INF2220 – høsten 2017, 19. okt.

Sortering (kap. 7.) – sekvensiell sortering II

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

Hva lærte vi for en uke siden

- Definerte sortering
  - Verdibaserte og sammenligningsbaserte alg.
  - Stabile og ikke-stabile algoritmer
  - Hastighet, velegnet for alle typer data og plassbehov
- 5 algoritmer
  - Boble (dårlig)
  - Innstikk(best for n < 30-500)
  - TelleSort (rask, men ubrukelig)
  - HøyreRadix (best), stabil men dobbelt plass
  - VenstreRadix (best), men dobbelt plass
- Så på Oblig 3

Dagens forelesning

- Forstå: **b[count[(a[i]>>shift) & mask]++] = a[i];**
- Se på flere sorteringsalgoritmer:
  - Maxsort
  - Heap og Tree sort
  - Shellsort
  - Kvikksort
  - Flettesort
- Oppsummering om sekvensiell sortering

Forstå: b[count[(a[i]>>shift) & mask]++] = a[i];

Shift –operasjonene (>> og <<) og & - operasjonen

(a[i]>>shift) & mask

Leser a[i] og flytter alle bit høyre-over 'shift' antall plasser slik at det sifferet vi er interessert i ligger 'nederst'
(endrer **ikke** a[i])

x & y = et tall som har 1 der både **x og y** har 1, og 0 ellers.

k

0100000000000001001111

k >> 2

0001000000000000000100111 1

Det er to skift operasjoner:

- Shift høyre-over (eks. 0100011 >>2 gir 0001000) –
  fyller på med 0 på toppen, det som skyves ut til høyre er tapt
- Shivt venstre-over (eks. 00011 << 3 gir 011000) –
  fyller på med 0 i bunn, nederst, det som skyves ut til vestre er tapt

Del2: b[count[(a[i]>>shift) & mask]++] = a[i];

eksempel: 7 8

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

Antall bit i dette sifferet (her 7)

Summen av bit-lengdene for alle sifre til høyre for dette siffer = 8

```
int mask = (1<<maskLen) -1;  // lager et tall som nå har 7  stk. 1-ere
                             // nederst (til  høyre) ,og  0-fyllt ellers
```

1<<maskLen = $2^{maskLen}$

1<< 1 (= 0..00000010) = 2^1 = 2, 2-1 = 001

1<< 2 (=0..00000100) = 2^2 = 4, 4-1 = 011

……….

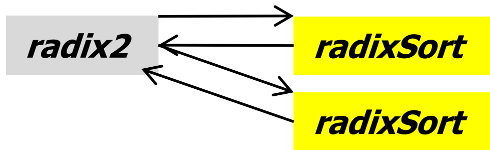
1<< 7 (= 0..10000000) = 2^7 = 256, 256 -1 = 0…01111111

7 ettall nederst, 0 ellers

Del 3: b[count[(a[i]>>shift) & mask]++] = a[i];

- Innenfra og ut , tolkningen
  - Sifferverdien av nåværende siffer i a[i]:
    - (a[i]>>shift) & mask
  - Hvor i b[] skal vi skal plassere a[i] ?
    - Jo, der count[sifferverdien] peker
      - Husk å øke count[sifferverdien] med 1 etterpå fordi neste element i a[] med same sifferverdi må komme på neste plass

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



1<<numBit

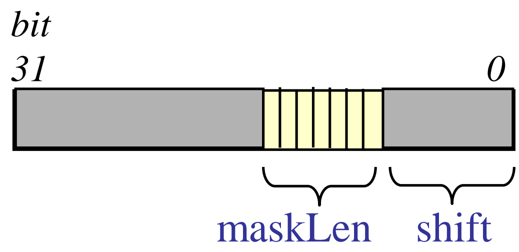
Betyr at vi skifter tallet 1 ‘*numbit*’ plasser oppover i en int.

