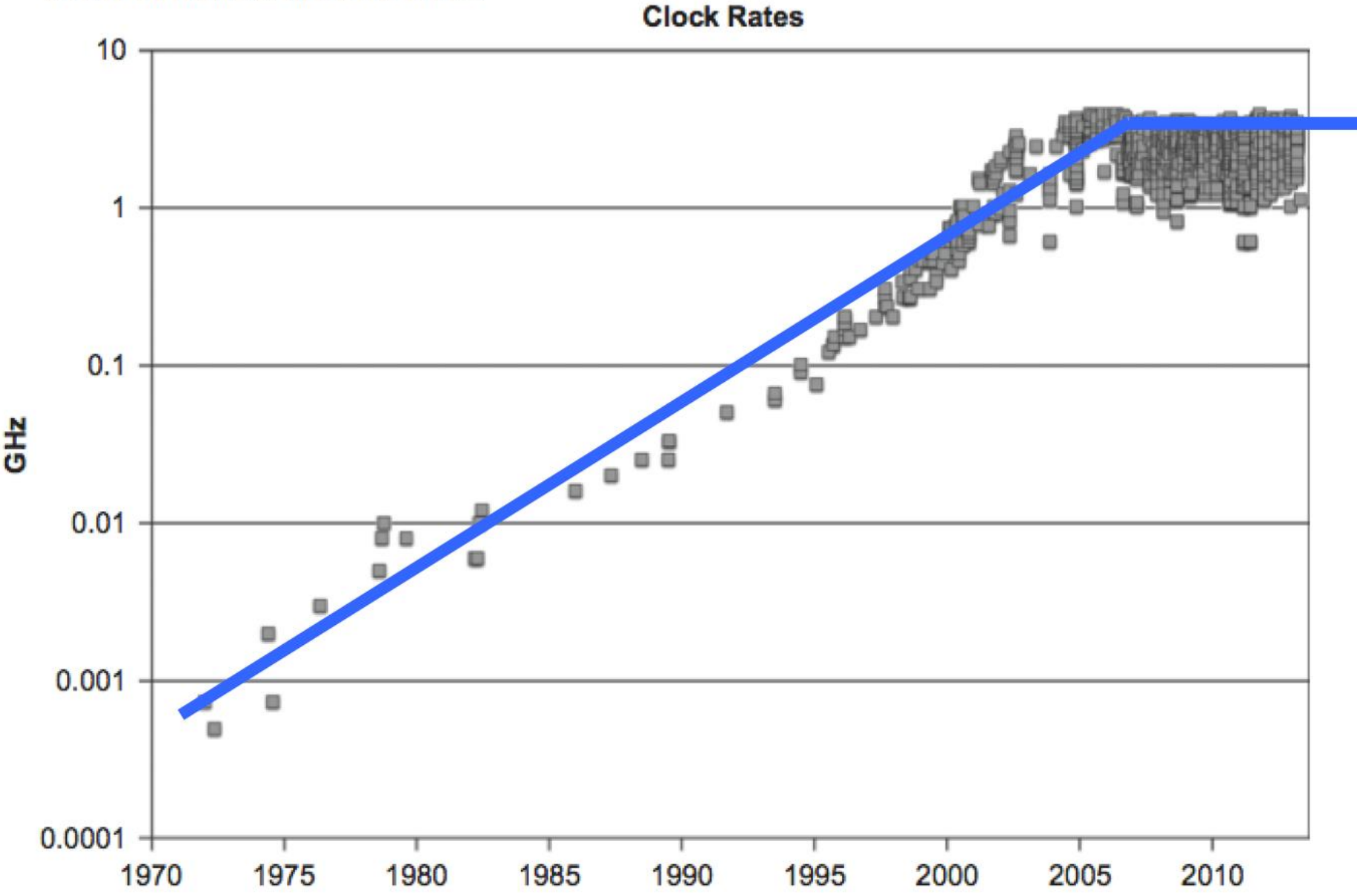
Transistors per Processor over Time

Continues to grow exponentially (Moore's Law)

