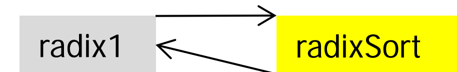
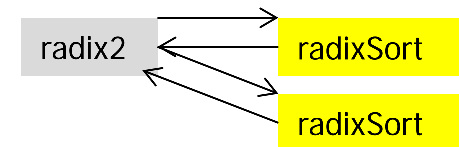
7) Radix-sortering sekvensielt – kode og effekten av cache

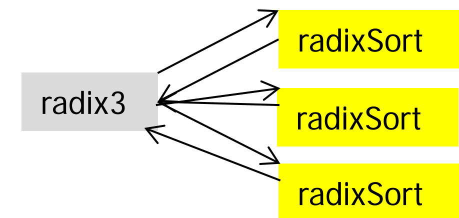
- Dels er denne gjennomgangen av vanlig Radix-sortering viktig for å forstå en senere parallell versjon.
- Dels viser den effekten vi akkurat så – tilfeldig oppslag i lageret med korte eller lange arrayer b[] i uttrykk som a[b[i]] kan gi uventede kjøretider.
- Ideen bak Radix er å sortere tall etter de ulike sifrene de består av og flytte de frem og tilbake mellom to arrayer a[] og b[] slik at de stadig blir sortert på ett siffer mer.

Om høyre, 'minst signifikant siffer først' Radix

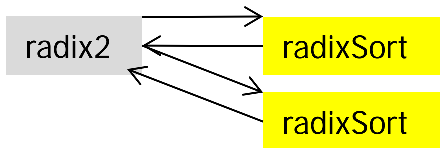
- Radix-sortering, her vist 3 varianter:
  - R1: Radix-sortering med ett siffer
  - R2: Radix-sortering med to sifre
  - R3: Radix-sortering med tre sifre
- Alle tre består av to metoder:
  - radix1,radix2 eller radix3 som
    - Først regner ut max-verdien i a[]. Så regnes ut noen konstanter som antall bit i det/de sifrene a[] skal sorteres med.
    - Deretter kalles metoden radixSort for hvert siffer det skal sorteres etter







Den første av to algoritmer som 2-siffer Radix består av.




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

   // finn max verdi i a[]
    for (int i = 1 ; i < n ; i++)
        if (a[i] > max) max = a[i];

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

    // bestem antall bit i siffer1 og siffer2
    int bit1 = numBit/2,
        bit2 = numBit-bit1;

    int[] b = new int [n];
    radixSort( a,b, bit1, 0);    // første siffer fra a[] til b[]
    radixSort( b,a, bit2, bit1);// andre siffer, tilbake fra b[] til a[]
 }
```

radix2 → radixSort
radixSort

```
/** Sort a[] on one digit ; number of bits = maskLen, shiftet up 'shift' bits */
static void radixSort ( int [] a, int [] b, int maskLen, int shift){
      int  acumVal = 0, j, n = a.length;
      int mask = (1<<maskLen) -1;
      int [] count = new int [mask+1];

     // a) count=the frequency of each radix value in a
      for (int i = 0; i < n; 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]>>shift) & mask]++] = a[i];