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

Innledende kode for å radix-sortere på to siffer:

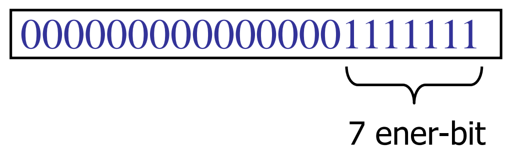
```
static void radix2(int [] a) {
    // 2 digit radixSort: a[ ]
    int max = 0, numBit = 2;

    // 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 her 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 = 0; i < a.length; 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 */
```

Sifferverdien til a[i] – tell opp antall

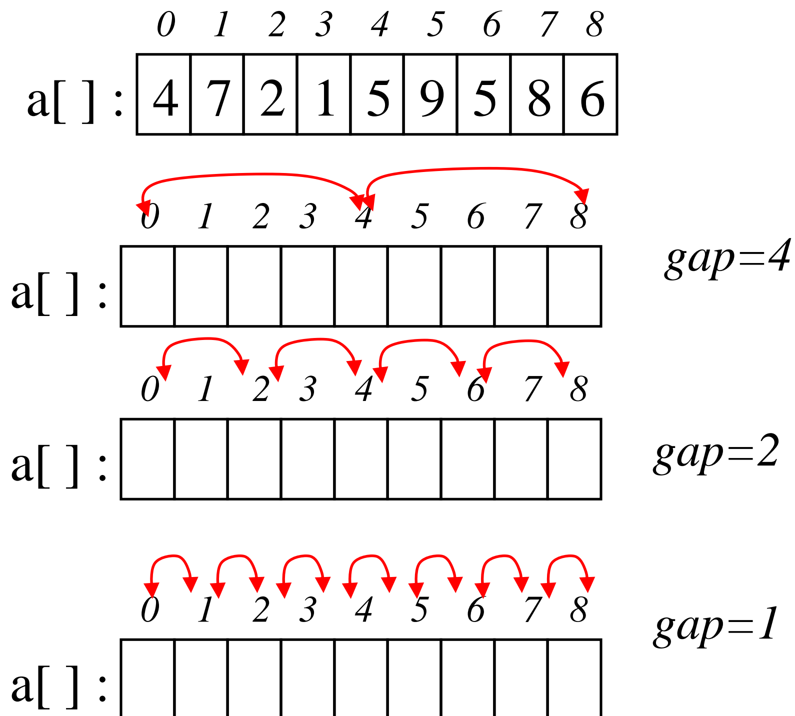
Hvor plassere denne sifferverdien til a[i] i b[]

Shell-sortering, forbedring av Innstikksort

Hva val 'galt' med Innstikk sortering :
Skyving av små tall lange avstander i a[] ; ett element av gangen.

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

```
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
```



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

Shell2 – en annen sekvens for gap

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

     for (int gapInd = gapVal.length -1; gapInd >= 0; gapInd --) {
                          gap = gapVal[gapInd];
                          for (int i = gap ; i < a.length ; i++)
                                   if (a[i] < a[i-gap] ) {
                                             int tmp = a[i],
                                                        j = i;
                                             do
                                             {  a[j] = a[j-gap];
                                                        j = j- gap;
                                             }  while (j >= gap && a[j-gap] > tmp);

                                             a[j] = tmp;
                                   }
              }
 } // 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:1 048 576 = 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 = 213**

Heap - sort	=	13
Shell-sort	=	22
Shell 2 -sort	=	11
Tree - sort	=	11

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

MaxSort – enkel og langsom ($O(n^2)$)

Ide: Finn største elementet i a[0..n-1]. Bytt det med a[n-1].
Gjenta dette for a[0..n-2],a[0..n-3], …, a[0..1] . Ferdig!