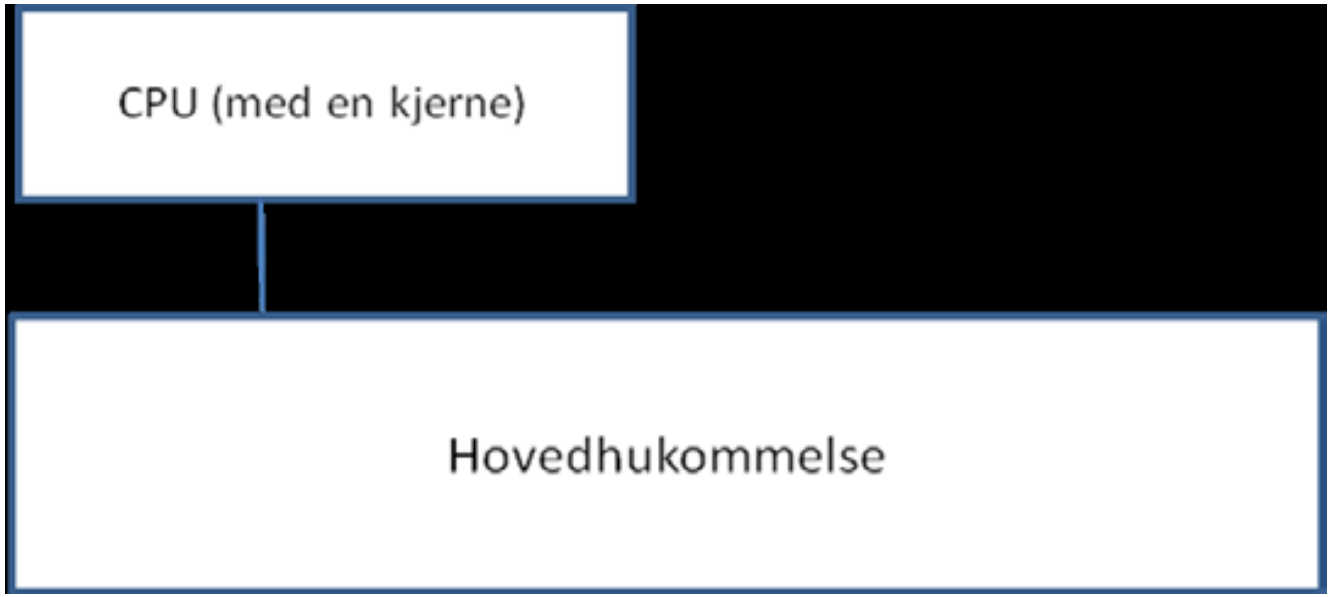


Processor Clock Rate over Time

Growth halted around 2005

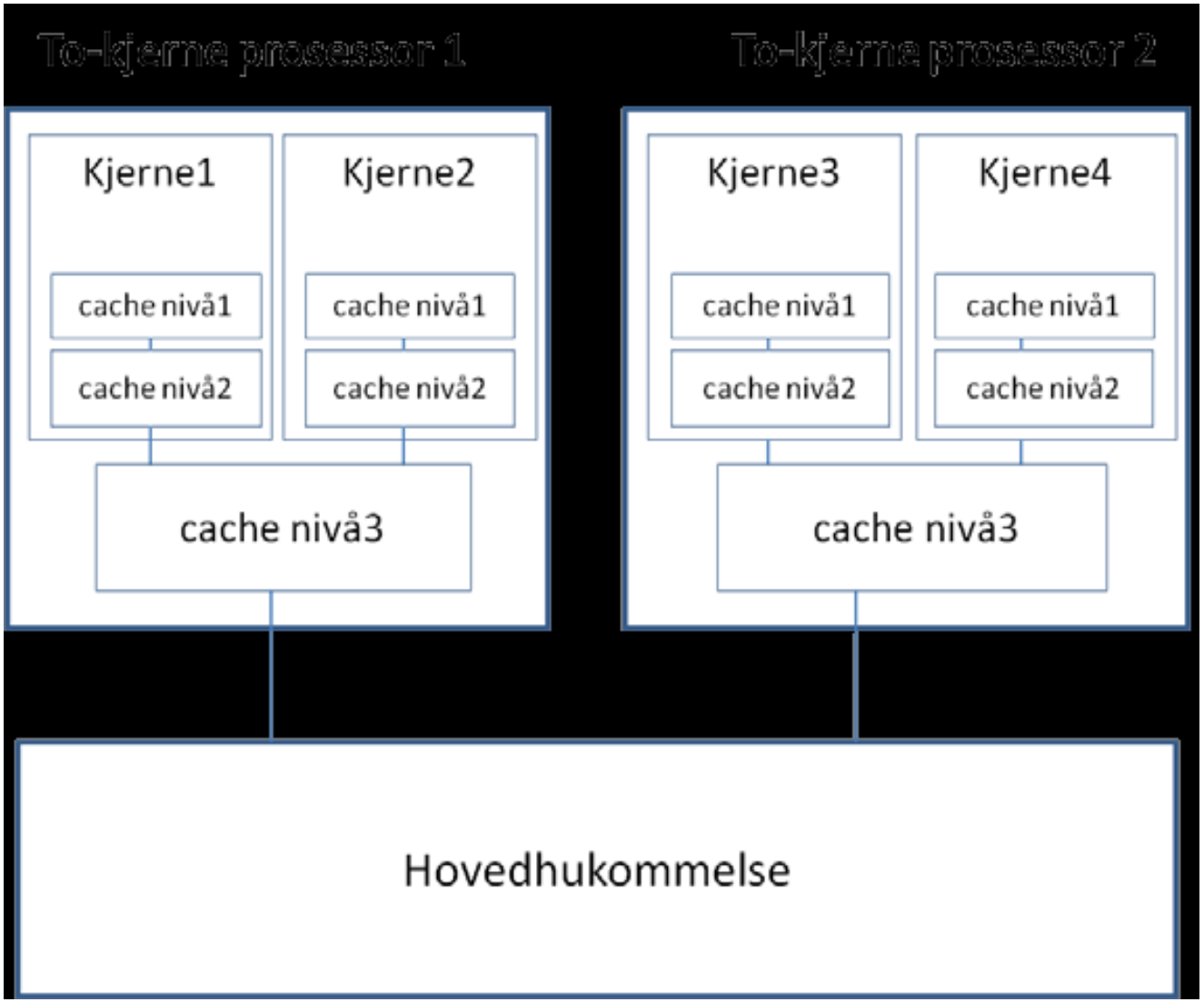


Maskin 1980 (uten cache)

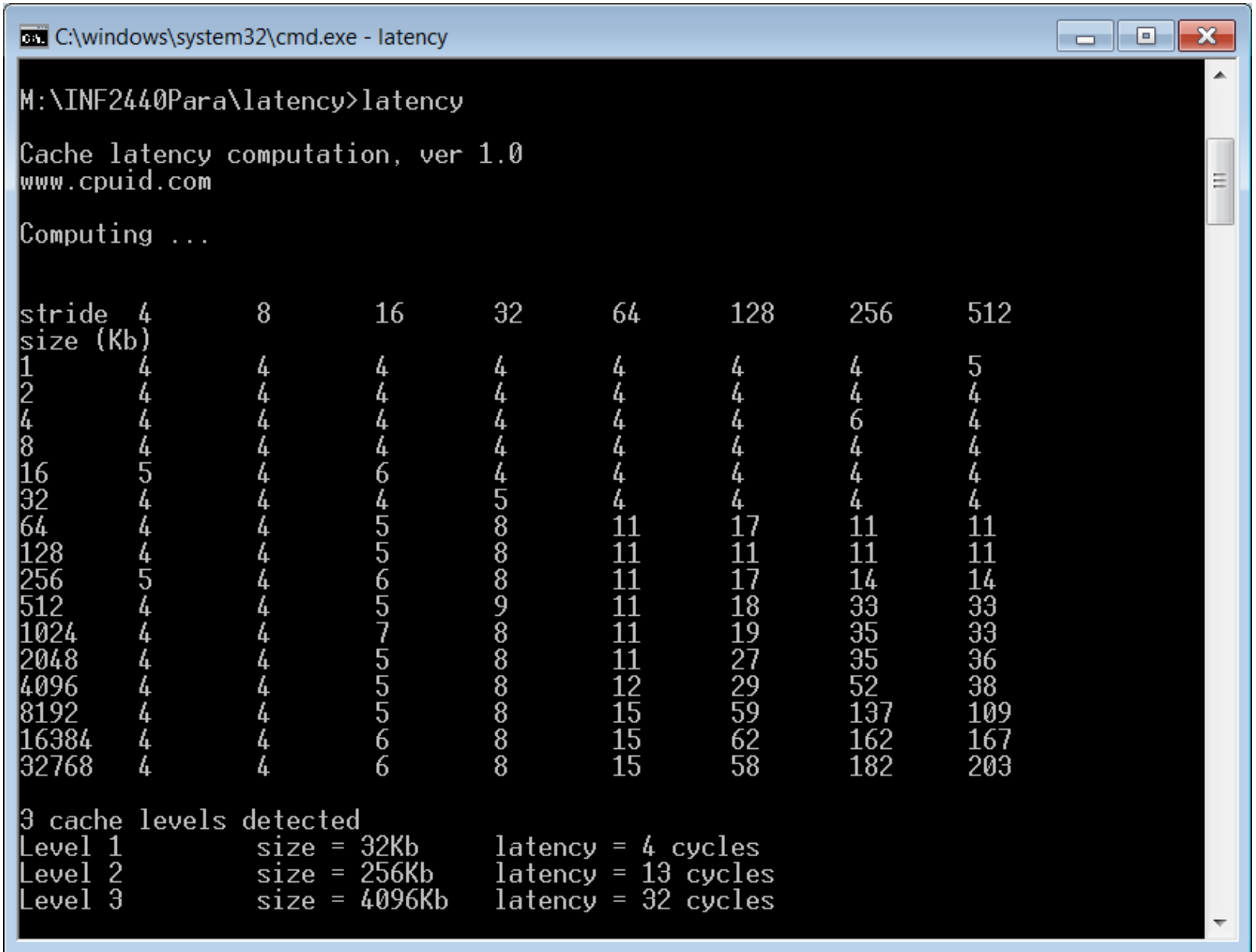


Skisse av en datamaskin i ca. 1980 hvor det bare var én beregningsenhet, en CPU, som leste sine instruksjoner og både skrev og leste data (variable) direkte i hovedhukommelsen.