}// end radixSort
```

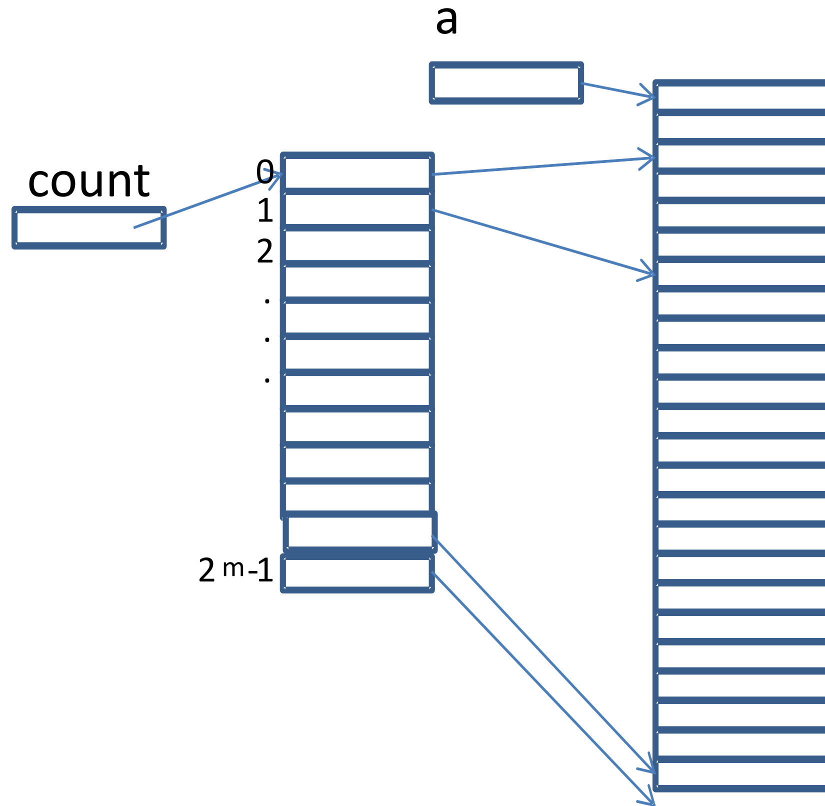
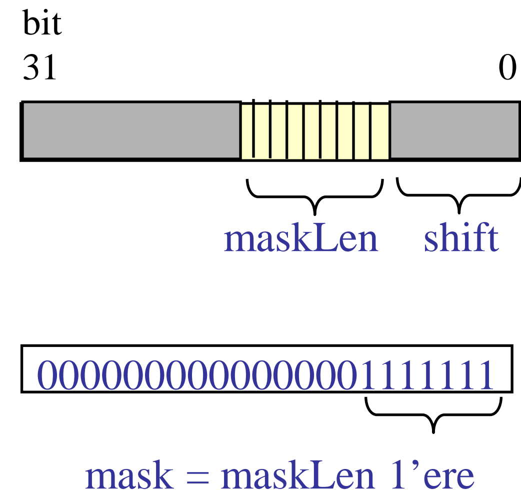


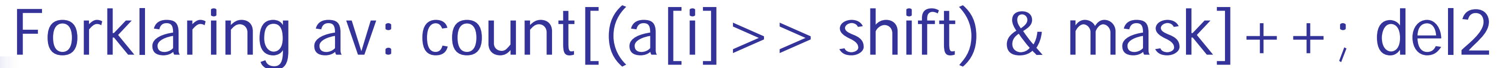
Figure 1. *The use of array count in any radix algorithm when sorting on a digit with* numbit *bits. The illustration is after sorting. We see that there are two elements in a[] with the value 0 on that digit, 4 elements with value 1,…,and 1 element with value* $2^{numbit}-1$.

Forklaring av: count[(a[i]>> shift) & mask]++; del1

- Tar det innenfra og ut; a[i]>> shift
  - Ethvert ord i lageret består a 0-ere og 1-ere (alt er binært)
  - Java har flere shift-operasjoner feks.:
    - a[i]>>b betyr: shift alle bit-ene i a[i] b antall plasser til høyre
      og fyll på med b stk 0-er på venstre del av a[i].
    - a[i]<<b betyr: shift alle bitene i a[i] b antall plasser til venstre
      og fyll på med b stk 0-er på høyre del av a[i].
    - De bit-ene som shiftes ut av a[i] går tapt i begge tilfeller.
    - a<<1 er det samme som a*2,
      a<<2 er det samme som a*4,
      a<<k er det samme som a*2^k



Forklaring av: count[(a[i]>> shift) & mask]++; del2



- Java har flere bit-logiske operasjoner, for eksempel & (og):
