Henning Christiansen

Sortering

Her: sortering af arrays af objekter

- Hvorfor beskæftige sig med sortering?
- Væsentligt til effektiv implementation af metoder på samlinger af objekter
  - Søgning, jvf. binær søgning
  - Sammenligning
  - ...
- Præsentation af store mænger output
  - søgesresultater efter ”prioritet”
  - postsystem, efter data, alfabetisk afsender, subject, længde, ...
- Et af de ældste og bedst undersøgte områder i datalogien
- Et lærestykke i algoritmekonstruktion & -analyse

Indhold

(— lidt om formel til vurdering af tidsforbrug ved del-og-hersk; opfølg fra sidste gang)

- Insertionsort, sortering ved indsættelse
  - m. kompleksitetsanalyse
- Shellsort — interessant optimering af Insertionsort
- Mergesort — eksempel på del-og-hersk m. stort lagerforbrug
- Quicksort — del-og-hersk
  - sorteringsalgoritm***en***
  - eksempel på meget elegant algoritme

Generel formel til vurdering af tidsforbrug

```
public static Løsning løs(Problem p)
  {  if(simpel(p)) return løsningen-på-det-simple-problem;
     split p op i p1, ..., pn;
     L1 = løs p1;
     L2 = løs p2;
     . . .
     Ln = løs pn;
     return kombiner(L1, ;2, ..., Ln); }
```

Parametre:
A, antallet af delproblemer (n ovenfor)
B, mål for relativ størrelse af delproblem (f.eks. B=2 for halvering)
k, bestemt ved ”overhead” Theta(N^k) (= prisen for at ”kombinere”)

$$T(N) = \begin{cases} O(N^{\log_B A}) & \text{hvis } A > B^k \\ O(N^k \log N) & \text{for } A = B^k \\ O(N^k) & \text{for } A < B^k \end{cases}$$

Anvende den på binær søgning ....

```
private static int binarySearch(Comparable[]a,Comparable x, int low, int high)
{if( low > high ) return -1
  int mid = ( low + high ) / 2;
  if(a[mid].compareTo(x)< 0) return binarySearch(a,x,mid+1,high);
  else if(a[mid].compareTo(x)>0) return binarySearch(a,x,low,mid-1);
  else return mid;}
```

A=1, B=2, Theta(N^k) = 1; dvs. k=0.
$B^k = 2^0 = 1 = A$, dvs. midterste tilfælde:

$T(N) = O(N^k \log N) = O(N^0 \log N) = O(1 \text{ x } \log N) = O(\log N)$

Anvende på ”maximum contiguous subsequence sum”, s. 261 i bog
A=2, B=2, Theta(N^k) = N; dvs. k=1.
$B^k = 2^1 = 2 = A$, dvs. midterste tilfælde:

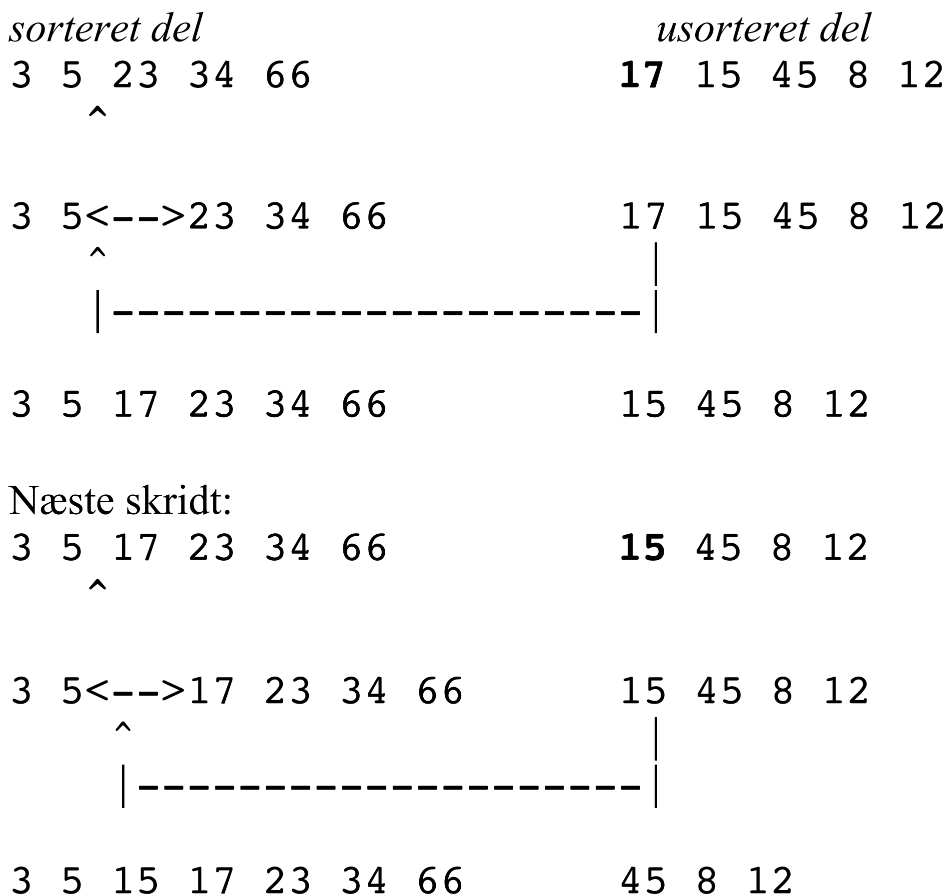
$T(N) = O(N^k \log N) = O(N^1 \log N) = O(N \log N)$