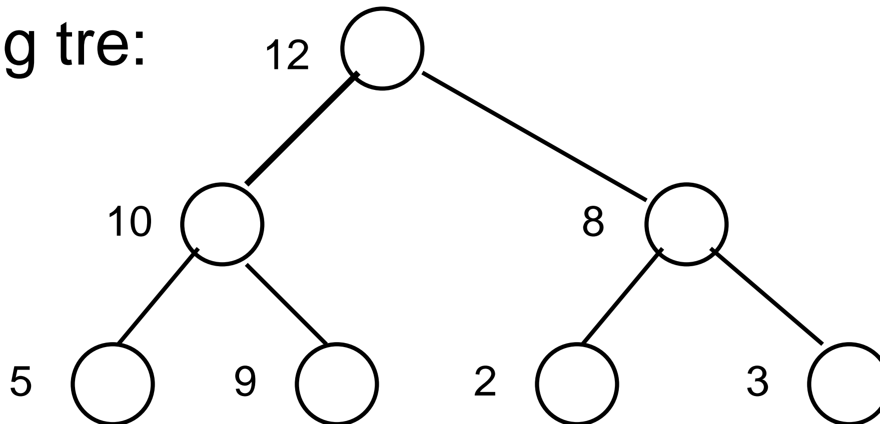
```
void  maxSort ( int [] a) {
     int max, maxi;

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

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

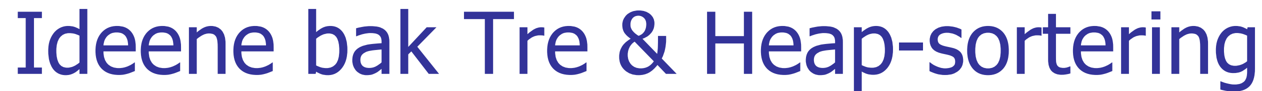
Rotrettet tre (heap)

- 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:

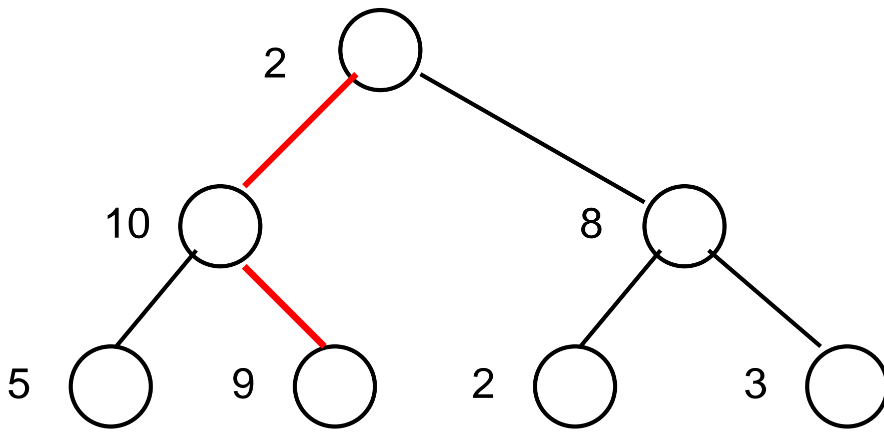


Ideene bak Tre & Heap-sortering



- 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]

Feil i rota, ’2’ er ikke størst:



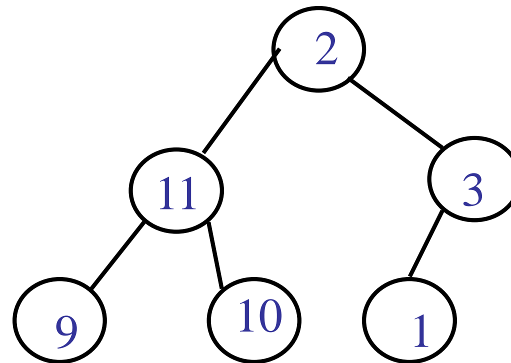
	0	*..*	*i*			*2i,*	*2i+1*
a[ ] :	**2**	**10**	**8**	**5**	**9**	**2**	**3**

0	*..*	*i*			*2i,*	*2i+1*
10	**9**	8	5	**2**	2	3

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

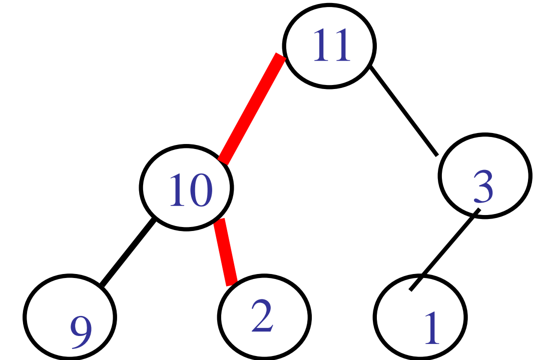
```
static void dyttNed (int i, int n) {
   // Rota i a[i] er (muligens) feilplassert – dytt 'gammel og liten' rot nedover
   //  få ny, større oppover, n = øverste indeks i a[] vi nå bruker
   int j = 2*i+1, temp = a[i];