  - a & b er et tall som har 1-ere der **både** a og b har en 1-ere, og resten er 0.
  - Eks: a&1 = et tall som er null over alt unntatt i bit_0 som har samme bit-verdi som bit_0 i a.
  - Vi kan betrakte b som en maske som plukker ut de bit-verdiene i a hvor b har 1-ere.
- Poenget er at: (a[i]>> shift) & mask er raskeste måte å finne hvilken verdi a[] har for et gitt siffer (sifferverdien) :
  - Først skifter vi bit-ene i a[i] ned slik at sifferet vi er interessert i ligger helt nederst til høyre.
  - Så &-er vi med en maske som bare har 1-ere for så mange bit vi har i det sifferet vi er interessert i nederst (og 0 ellers).
- count[(a[i]>> shift) & mask] er da det elementet i count[] som har samme indeks som sifferverdien i a[i].
- Det elementet i count[] øker vi så med 1 (++ operasjonen)

Eksempel (shift = 3 og mask =7) – vi vil ha 2dre siffer

- a[i] = 764 (i 8-tallsystemet)= 0000..000111110100
- a[i] >>3 = 0000000..000111110
- (a[i] >>3) & 0000000..000000111 = 00000000..00110 =6

Vi kan velge fritt hvor lange (antall bit) sifre og hvor mange sifre vi vil ha sortere på, men summen av antall bit i sifrene vi sorterer på må være større eller lik antall bit i max, det største tallet i a[].