Insertionsort; sortering ved indsættelse

Simpel, effektiv for små datasæt, god introduktion til emnet

Princippet: Som du sorterer en håndfuld spillekort

Billede midtvejs:



At skrive det som generisk algoritme i Java:

— anvender på objekter som er underklasse af Comparable

— datastrukturen er arrays, dvs.

- man kan ikke møve sig til plads,
- elementerne må flyttes enkeltvis
- der er ikke “globalt overblik” (som kortspilleren tror at have)

Hvordan var det nu med de der Comparable??

package java.lang;

public interface Comparable{ int CompareTo(Object other); }

Konvention

	repræsenterer intuitivt
x.CompareTo(y) < 0	x < y
x.CompareTo(y) = 0	x = y
x.CompareTo(y) > 0	x > y

Nu koder vi lige ud af landevejen:

```
public static void insertionSort( Comparable [ ] a )
{
   for( int p = 1; p < a.length; p++ )
   {
      Comparable tmp = a[ p ];
      int j = p;

      for( ; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
         a[ j ] = a[ j - 1 ];
      a[ j ] = tmp;
   }
```

Teknikken:
Flytning af elementer og sammenligninger for at finde ny plads slået sammen, så "tom" plads til nyt element "ruller" nedefter.

Tidsforbrug: værste = gennemsnit = $O(n^2)$
bedste = $O(n)$ (når sorteret i forvejen)

Nu kan vi sortere lystigt derudaf:

```
class RumskibFraMars implements Comparable{
   £¶∞™≠’’≠[™£»ŒÂ‡˘Œ ;
   int CompareTo(Object other); { Ÿ‡ﬁ‰¢&\‰#€\  } }

marsFlåde = new RumskibFraMars [];
.....
insertionSort(marsFlåde);

minListeAfTalCeller = new Integer [];
..... minListeAfTalCeller[17] = 534;
insertionSort(listeAfTalCeller)
```

Men.... Java er noget hø!

```
minListeAfTal = new int [];
.....
insertionSort(minListeAfTal);
TYPEFEJL — INGEN METODE insertionSort( int [] )
```

For effektiv repræsentation og sortering af arrays af tal er vi nødt til at skrive en ny:

```
public static void insertionSort( int [ ] a )
{for( int p = 1; p < a.length; p++ )
   { Comparable tmp = a[ p ]; int j = p;
     for( ; j > 0 && tmp < a[ j - 1 ] ) < 0; j-- )  a[ j ] = a[ j - 1 ];
      a[ j ] = tmp; }
```

og også

```
public static void insertionSort( byte [ ] a ) {...};
public static void insertionSort( short [ ] a ) {...};
public static void insertionSort( long [ ] a ) {...};
```

.... og for hver af de 4 øvrige primitive typer

Se selv tilsvarende eksempler i Java API ... Java er noget hø!

Shellsort: Optimering af insertionSort vha. insertionSort

Tidsforbrug for insertionSort

værste = gennemsnit = $O(n^2)$

bedste = $O(n)$ (når sorteret i forvejen)

Generelt: $\approx O(n)$ for "næsten sorterede" arrays.