Maskin ca. 2015 med to dobbeltkjerne CPU-er

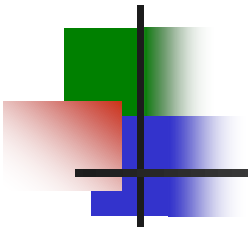


Test av forsinkelse i data-cachene og hovedhukommelsen - latency.exe (fra CPUZ)



Oppsummering – ideen om at vi har uniform aksesstid i hukommelsen er helt gal

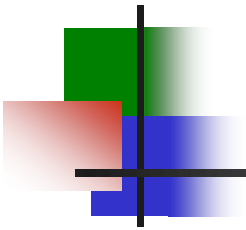
- Hukommelses-systemet i en multicore CPU (Intel Core i5-459 3.3 GHz) – mange lag (typisk aksesstid i instruksjonssykler):
  1. Registre i kjernen (1) – 8/32 registre
  2. L1 cache (4) – 32 Kb
  3. L2 cache (13) – 256 kb
  4. L3 cache (32) – 8Mb
  5. Hovedhukommelsen (virtuell hukommelse) (ca. 200) – 8-16 GB
  6. Disken (15 000 000) = 5 ms – 1000 GB



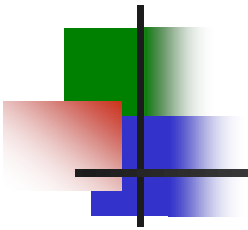
Sortering

- Hva sorterer vi i praksis
  - Bare tall eller tekster – eller noe mer (og sammensatt)
- Hvordan definere problemet
  - Krav som må være oppfylt
- 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?

Hva er kravene til en sorteringsalgoritme



- 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 (forklares senere)
- Bruker lite plass
  - Helst ikke O(n) ekstra plass, men ikke så viktig lenger



To klasser av sorteringsalgoritmer

- Sammenligning-baserte:
  Baserer seg på sammenligning av to elementer i a[ ]
  - 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

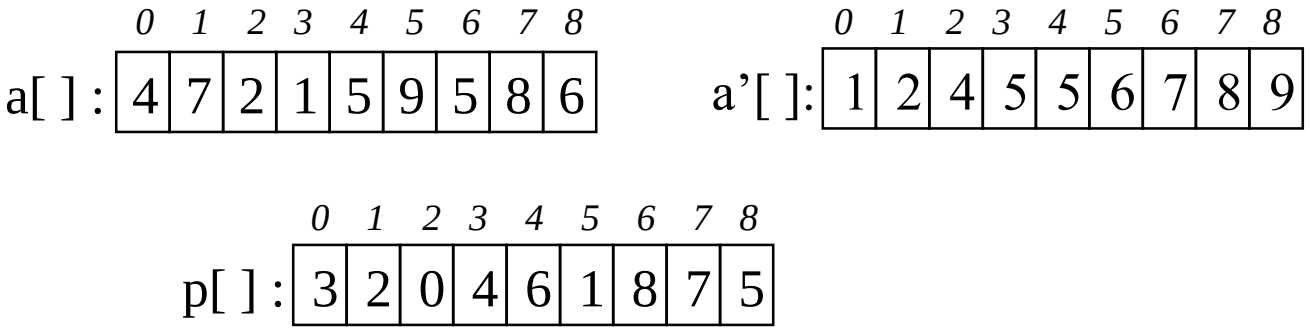
Sorteringsproblemet, definisjon.

- Kaller arrayen a[] før sorteringen og a'[ ] etter
  - n = 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 må virke hvis er to (eller fler) like verdier i a[ ]
  - I 'bevisene' antar vi alle a_i er forskjellige: $a[i] \neq a[j]$, når $i \neq j$.
- 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
  - 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 ) {
                    ..........

}
```

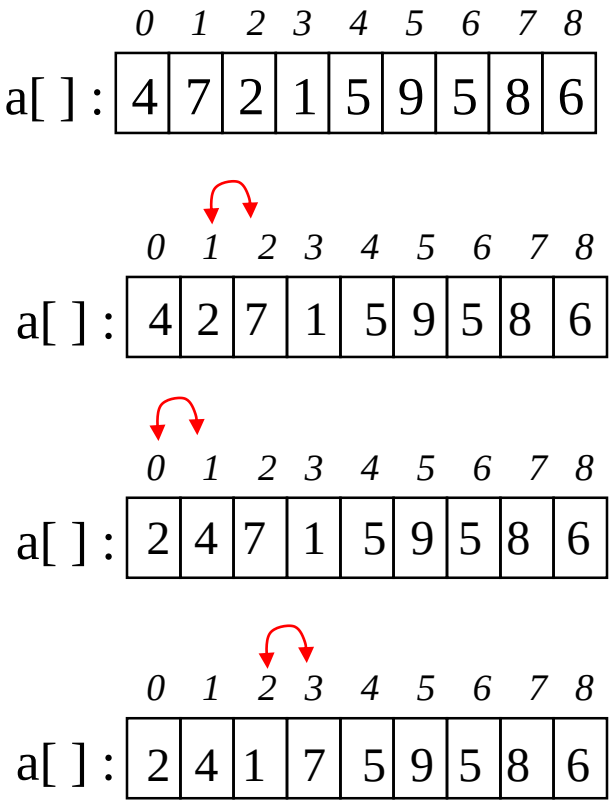
HER:

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

        tmp = a[i],

            if ( tmp < a[j] ) {

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

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

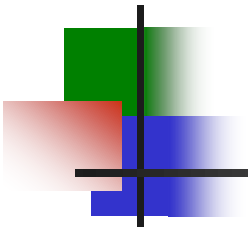
Det er gjennomsnittlig n(n-1)/4 inversjoner i en array av lengde n.

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

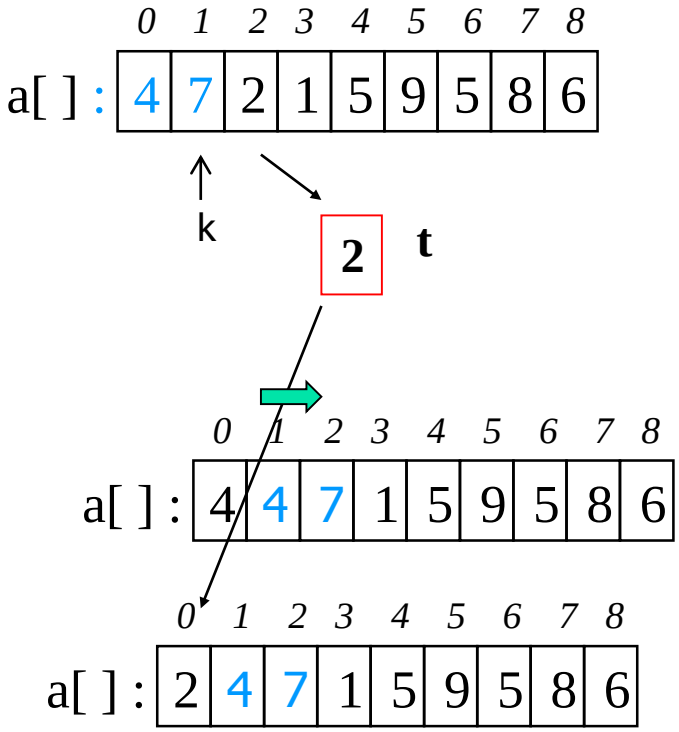


- Boble er opplagt $O(n^2)$ fordi:
  - Th. 7.1 sier at det er $O(n^2)$ inversjoner, og en naboombytting fjerner bare en slik inversjon.
- Kunne også argumentert som flg.:
  - Vi går gjennom hele arrayen, og for hver som er i gal rekkefølge (halvparten i snitt) – bytter vi om disse (i snitt) halve arrayen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

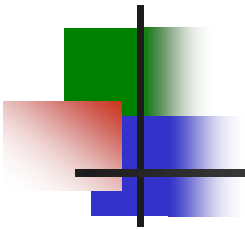
Innstikk-sortering – likevel best for n < 50

