

Uke 11, Forelesning 2



Sortering:

Sammenligning-baserte:

Baserer seg på sammenligning av
elementene i $a[]$

Eksempler:

Instikk, boble, utplukk
Alle tar kvadratisk tid





*Fortsetter med
sortering...*

To klasser av sorteringsalgoritmer

- **Sammenligning-baserte:**
Baserer seg på sammenligning av elementene i $a[]$
 - Instikk, boble, utplukk
 - Merge, Heap, Shell, Tree
 - Quicksort
- **Verdi-baserte :**
Direkte plassering basert på verdien av hvert element – ingen sammenligninger med nabo-elementer e.l.
 - Bøtte
 - Radix
 - PSort

Shellsort

```
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
```

Ide: Gjør essensielt innstikksortering langs $a[i], a[i-gap], a[i-2gap]$... for $gap = n/2, n/4, \dots, 1$ og $i = gap, gap+1, \dots, n-1$. Dvs. alle sekvenser i $a[]$ av lengde $\dots, n/2, n/4, \dots, 2$ og til sist 1

0 1 2 3 4 5 6 7 8
a [] :

4	7	2	1	5	9	5	8	6
---	---	---	---	---	---	---	---	---

0 1 2 3 4 5 6 7 8
a [] :

--	--	--	--	--	--	--	--	--

 gap=4

0 1 2 3 4 5 6 7 8
a [] :

--	--	--	--	--	--	--	--	--

 gap=2

0 1 2 3 4 5 6 7 8
a [] :

--	--	--	--	--	--	--	--	--

 gap=1



Analyse av Shellsort

- 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 innstikksortering 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 + $n/(\text{på de samme primtallene})$:
(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 annensekvens 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: 1048576 = 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 = 2**13	
Heap - sort	= 13
Shell-sort	= 22
Shell 2 -sort	= 11
Tree - sort	= 11

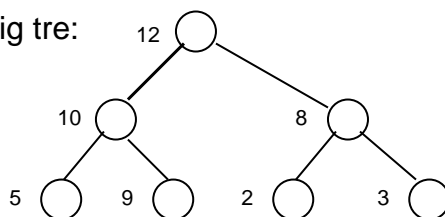


Rotrettet tre

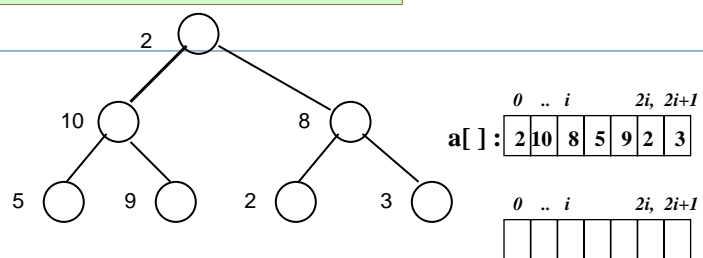
Ide for Heap & Tre sortering – rotrettet tre i arrayen:

- Rota er største element i treet (også i rota i alle subtrær – rekursivt)
- Det er ingen ordning mellom vsub og hsub (hvem som er størst)
- Vi betrakter innholdet av en array $a[0:n-1]$ slik at vsub og hsub til element 'i' er $2i+1$ og $2i+2$ (Hvis vi ikke går ut over arrayen)

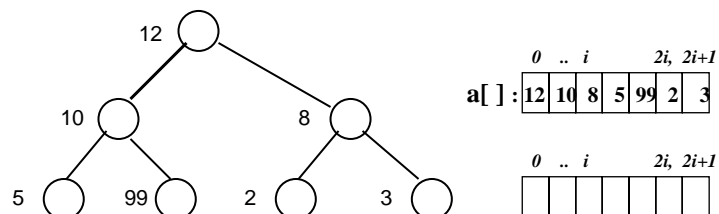
Eks på riktig tre:



Feil i rota, '2' er ikke størst:



Feil i bladnode, '99' er større enn sin rot:



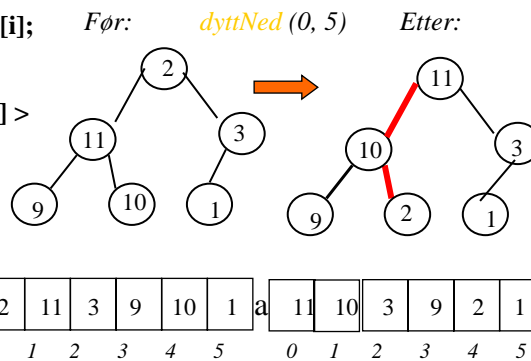
Hjelpemetode 1 – roten i et (sub)tre muligens feil :

```

static 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;
}

```



Eksekveringstider for dyttNed

```

static void dyttNed (int i, int n)
{ 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;
}

