Uke 12,
Forelesning 1

Sortering:

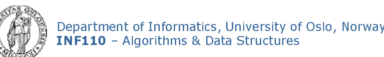
Sammenligning-baserte:
Baserer seg på sammenligning av elemntene i a[ ]

- Instikk, boble, utplukk
- Merge, Heap, Shell, Tree

Quicksort – avslutter i dag, fortsetter med
verdi-baserte algoritmer

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

- Bøtte
- Radix
- PSort

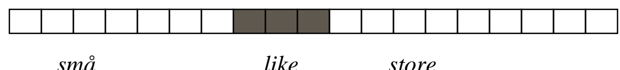


Fortsetter med sortering…

Quicksort – generell idé

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

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



små *like* *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.

Quicksort i praksis I

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

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

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

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

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

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

} // end quickSort
```

Quicksort, tidsforbruk

Vi ser at ett **gjennomløp av** quickSort **tar O(r-l) tid, og første gjennomløp** O(n) **tid fordi r-l = n første gang**

Verste tilfellet

Vi velger 'part' slik at det f.eks. er det største elementet hver gang. Da får vi totalt n kall på quickSort **, som hver tar O(n/2) tid i gj.snitt – dvs** $O(n^2)$ **totalt**

Beste tilfellet

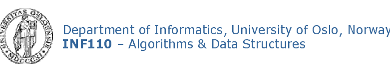
Vi velger 'part' slik at den deler arrayen i to like store deler hver gang. Treet av rekursjons-kall får dybde log n**. På hvert av disse nivåene gjennomløper vi alle elementene (høyst) en gang – dvs:**

O(n) + O(n) + … +O(n) **=** O (n log n)

(log n ledd i addisjonen)

Gjennomsnitt

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



Quicksort analysis

Quick Sort Analysis

This is more difficult than Merge Sort. The reason is that in Merge Sort we always knew we were getting recursive calls with equal sized inputs. But in Quick Sort, each recursive call could have a different sized set of numbers to sort. Here are the three analyses we must do:

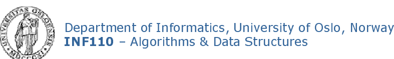
1) Best case
2) Average case
3) Worst case

Quicksort, best case

In the best case, we get a perfect partition every time. If we let T(n) be the running time of Quick Sorting n elements, then we get:

$T(n) = 2T(n/2) + O(n)$, since partition runs in $O(n)$ time.

This is the same exact recurrence relation as we got from analyzing Merge Sort. Just like that situation, here we find that in the ideal case, QuickSort runs in $O(n\log n)$ time.



Quicksort, average case

Assume that you run Quick Sort n times. In doing so, since there are n possible partitions, each equally likely, on average, we have each partition occur once. So we have the following recurrence relation:

$nT(n) = T(0)+T(n-1)+T(1)+T(n-2)+...+T(n-1)+T(0) + n*n$
$nT(n) = 2[T(1)+T(2)+...T(n-1)] + n^2$

(The n is for the work done by the partition method, simplified from O(n) to make the analysis easier.)

Now, plug in n-1 in the equation above to get the following one:

$(n-1)T(n-1) = 2[T(1)+T(2)+...T(n-2)] + (n-1)^2$

Quicksort, average case

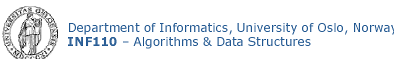
Subtracting these two equations we get:

$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1$
$nT(n) = (n+1)T(n-1) + (2n - 1)$
$T(n) = [(n+1)/n]T(n-1) + (2n - 1)/n$

Since we are only trying to do an approximate analysis, we will drop the -1 at the end of this equation. Dividing by n+1 yields:

$T(n)/(n+1) = T(n-1)/n + 2/(n+1)$

Now, plug in different values of n into this recurrence to form several equations:



Quicksort, average case

$T(n)/(n+1) = T(n-1)/n + 2/(n+1)$
$T(n-1)/(n) = T(n-2)/(n-1) + 2/(n)$
$T(n-2)/(n-1) = T(n-3)/(n-2) + 2/(n-1)$
...
$T(2)/3 = T(1)/2 + 2/1$

Now, adding all of these equations up reveals many identical terms on both sides. In fact, after cancelling identical terms, we are left with:

$T(n)/(n+1) = T(1)/2 + 2[1/1 + 1/2 + 1/3 + ... + 1/(n+1)]$

The sum on the right hand side of the equation is a harmonic number. The nth harmonic number(H_n) is defined as 1 + 1/2 + 1/3 + ... 1/n.

Quicksort, average case

Through some calculus, it can be shown that $H_n \sim \ln n$. (ln is the natural log. It is a logarithm with the base e. e ~ 2.718282.)

Now, we have:

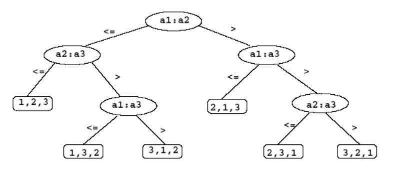
$T(n)/(n+1) \sim 1/2 + 2\ln n$
$T(n) \sim n(\ln n)$, simplifying a bit.

Thus, even in the average case for Quick Sort, we find that $T(n) = O(n \log n)$.

Lower bound on sorting based on comparisons

We will now discuss the Lower bound on a problem:

http://home.wlu.edu/~vermeerp/Classes/211w99/Lectures/Ch8_9/sld003.htm

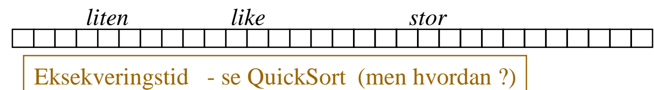


Finne det k' største elementet

Generell idé:

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

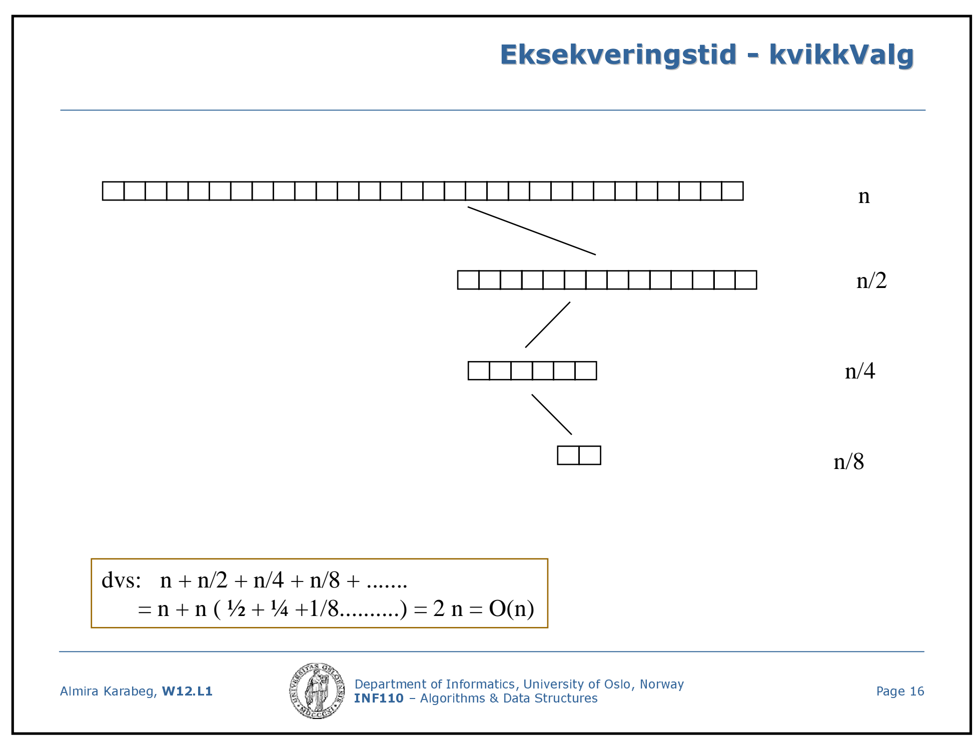
   if (k <= (lengden av 'liten)) let i 'liten' **else**
   if (k <= (lengden av lite + like) funnet i 'like' **else**
   let videre i 'stor'



Eksekveringstid - se QuickSort (men hvordan ?)

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

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

Verdi-baserte sorteringsmetoder

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

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

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

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

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


Bøtte-sortering og sortering av objekter

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

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

 s = bucketSort( s );
…
```