```
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.*

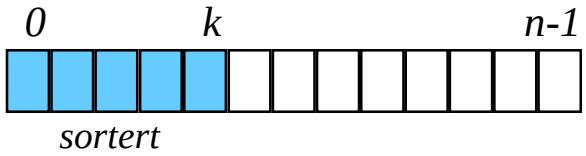


Generell design-metodikk



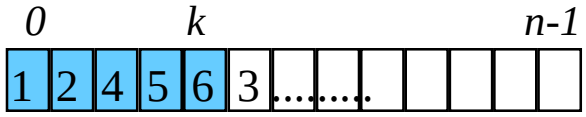
- Gitt en klar spesifikasjon med flere ledd
- En av delene i spesifikasjonen nyttes som invariant i programmets (ytterste) hovedløkke i en litt endret form.
  (løkke-invariant = noe som er sant i begynnelsen av løkka)
- Dette kravet svekkes litt (gjøres litt enklere); gjelder da typisk bare for en del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

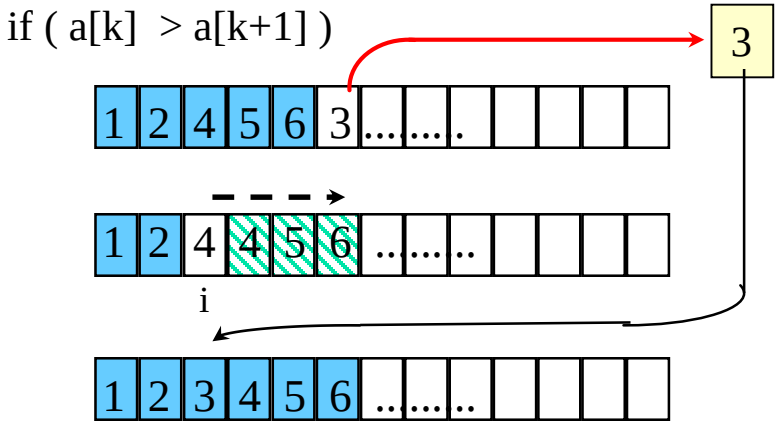
Design, innstikksortering



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*)

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) { // tvilsom
        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 og rask)
- LR: fra venstre og høyre-over (rask - rekursiv)

- Sorterer en array a [] på hvert siffer
  - Et siffer et 'bare' ett visst antall bit vi velger (eks. 6-13)
  - RR sorterer *på **siste** (mest høyre) siffer **først***
  - LR sorterer *på **første** (mest venstre) siffer **først***
- Algoritmene:

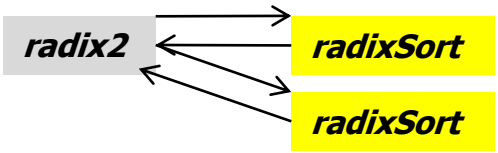
  Begge: Finner først max verdi i a [ ]
  - = bestem største 'siffer' i alle tallene

  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

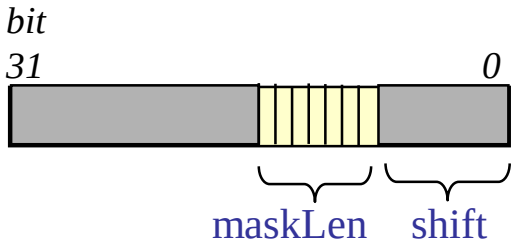
Betyr at vi skifter tallet 1 '*numbit*' plasser oppover i en int.

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

Innledende kode for å høyre-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;