     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

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

a

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

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 $\geq$ log(n))

Tre sortering

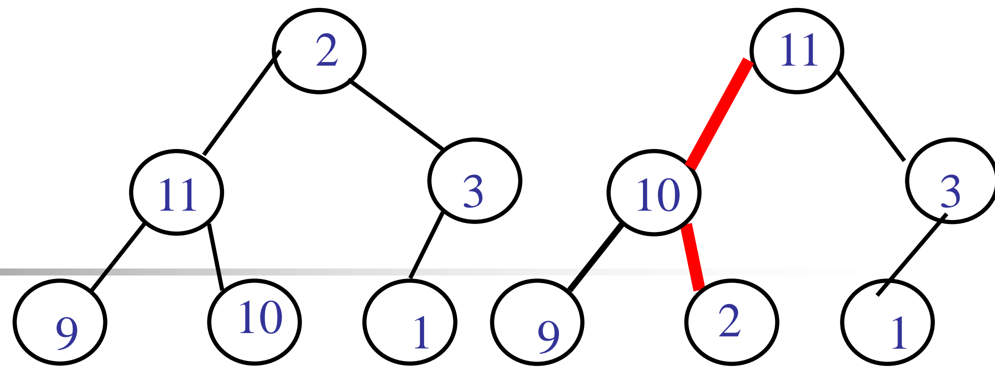
```
void dyttNed (int i, int n) {
 //  Rota er (muligens) feilplassert
//   Dytt gammel nedover
//   få ny større oppover
int j = 2*i+1, temp = a[i];
   while(j <= n )
   {   if ( j < n && a[j+1] > a[j] ) j++;
        if (a[j] > temp) {
                a[i] = a[j];
                 i = j;
          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

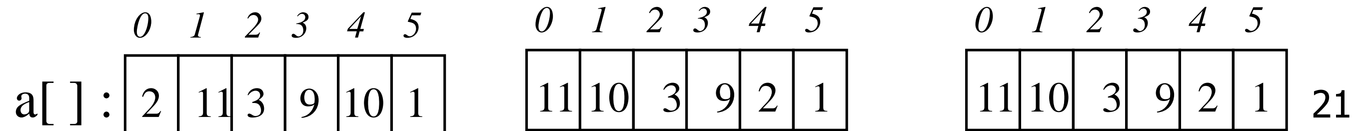
Tre sortering