s

list

```
BNode  bucketSort ( BNode s )
   {int max =0;
    BNode t =s;
    while (t!= null) {
       if (t.value >max ) max = t.value;
          t = t.next;
     }
     BNode [] list = new BNode [max+1];
     BNode t;

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


Radix-sortering

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

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

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

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

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

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


Ett element i   fra[]:

bit 31 — bit — 0

numBit

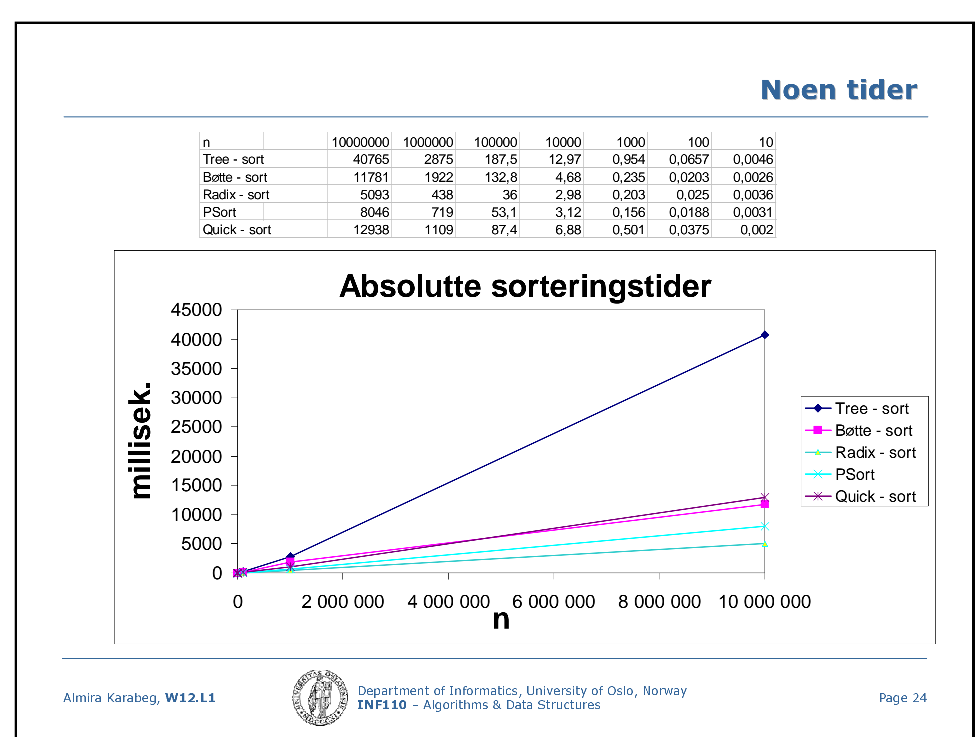
rMax:

000000000001111111100000

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

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

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

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

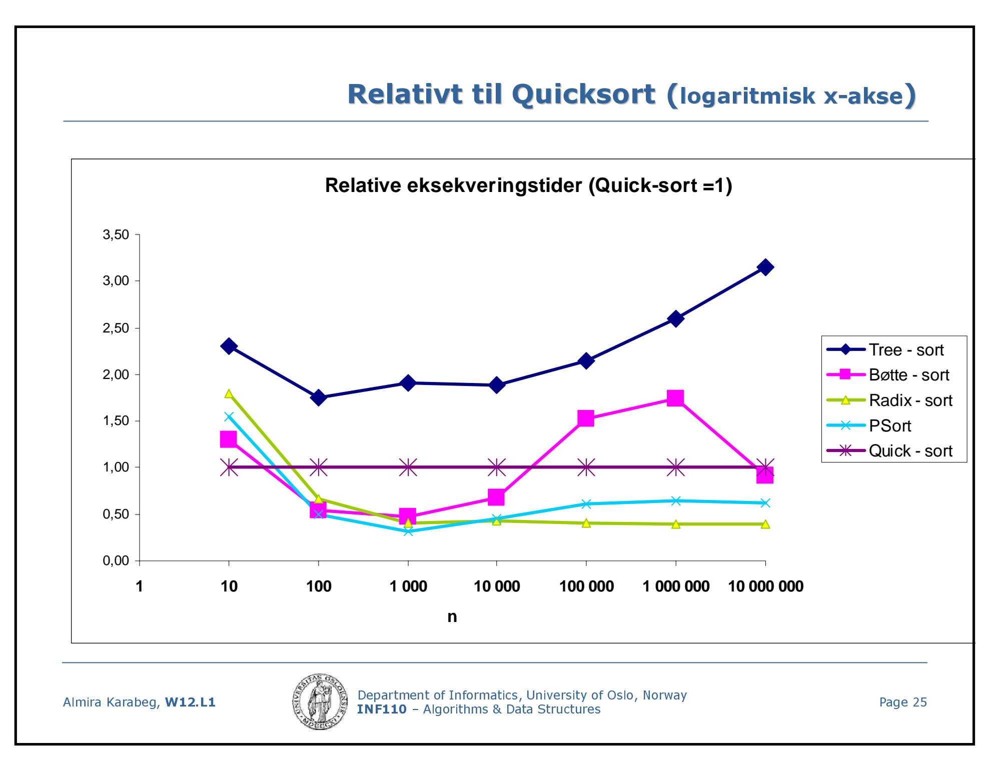
Tidsforbruket til Radix

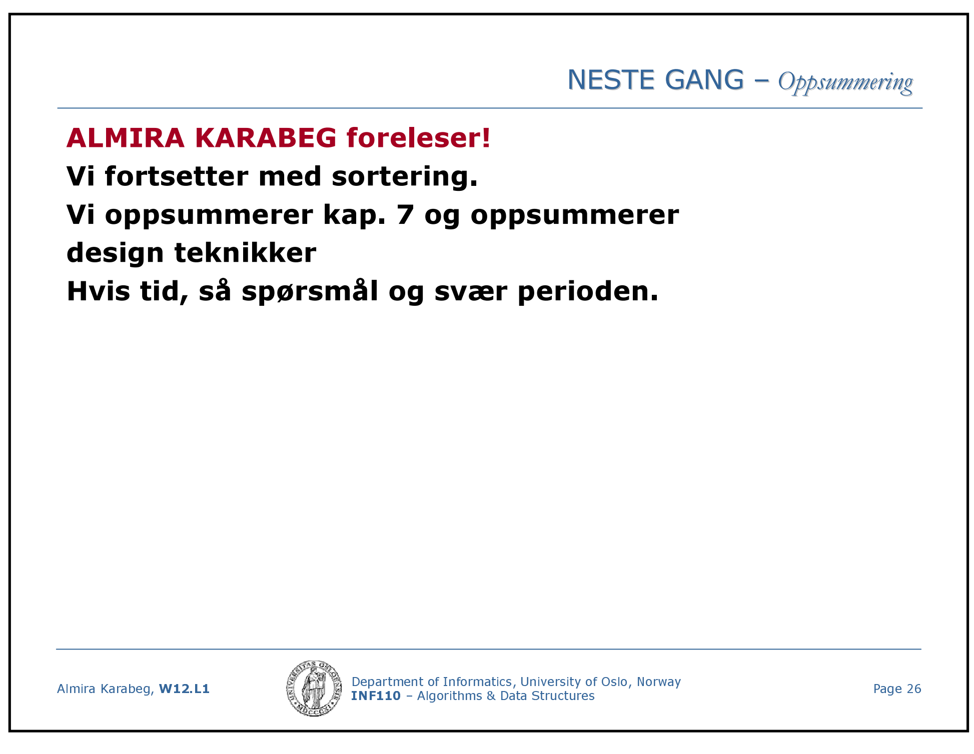
1. Først går vi gjennom a[] for å finne max = O(n)