00000000000000001111111

7 ener-bit

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].*

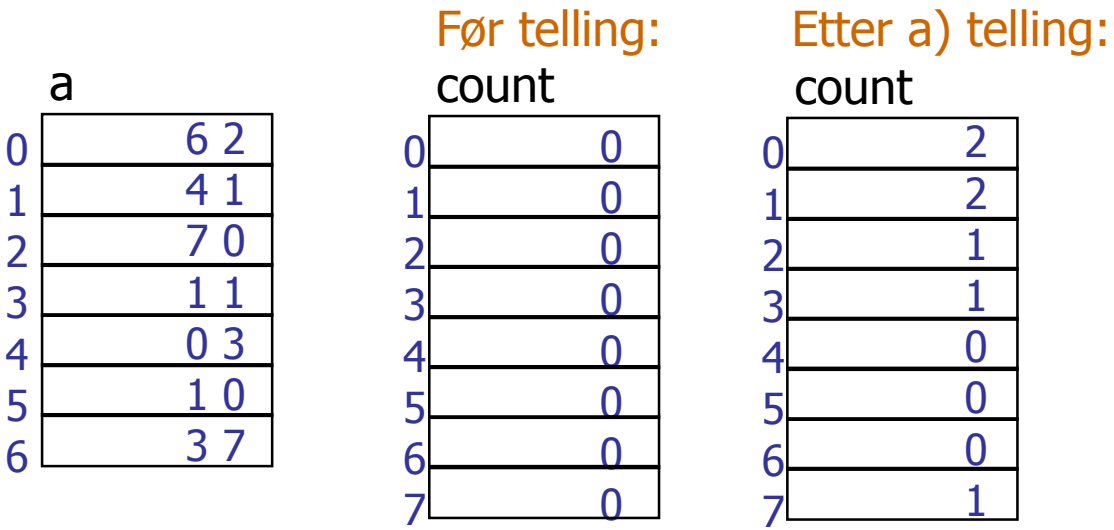
(a[i]>> shift) & mask
gir ett tall mellom 0 - 127

```
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

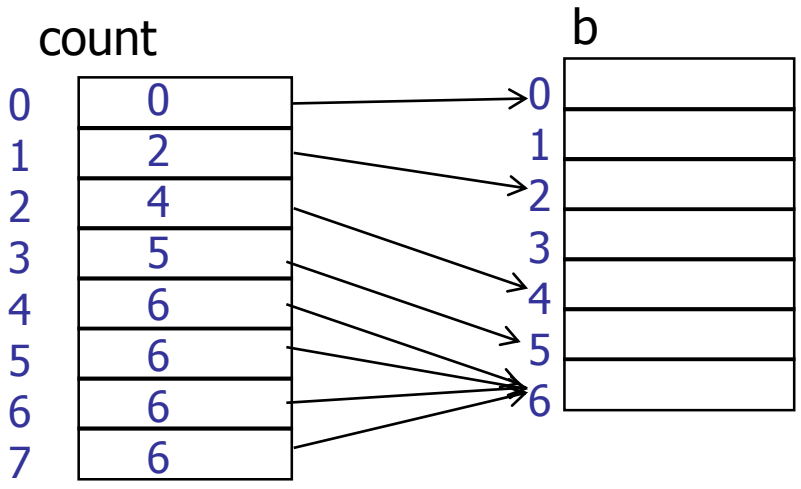
b) Adder opp sifferverdier i count[]: count[i] = $\sum_{k=0}^{i-1} count[k]$

Før addering:

count

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

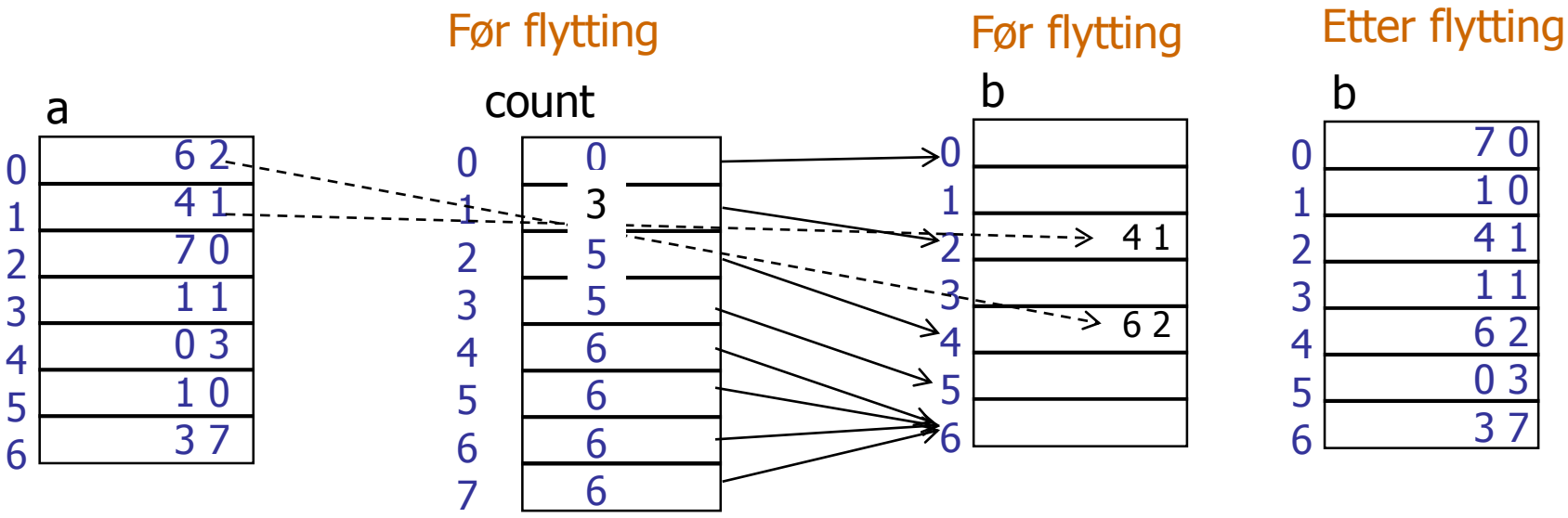
Etter addering :



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]

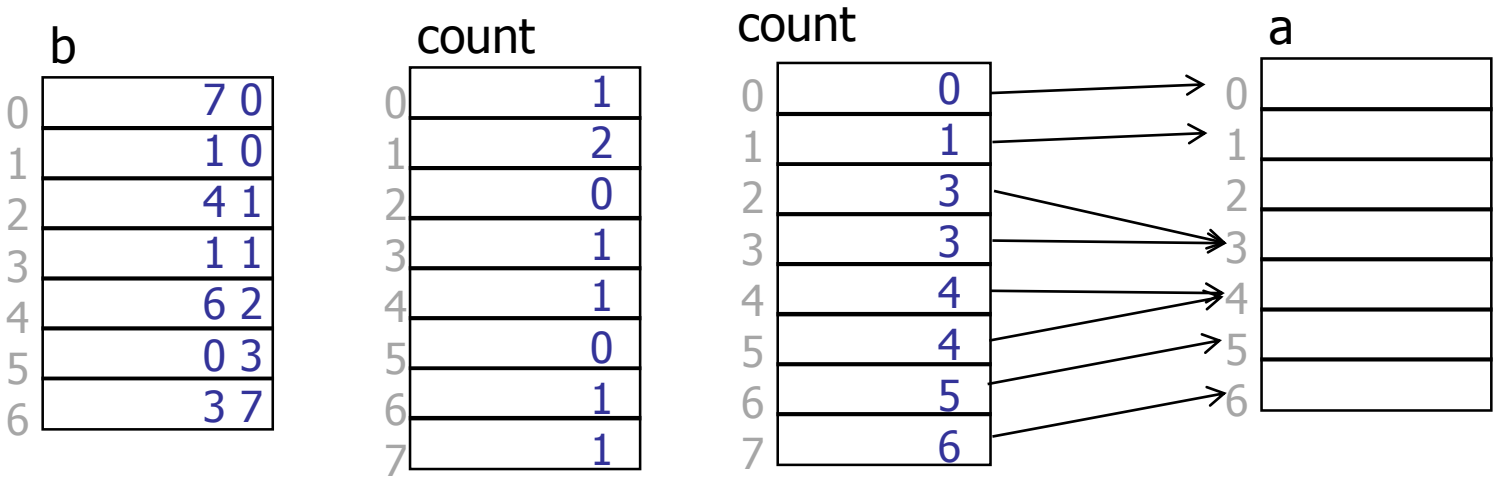
c) flytt a[k] til b[] der count[s] 'peker', hvor s= sifferverdien i a[k], øk count[s] med 1.



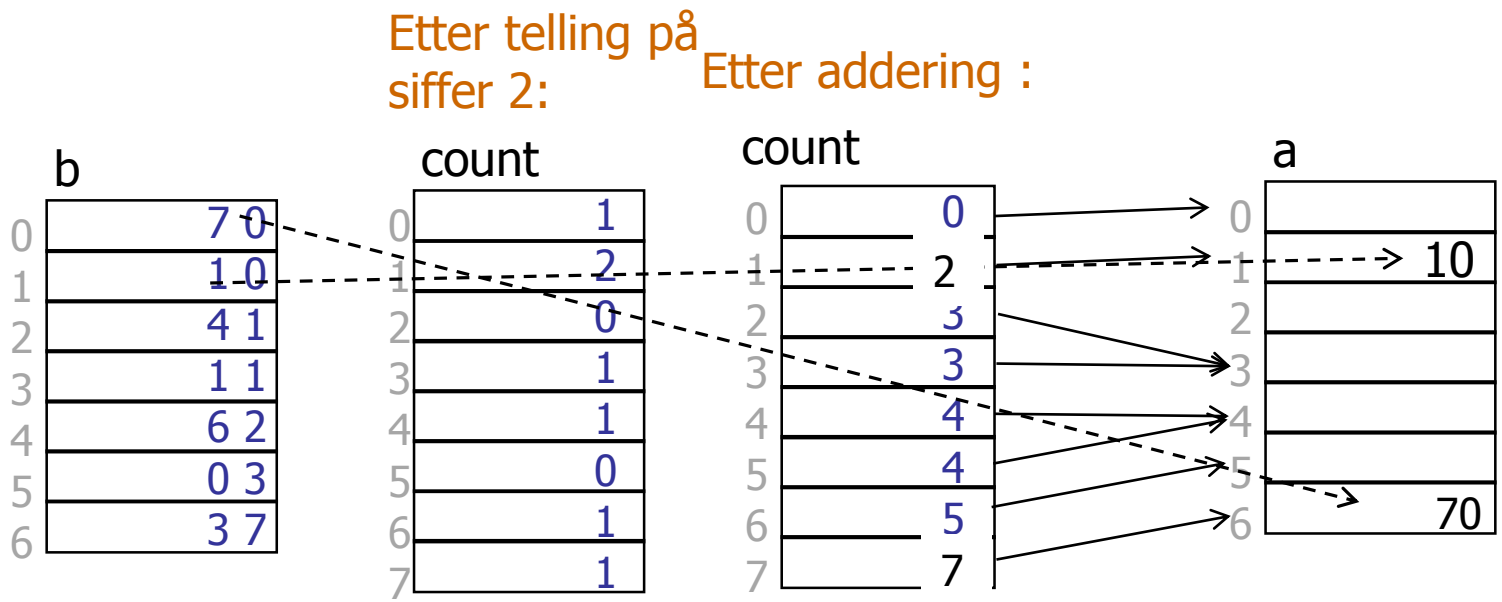
Så sortering på siffer 2 – fra b[] til a[] trinn a) og b)

Etter telling på siffer 2:

Etter addering :

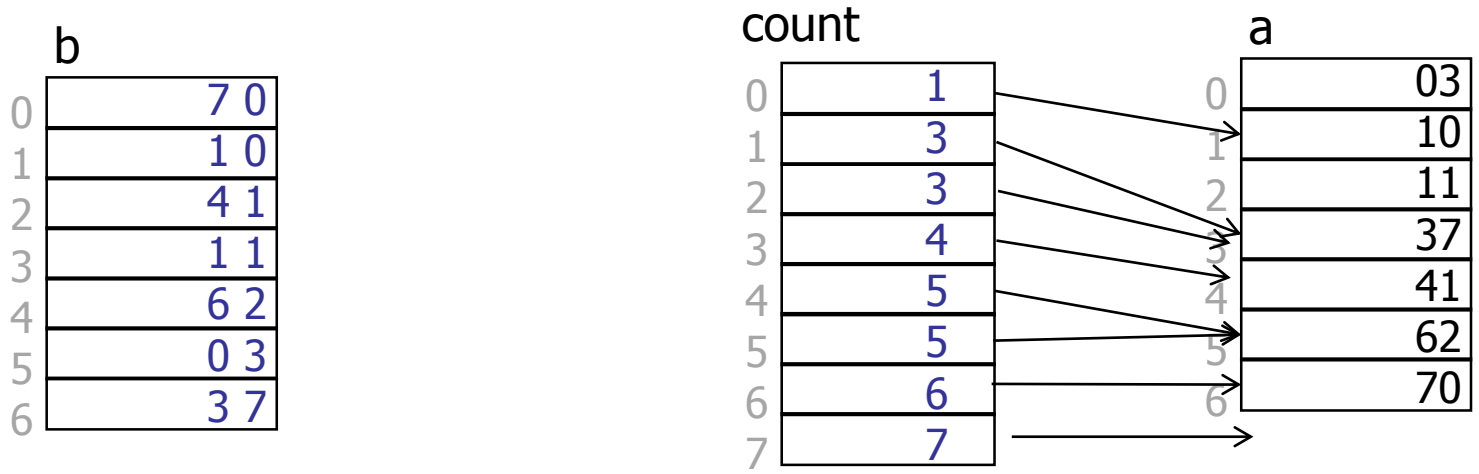


Så sortering på siffer 2 – fra b[] til a[] trinn c)



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

Etter flytting



a[] er sortert !

VenstreRadix – hvordan programmere Oblig3 ?

Oblig 3 i INF2220 – h2017. Sekvensiell rekursiv VenstreRadix sammenlignet med Kvikksort.

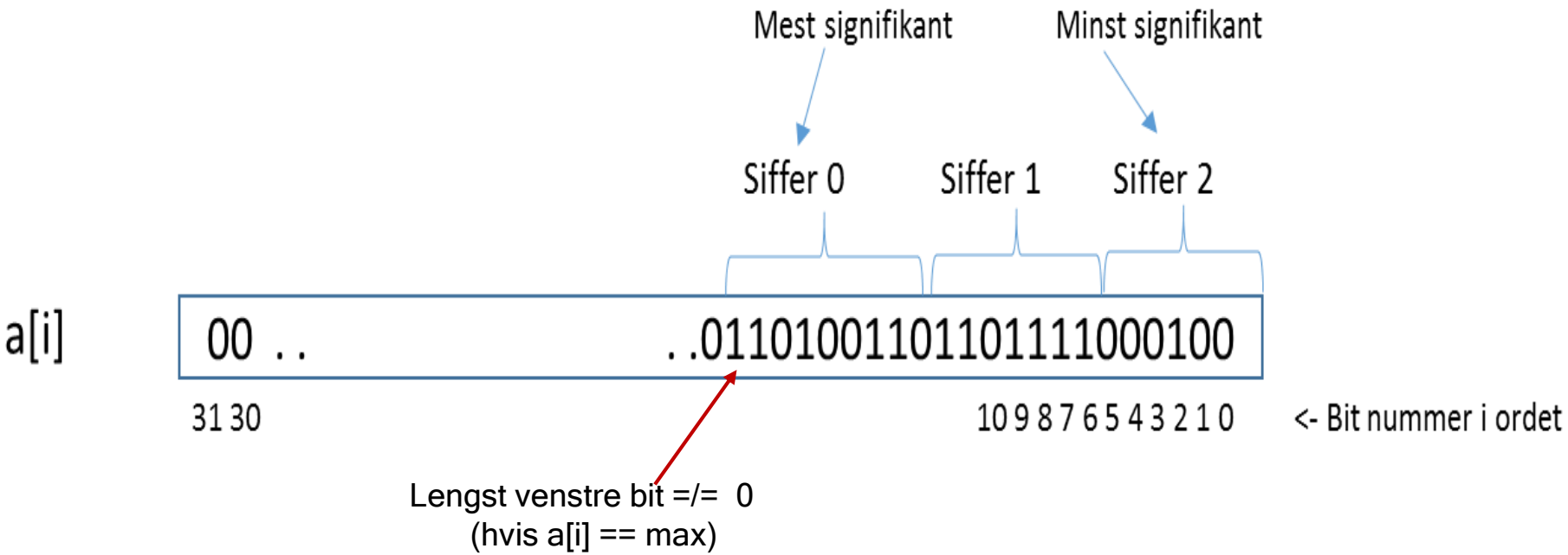
Radix-sortering finnes i to høyst ulike varianter – høyre-radix og venstre-radix. Felles for dem begge er at de ser på verdien av ett element av gangen og sorterer det på denne verdien (sammenligner ikke to elementer for å sortere). Begge algoritmene deler altså opp de delene av alle tallene som ikke bare er 0, i ett eller flere sifre (et siffer er et antall sammenhengende bit i tallet). Hvor mange bit man bruker i hvert siffer er bare et optimaliseringsspørsmål.

HøyreRadix er en iterativ algoritme som går gjennom alle tallene 2 x antall-sifre og starter med det høyre, minst signifikante sifferet først,

mens VenstreRadix er en rekursiv algoritme som sorterer en array og starter med å sortere på det første, mest signifikante sifferet, så på det nest mest signifikante sifferet, …, osv. helt til alle sifrene i tallene er sortert.

Vi skal altså sortere en : **int [] a = new int[n] ;**

Vi deler opp tallene i a[] i et antall sifre. Alle tallene er delt opp på samme måte.



Hvis et siffer er 1 bit lang, har vi et totall-system, har vi f.eks. 3 bit i et siffer, har vi et åtte-talls system (oktale tall), og har vi 9 bit i et siffer, har vi et 512-talls system,... - det f.eks. er 8 mulige verdier i 3 bit betraktet som et tall

VenstreRadix kan sees på som en variant av KvikkSort (som riktignok ble laget senere). I Kvikksort skiller vi på to verdier (store og små). Det gjøres med sammenligninger av alle elementene med en felles verdi (pivot).