```
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);
   }
 }
```

***Idé**: 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 gjennomsnittshøyde = (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)

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 =	1,00

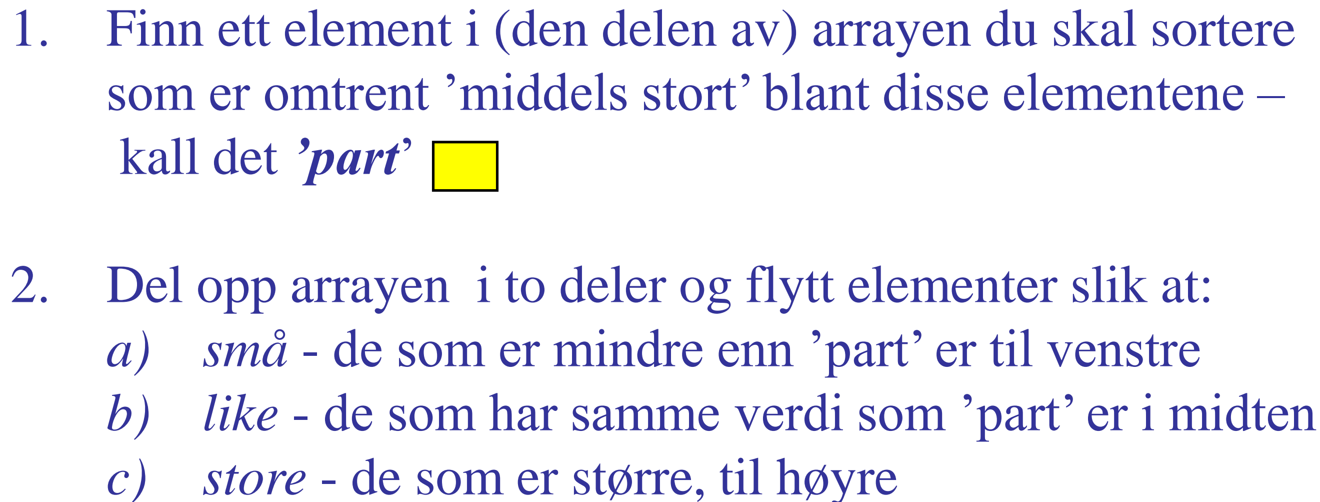
n =	**1 mill**
Boble-sort =	627 800,
Innstikk-sort =	130 800,
Tree - sort =	130,8
Quick-sort =	97,3