Et godt valg er å ha en øvre grense på bit-lengden av et siffer – f.eks = 11, og da heller ta så mange sifre det trengs for å sortere a[] .

Stegene i en radix-sortering

radix1,2,eller3:

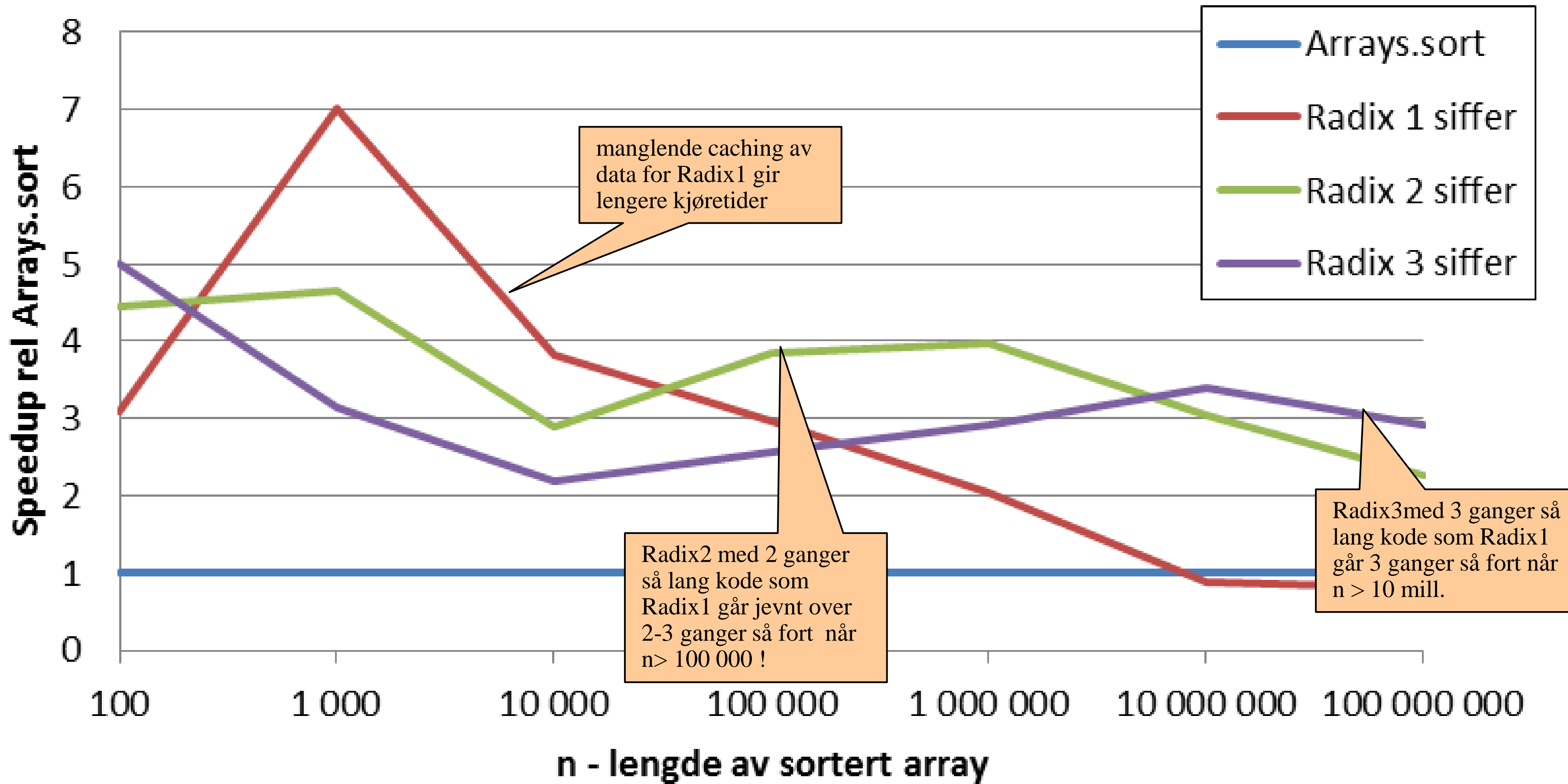
- Finn maks verdi i a[] og bestem antall sifre med mer.
  - FinnMax har vi parallellisert

radixSort (en gang for hvert siffer):

a) Tell opp hvor mange det er i a[] med de ulike mulige sifferverdiene på dette sifferet.
b) Adder sammen verdiene til en array som sier hvor vi skal flytte et element i a[] med en gitt sifferverdi.
c) Flytt elementene fra a[] til b[] slik at de minste verdier kommer øverst,..osv
d) Kopier b[] tilbake til a[] (trenges ikke i radix2,radix4,..)

Stegene a, b og c skal vi senere parallellisere (d kan fjernes)

Hva viser dette om cachen ?



Radix-sortering– den sekvensielle algoritmen

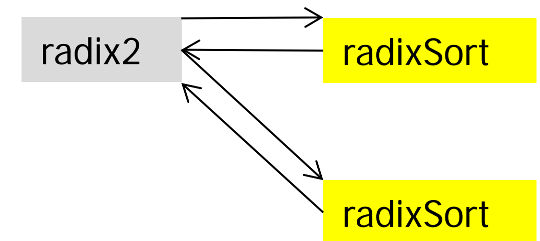
- Vi aksepterer at vi forrige gang greide å finne verdien av et siffer i a[]:
  - (a[i]>> shift) & mask – regner ut sifferverdien av et siffer i a[i] som :