```

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 $j \leq n$:
dvs. while-løkkka går maks. $\log(n-i)$ ganger = $O(\log n)$
(dette er det samme som at høyden i et binærtre er $\log(n)$)



```

static void dyttOpp (int i)
// Bladnoden på plass i er
(muligens) feilplassert
// dytt den oppover mot rota
til hele treet
{ 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;
}

```

Før:

a

22	11	3	9	10	55
0	1	2	3	4	5

dyttOpp (5)

Etter:

a

55	11	22	9	10	3
0	1	2	3	4	5

Almira Karabeg, W11.L2

 Department of Informatics, University of Oslo, Norway
 INF110 – Algorithms & Data Structures

Page 13

Eksekveringstider for dyttOpp

```

static void dyttOpp (int i)
// Bladnoden på plass i er (muligens)
feilplassert
// dytt den oppover mot rota til hele treet
{ 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;
}

```

Vi ser at metoden starter på det treet med rot i a[0] som går til og med a[i], og i verste må flytte gamle a[i] helt opp til rota a[0]. Avstanden er 'i+1' i arrayen og hver gang **halverer** vi i inntil i verste fall i = 0; dvs. while-løkka går maks. $\log(i+1)$ ganger = **$O(\log n)$** fordi 'i' maksimalt er lik 'n' og gjennomsnittlig 'n/2' (dette er også samme som at høyden i et binærtre er $\log(n)$)

Almira Karabeg, W11.L2

 Department of Informatics, University of Oslo, Norway
 INF110 – Algorithms & Data Structures

Page 14

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)
- 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 $a[0]$



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 ($k= n, n-1, \dots$)

$a[]$:

0	1	2	3	4	5	6	7	8
4	7	2	1	5	9	5	8	6



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 gjennomstithøyde = $(\log n) / 2 = n \cdot \log n / 4$
 - Så setter vi inn en ny node 'n' ganger i toppen av det treet som er i $a[0..k]$, $k = n, n-1, \dots, 2, 1$
I snitt er høyden på dette treet (nesten) $\log n$ – dvs $n \log n$
 - Summen er klart $O(n \log n)$



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 = $O(n \log n)$



Flette - sortering (merge)

Velegnet for sortering av filer.

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. 1024 MB intern hukommelse er i dag meget billig (noen får 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.first;}
    else      { C.wite (b); b = B.first;}

  while (a!= null) { C.write (a); a = A.first;}

  while ( b!= null) { C.write (b); b = B.first;}
}
```

VIKTIG: Se på analysen i boken deres.



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



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.





```

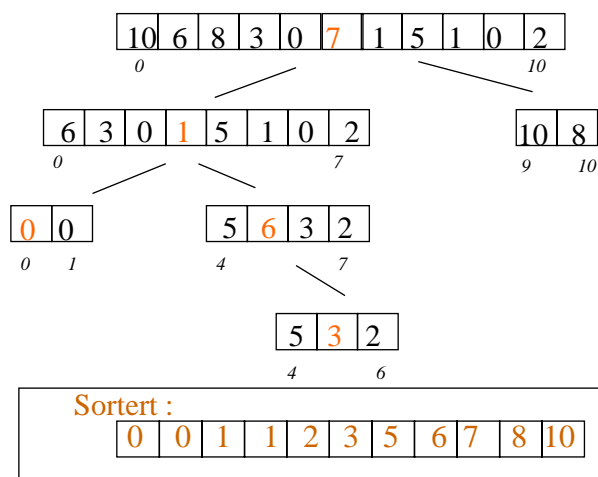
void quickSort ( int [] a, int l, int r)
{ int s = l-1, like = 0, ind;
  int t, part = a[(l+r)/2];

  for ( int b = l; b <= r; b++)
  if ( a[b] == part )
  { like++; // bytt om den venstre store
    t = a[s+like]; // og den nye a[b]
    a[s+like] = a[b];
    a[b] = t;
  } else
  if ( a[b] < part ) {
    s++; // bytt om syklisk den venstre
    ind = s+like; // store , den venstre like og
    t = a[b]; // den nye a [b]
    a[b] = a[ind];
    a[ind] = a[s];
    a[s] = t;
  }
  if ( l < s ) quickSort (a,l,s);
  if ( s+1+like < r ) quickSort (a,s+1+like,r);
}

```



QuickSort - eksempel



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 i praksis II

- Valg av partisjonerings-element 'part' er vesentlig
- Bokas versjon av Quicksort OK, men flytter partisjonerings elementet ut på sidelinje, og tar ikke høyde for flere like elementer
- Velger derfor ofte medianen (det midterste i verdi) av:
 - det første
 - det midterste
 - det sisteelementet i det området vi skal sortere

OBS: se på samme animasjoner av forskjellige algoritmer...

<http://cs.smith.edu/~thiebaut/java/sort/demo.html>

<http://www.aesiesoft.ru/Projects/SortAlg/>