2. Legg merke til at ant[] har lengde log rMax = 10 – en konstant
3. Følgende operasjoner gjøres ⌈log max/log rMax⌉ ganger:
    1. Tell opp i ant[] hvor mange det er av hver siffer-verdi = O(n)
    2. Juster pekere i ant[] = O(log rMax)
    3. Flytte alle n data (fra: fra[] til: til[]) = O(n)
4. Dvs Totalt O(n log max), og siden max ofte er en funksjon av n, blir dette ofte O(n logn) – men koeffisienten er meget liten



Noen tider

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


Relativt til Quicksort (logaritmisk x-akse)
Relative eksekveringstider (Quick-sort =1)
3,50
3,00
2,50
2,00
1,50
1,00
0,50
0,00
1
10
100
1 000
10 000
100 000
1 000 000
10 000 000
n
Tree - sort
Bøtte - sort
Radix - sort
PSort
Quick - sort
Almira Karabeg, W12.L1
Department of Informatics, University of Oslo, Norway
INF110 – Algorithms & Data Structures
Page 25


NESTE GANG – Oppsummering
ALMIRA KARABEG foreleser!
Vi fortsetter med sortering.
Vi oppsummerer kap. 7 og oppsummerer
design teknikker
Hvis tid, så spørsmål og svær perioden.
Almira Karabeg, W12.L1
Department of Informatics, University of Oslo, Norway
INF110 – Algorithms & Data Structures
Page 26