    - Har ett eller flere sifre til høyre for seg (mindre signifikante) som til sammen i sum har 'shift' bit
    - Mask inneholder så mange 1-ere nederst som det er bit i det sifferet vi vil finne nå – og er ellers 0.

- Anta at vi skal sortere denne a[] på to sifre,

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

Høyre, 'minst signifikant siffer først' sortering på to sifre: Radix2 som vi nå bruker

- Radix-sortering, nå 2 siffer:
  - Radix2: Radix-sortering på to sifre
- Radix 2 bestås av to metoder:
  - radix2 som først regner ut max-verdien i a[]. Så regnes ut noen konstanter, som antall bit i de to sifrene a[] skal sorteres med.
  - Deretter kalles metoden radixSort for hvert av de to sifrene (dvs. to ganger)



Stegene i en radixSort:

a) Tell opp i en array count slik at count[k] = hvor mange ganger k er en sifferverdi a[].
   - Eks. hvor mange tall i a[] = 0 i dette sifferet ?

b) Legg sammen antallene i count slik at count[k] sier hvor i b[] vi skal plassere første element i a[] vi finner med sifferverdien 'k'

c) Finn sifferverdien i a[k] og flytt a[k] til b[] der count[sifferverdien] sier a[k] skal være.
   Øk count[sifferverdien] med 1 til neste plass i b[]