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 mill**
Boble-sort =	---?? ----,
Innstikk-sort =	---?? ----,
Tree - sort =	2301
Quick-sort =	1136

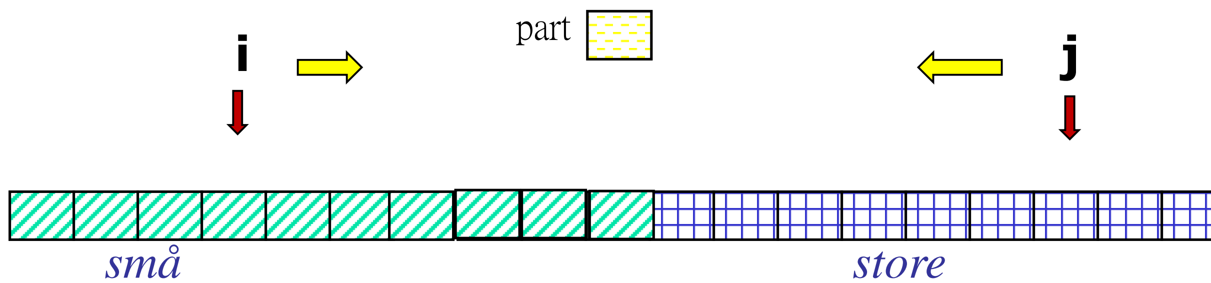
Quicksort – generell idé



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

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.

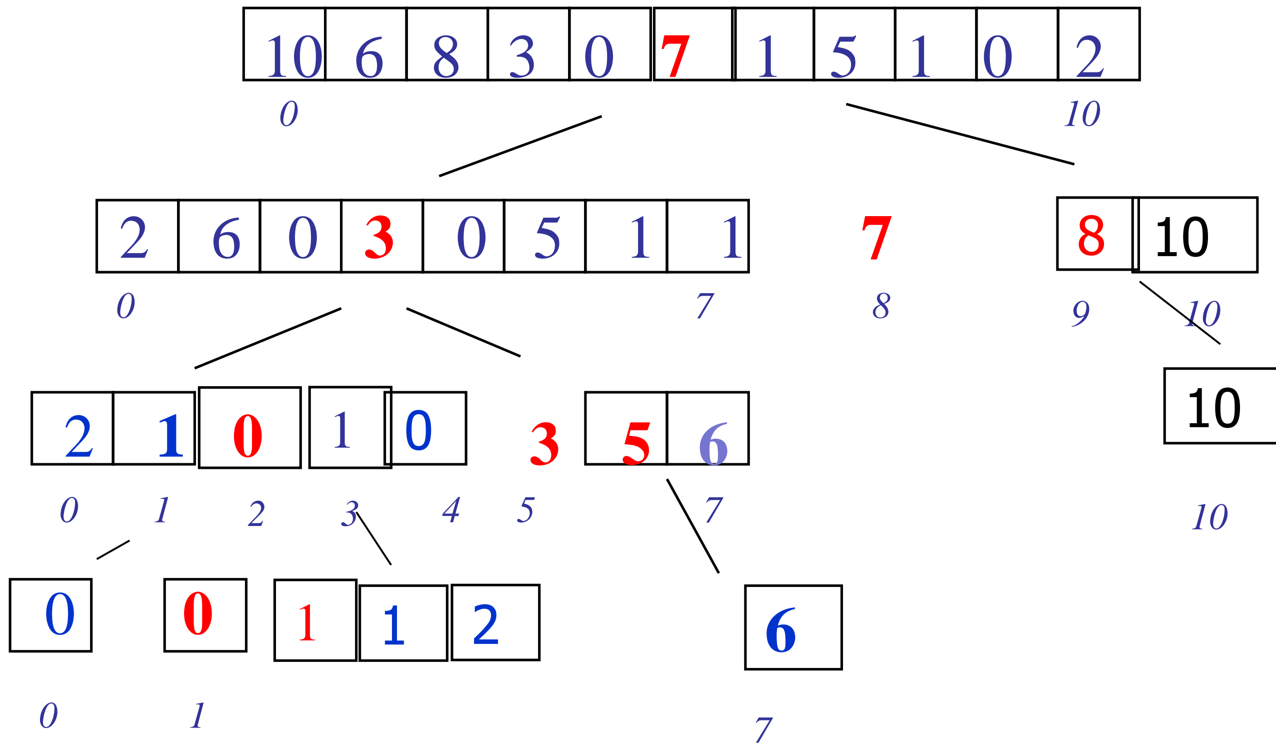


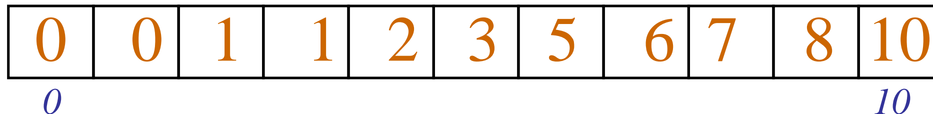