*I VenstreRadix skiller vi f.eks. på **256** mulige verdier ved 'bare' å se på verdien av et bestemt siffer i a[i] i hver rekursjon (**hvis dette sifferet er 8 bit langt**).*

Riktig implementert er VenstreRadix vesentlig raskere enn KvikkSort, og en del av oblig3 går ut på å sammenligne din implementasjon av VenstreRadix med Javas innbygde sorterings-algoritme Arrays.sort(int[] a) som bruker Kvikksort.

Et siffer er altså et bestemt antall bit, alt fra 1 til 30, og **vi velger selv** hvor stort siffer vi sorterer med.

Når vi sorterer med flere sifre, **behøver ikke alle sifrene** være like lange, men **for hvert siffer blir selvsagt alle tallene sortert med samme antall bit**.

Er et siffer numBit langt, er de mulige sifferverdiene **:**

$0,1,\ldots, (2^{numbit} - 1)$

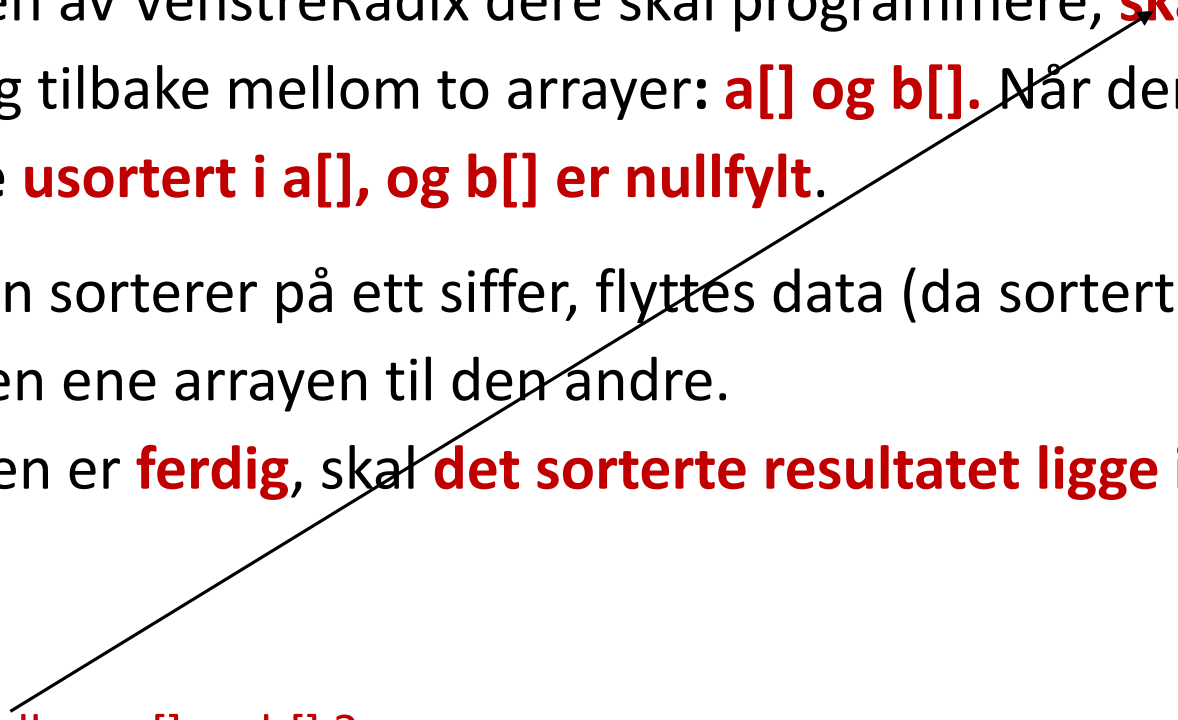
(f.eks. dersom sifferet er 9 bit langt, er sifferverdiene: 0,1, …, 511).

Effektivitetsmessig lønner det seg at det er fra 6 til 10 bit i ett siffer.

I den versjonen av VenstreRadix dere skal programmere, **skal** data flyttes frem og tilbake mellom to arrayer**: a[] og b[].** Når dere **starter er alle tallene usortert i a[], og b[] er nullfylt**.

Hver gang man sorterer på ett siffer, flyttes data (da sortert på dette sifferet) fra den ene arrayen til den andre.

Når sorteringen er **ferdig**, skal **det sorterte resultatet ligge i arrayen a[].**



Hvorfor flytte mellom a[] og b[] ?

Svar: Fordi det bla. på mine hjemmesider ligger to-tre litt komplisert venstre-radix koder som flytter elementene syklisk rundt i en array; effektivt men vanskelig. Det hjelper dere **intet** å se på den, eller kopiere den.

Denne versjonen av Radix-sortering som dere skal lage i Oblig3, MultiVRadix, velger selv **hvor mange** sifre den vil sortere på avhengig av hvor mange bit det er i den største verdien i arrayen a [] (=max) som skal sorteres. Denne max-verdien finner dere først, før sorteringen begynner.

For å få riktig sortering og fart på sorteringen, bør man vurdere følgende punkter:

- Er det den delen man skal sortere, rimelig kort, er **InnstikkSort** en raskere algoritme for den jobben (og da er sorteringen av denne delen av tallene avsluttet for alle sifrene).
- Når vi **rekursivt går nedover** med f.eks. med sifre på 8 bit, deler vi opp tallene hver gang i 256 ulike deler, og det er ikke sikkert at rekursjonen går like dypt i hver av disse delene (f.eks. hvis det ikke er noen tall med én av de 256 ulike verdiene eller hvis vi har brukt Innstikksortering, er jo rekursjonen avsluttet nedover den grenen).
- Ett element er sortert (men er det i riktig array ? – a[] eller b[])

- Når vi har sorterer ferdig en del av arrayen, hvis vi har sortert et ulike antall ganger (= antall sifre), vil det sorterte resultatet ligge i b[]. Det må da som en siste operasjon i denne delen kopieres tilbake til a[].