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[]:

a

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

Før telling:

count

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

Etter telling:

count

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

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[]:

Før addering:

count

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

Etter addering :

count

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

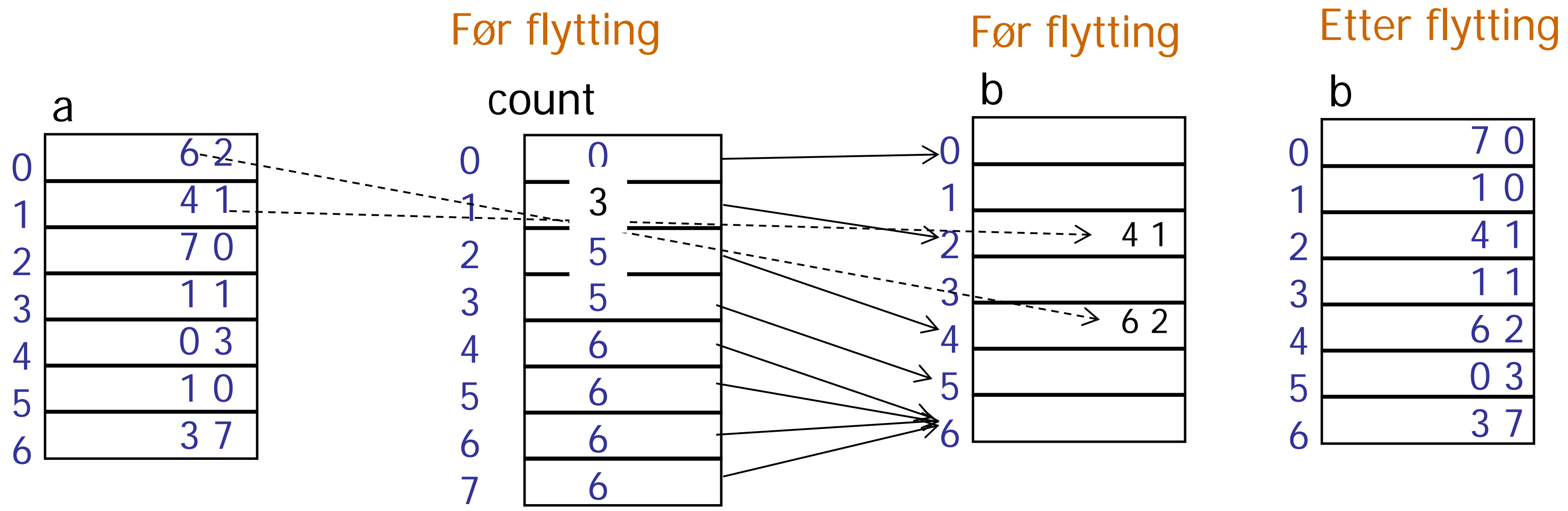
b

0	
1	
2	
3	
4	
5	
6	

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 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)

b

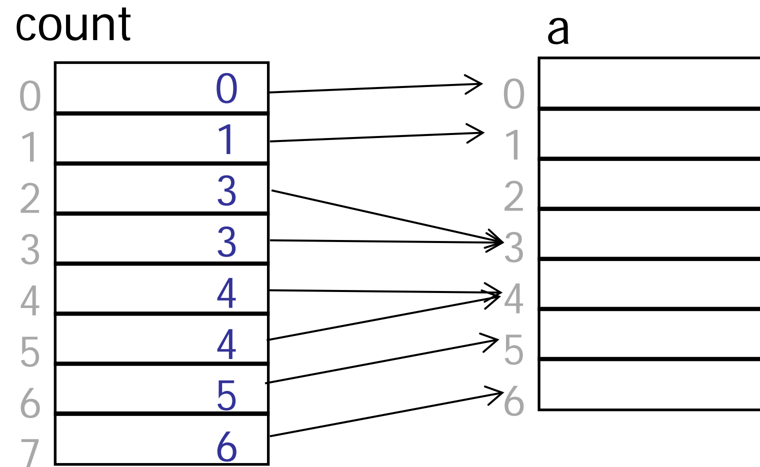
0	7 0
1	1 0
2	4 1
3	1 1
4	6 2
5	0 3
6	3 7

Etter telling på siffer 2:

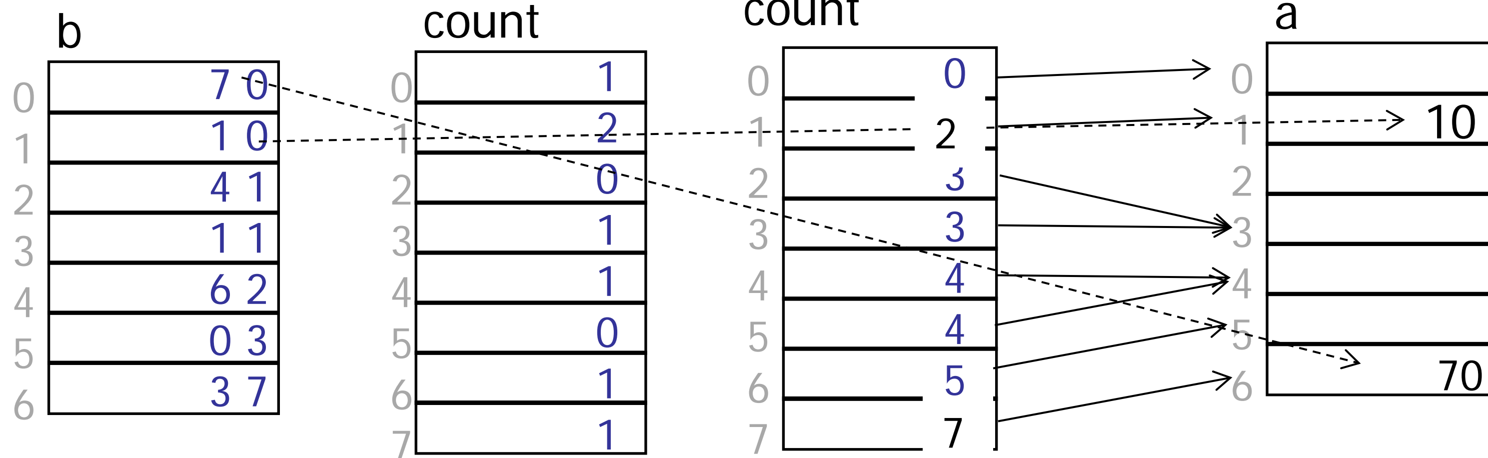
count

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

Etter addering :



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

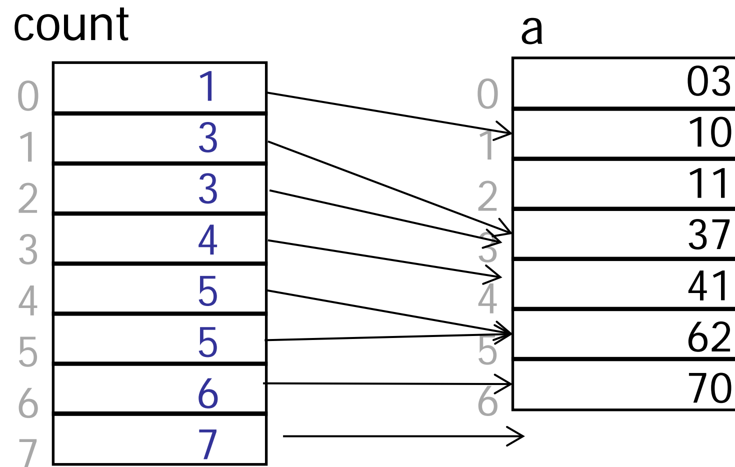


Situasjonen etter sortering fra b[] til a[] på siffer 2

Etter flytteing

b

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



a[] er sortert !