```
void  quicksort ( int [] a, int left, int right)
  { int i= l, j=r;
     int t, part = a[(left+right)/2];

      while ( i <= j) {
            while ( a[i] <  part ) i++;  //hopp forbi små
            while (part < a[j]  )  j--;   // hopp forbi store

            if (i <= j) {
               // swap en for liten a[j] med for stor a[i]
               t    = a[j];
               a[j]= a[i];
               a[i]= t;

                i++;
                j--;
            }
     }
     if ( left < j ) {   quicksort (a,left,j); }
     if ( i < right ) {   quicksort (a,i,right);  }
} // end quickSort
```

Kall:

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

QuickSort – eksempel (litt forenklet for sortering av 2 elementer)





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)

Tider , Intel Core i7 , 2(4) kjerner, 2,7GHz – okt 2017

N: 50, median av: 10 000

Quick - sort:: 0.00000 ms.
Tree - sort:: 0.00000 ms.
Heap - sort:: 0.00000 ms.
Hradix - sort:: 0.00038 ms.
Insert -sort:: 0.00000 ms.
MaxSort :: 0.00038 ms.

N: 500, median av: 1000

Quick - sort:: 0.00304 ms.
Tree - sort:: 0.00266 ms.
Heap - sort:: 0.00266 ms.
Hradix - sort:: 0.00456 ms.
Insert -sort:: 0.00190 ms.
MaxSort :: 0.00266 ms.

N: 5000, median av: 100

Quick - sort:: 0.57476 ms.
Tree - sort:: 0.41016 ms.
Heap - sort:: 0.39610 ms.
Hradix - sort:: 0.16612 ms.
Insert -sort:: 3.09239 ms.
MaxSort :: 5.06908 ms

N: 50 000, median av: 10

Quick - sort:: 5.63 ms.
Tree - sort:: 4.56 ms.
Heap - sort:: 5.97 ms.
Hradix - sort:: 1.82 ms.
Insert -sort:: 312.96 ms.
MaxSort :: 468.31 ms.

N: 500 000, median av: 5

Quick - sort:: 39.4 ms
Tree - sort:: 67.1 ms
Heap - sort:: 62.7 ms
Hradix - sort:: 7.9 ms.

N: 5mill., median av: 5

Quick - sort:: 494 ms.
Tree - sort:: 1099 ms.
Heap - sort:: 1100 ms.
Hradix - sort:: 94 ms.

.

N: 50 mill, median av: 5

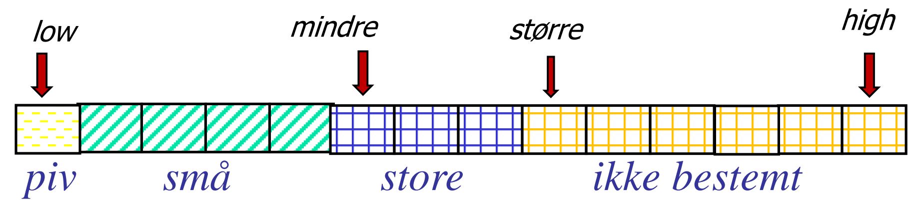
Quick - sort:: 5 549 ms.
Tree - sort:: 17 777 ms.
Heap - sort:: 17 489 ms.
HRadix-sort :: 784 ms.

N: 500 mill, median av: 5

Quick - sort:: 68 333 ms
Tree - sort:: 286 351 ms.
Heap - sort:: 317 726 ms.
HRadix-sort : 31 175 ms.

- **Hvorfor 100x fra 500 til 5000?**
- **Quick stiger mer enn O(n)**
- **Radix stiger omlag O(n) ??**

En helt annen koding av Quicksort (ganske rask) etter Lamoto:




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

Fordeler med Lamotos versjon

- Meget enklere å få riktig
  - Bare ett ulikhetstegn , mot 4 i O-J Dahl formulering
- Litt langsommere, men lett optimaliseres ved å legg inn flg. linjer hvis det er flere like elementer i a[]:

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

- Hvor legges disse inn
- + litt endriner i ett av de rekursive kallene

Flette - sortering (merge)

Velegnet for sortering av filer og data i hukommelsen.

Generell idé:

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

I praksis skal det meget store filer til, før du bruker 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;}

}
```

Stabil eller ikke stabil (kommer like verdier ut i samme rekkefølge som på input)

- Hvilken av følgende sorterings-algoritmer er stabile?
  - Innstikk
    - Stabil
  - Quick
    - IKKE stabil
  - HRadix
    - Stabil
  - VRadix
    - Du svarer selv i Oblig3 med begrunnelse
  - TreSort
    - Vel ikke Stabil.
  - MaxSort
    - Nei, avhengig av if (a[i] > max) eller if (a[i] >= max)
  - Flette
    - IKKE nødvendigvis Stabil

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
  - Boblesort : bare dårlig (langsomst)
  - Innstikksort: raskest for $n < 0 - 50$
  - Maxsort – langsom, men er et grunnlag for Heap og Tre
  - Tre-sortering: Interessant og ganske rask : O(n logn)
  - Quick: rask på middelstore datamengder (ca. $n = 50$ -5000)
  - Radix-sortering: Klart raskest når $n > 500$, men HøyreRadix trenger mer plass (mer enn n ekstra plasser – flytter fra a[] til b[] + count[])