- For å få en raskest algoritme, må man bare kopiere tilbake fra b[] til a[] de gangene det er nødvendig. Husk at rekursjonen kan gå til ulike dybder i de ulike grenene.
- For å eksperimentere med hastigheten for din løsning bør du ha to konstanter, en som sier hvor kort en del av a[] skal være før du bruker InnstikkSort på den, og en konstant som sier hvor mange bit du i utgangspunktet vil bruke i hvert siffer.

Det du skal levere er **programkoden og en rapport** som først viser kjøretider av algoritmen din og Arrays.sort() for n = 100, 1000, 10 000, 100 000, 1 mill. og 10 mill. som du har trukket med nextInt(n) metoden i bibliotekklassen Random.

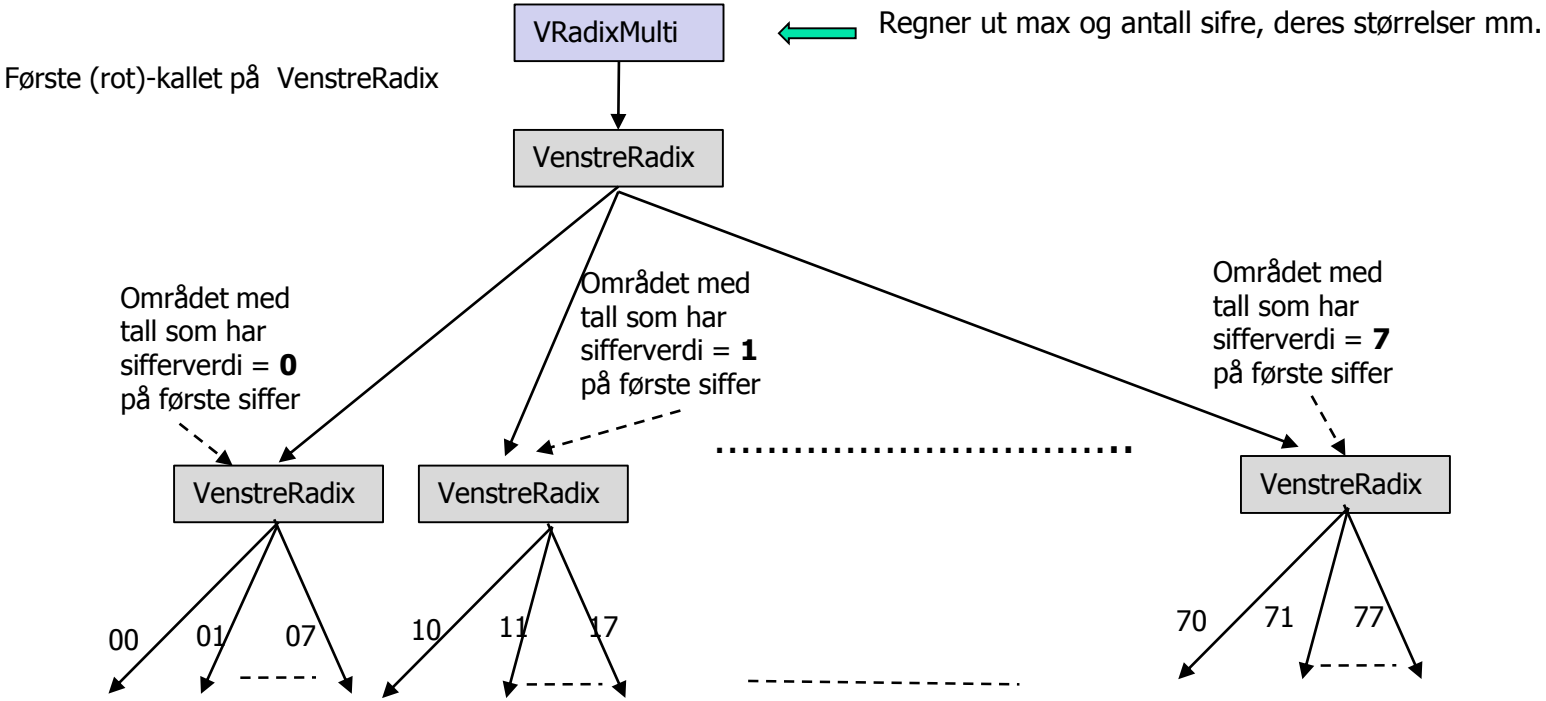
- Løsningen skal også inneholde en enkel test på om arrayene er sortert (a[i-1] ≤ a[i], i=1,2, …, a.length-1) og den skal gi en feilmelding (se forslag til kode). Denne testen kjøres utenfor tidtakingen.
- For at dette skal bli en sammenligning med KvikkSort lar du også Arrays.sort (int [] a) sortere samme usorterte array (du må da trekke verdiene i a[] om igjen).
- I tabellen din skal du altså ha tre rader, en for tidene for hver verdi av n for din VRadixMulti og en for Arrays,sort() og en linje for **speedup** definert som **Tid(KvikkSort)/Tid(VRadixMulti).** Kommenter hvorfor VRadixMulti er så mye raskere (i forelesers implementasjon er den i snitt vel 4x raskere).
- Tidene du rapporterer (av å kjøre både VenstreRadix og Arrays.sort) skal være **medianen** av tidene du får ved å kjøre samme sortering 3 eller 5 ganger (hver gang selvsagt med nytt, usortert innhold i a[]). Dette fordi første gang du kjører en Java-metode får du langt høyere tider enn neste gang. (Hint: medianen finner du ved å samle tidene for en test, Innstikk-sortere dem og returnere midtelementet).
- Første gang oversettes koden til maskinkode fra bytekoden som er på .class-filen. Kjører man en metode mange ganger, optimaliseres denne maskinkoden ytterligere slik at det da går enda raskere.

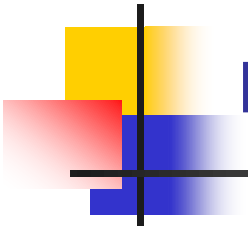
- Beskriv deretter med egne ord i rapporten om du kan si at VenstreRadix er en **stabil eller ustabil** sorteringsalgoritme og begrunn det.
- Tallene som skal sorteres, skal trekkes uniformt mellom 0 ... n-1 (og denne trekkingen holdes utenfor tidtakingen). Tidene du skal levere, beregnes som medianen av minst 3 (eller 5) kjøringer for hver verdi av n.
- Obliger i INF2220 innleveres i Devilry. Husk at det sammen med selve koden skal det ligge en rapport med tabell som beskrevet. Du **vil ikke få denne obligen godkjent uten denne rapporten** med forklaringer på kjøretidene som beskrevet ovenfor. Oblig 3 leveres individuelt og senest innen tirsdag 24. oktober kl. 23.59.

```
final static int NUM_BIT =9;   // eller: 6-13
final static int MIN_NUM = 31; // mellom 16 og 60

 double  VRadixMulti(int [] a) {
       long  tt = System.nanoTime();
       int [] b  = new int [a.length];

        // a) finn 'max' verdi i a[]

       // b) bestem numBit = mest venstre bit i 'max'
        //      som ==1

       // c) Første kall (rot-kallet)på VenstreRadix med
       //     a[], b[] , numBit, og lengden av første siffer

        double tid = (System.nanoTime() -tt)/1000000.0;
        testSort(a);
        return tid;       // returnerer sorterings tiden i ms.

  } // end VRadixMulti

  void testSort(int [] a){
        for (int i = 0; i< a.length-1;i++) {
            if (a[i] > a[i+1]){
                System.out.println("SorteringsFEIL på: "
                 + i +" a["+i+"]:"+a[i]+" > a["+(i+1)+ "]:"
                 + a[i+1]);
                 return;
                }
    } // end testSort
```

```
// Sorter a[left..right] på siffer med start i bit:
// leftSortBit, og  med lengde: maskLen  bit,

void VenstreRadix ( int [] a, int [] b, int left, int right,
                                        leftSortBit, int maskLen){
     int mask = (1<<maskLen]) -1;
     int [] count = new int [mask+1];

     ..... Andre deklarasjoner .....
    // d) count[] = antallene med de ulike  verdiene
    //       av   dette sifret I  a [left..right]

     // e) Tell opp verdiene i count[] slik at
     //      count[i] sier hvor i b[] vi skal flytte
     //      første element med verdien 'i'  vi sorterer.

      // f) Flytt tallene fra a[] til b[] sorter på
      //      dette sifferet I a[left..right] for
      //       alle de ulike verdiene for dette sifferet

      // g) Kall  enten innstikkSort eller rekursivt
      //     VenstreRadix på neste siffer (hvis vi ikke er
      //     ferdige)  for alle verdiene vi har av
       //    nåværende siffer

       //  Vurdér når vi skal kopiere tilbake b[] til a[]

 }// end VenstreRadix
```

Anta at vi bruker 3 bit i hvert siffer, dvs; verdiene 0,1,2,….., 7
Her er starten på kall-treet:





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 foreløpig gjennomgått
- JIT-kompilering og Cache-systemet
- Boble-sortering - elendig
- Innstikksorteing: raskest for $n < 50$
- HøyreRadix-sortering: Klart raskest når $n > 500$, men trenger mer plass (mer enn n ekstra plasser – flytter fra a[] til b[])
- Hvilken av sorterings-algoritmer er stabile?
  - Innstikk
  - Boble
  - Telle
  - HøyreRadix

  Svar: Innstikk, Boble og HøyreRadix er alle stabile,
  Telle er ikke definert og Quick (neste uke) er ikke stabil.