Tricket i Shellsort:
Sorterer du "delarrays" bestående af hvert h'te element, bliver arrayet lidt tættere på næsten-sorteret (h et eller anden tal).
Eksempel h = 4

97 87 43 55 48 19 72 77 65 44 23 88 15 7 46

Betragt som 4 sammenvævede delarrays:

97 87 *43* 55 **48** 19 *72* 77 **65** 44 *23* 88 **15** 7 *46*

Sortér hver af disse indbyrdes insertionSort (hvor "+1" erstattes af "+4" og "-1" af "-4"):

15 7 *23* 55 **48** 19 *43* 77 **65** 44 *46* 88 **97** 87 *72*

Sortering med h = 3 må forventes at gå lidt hurtigere end $O(n^2)$

15 7 *23* **55** 48 *19* **43** 77 *65* **44** 46 *88* **97** 87 *72*

Observation: h=1, så er vi tilbage ved insertionSort

Shellsort: udføre en række insertionSort-eringer med h'er

$$h_t > h_{t-1} > h_{t-2} > ... > h_2 > h_1 = 1$$

Varianter over Shellsort = forskellige sekvenser af h’er:

Forskellige strategier:

	worst case	gennemsnit
/2	O(n2)	O(n3/2) = O(n1,5)
/2 og +1 hvis lige	O(n3/2) = O(n1.5)	??O(n5/4) = O(n1.25)
/2.2	–	??O(n7/6) = O(n1.17)

Ifølge vor bog :)

		Shellsort		
N	Insertion Sort	Shell's Increments	Odd gaps only	Dividing by 2.2
10,000	575	10	11	9
20,000	2,489	23	23	20
40,000	10,635	51	49	41
80,000	42,818	114	105	86
160,000	174,333	270	233	194
320,000	NA	665	530	451
640,000	NA	1,593	1,161	939

Shellsort stammer fra 1959, men tidskompleksitet stadig mål for ny forskning:

http://www.informatik.uni-trier.de/~ley/db/indices/t-form.html

Search Title

Keyword: shellsort Submit Reset

DBLP: [Home | Search: Author, Title | Conferences | Journals]

Resultat:

Bronislava Brejová: Analyzing variants of Shellsort. Information Processing Letters 79(5): 223-227 (2001)

Marcin Ciura: Best Increments for the Average Case of Shellsort. FCT 2001: 106-117

Tao Jiang, Ming Li, Paul M. B. Vitányi: A lower bound on the average-case complexity of shellsort. JACM 47(5): 905-911 (2000)

Svante Janson, Donald E. Knuth: Shellsort with three increments. Random Structures and Algorithms 10(1-2): 125-142 (1997)

Renren Liu: An Improved Shellsort Algorithm. TCS 188(1-2): 241-247 (1997)

og mange-mange flere

Søgning på Shell<mellemrum>sort:

Robert T. Smythe, J. Wellner: Stochastic Analysis of Shell Sort. Algorithmica 31(3): 442-457 (2001)

Mergesort – del-og-hersk med O(n log n)

baseret på feltning af allerede sorterede sekvenser.

En billede fra dengang mor var dreng:

Store datamængder opbevaredes på bånd!
Altid sorterede!
Tre båndstationer:

bånd 1: status fra igår
bånd 2: transaktioner fra idag – sorteret!
bånd 3: her skrives status i dag ved lukketid

Eksempel:

Bånd 1: 11 22 33 44 55 66
Bånd 2: 8 27 47
Bånd 3:

Bånd 1: 11 22 33 44 55 66
Bånd 2: 27 47
Bånd 3: 8

Bånd 1: 22 33 44 55 66
Bånd 2: 27 47
Bånd 3: 8 11

Bånd 1: 33 44 55 66
Bånd 2: 27 47
Bånd 3: 8 11 22

Bånd 1: 33 44 55 66
Bånd 2: 47
Bånd 3: 8 11 22 27
....

Bånd 3: 8 11 22 33 44 47 55 66

Fletning af to *arrays* over i et tredje; som før men med tællere

```
private static void merge( Comparable [ ] a, Comparable [ ] tmpArray,
                           int leftPos, int rightPos, int rightEnd )
   // [] a indeholder to sorterede sekvenser
   //      1. fra position leftPos til rightPos-1
   //      2. fra position rightPos til rightEnd
   // De to sekvenser flettes over i tmpArray
   // og kopieres tilbage i a
       {....};
```

Mergesort lige ud ad landevejen:

```
public static void mergeSort( Comparable [ ] a )
   {Comparable [ ] tmpArray = new Comparable[ a.length ];
     mergeSort( a, tmpArray, 0, a.length - 1 ); }

private static void mergeSort( Comparable [ ] a, Comparable [ ] tmpArray,
            int left, int right )
   { if( left < right ) {
         int center = ( left + right ) / 2;
         mergeSort( a, tmpArray, left, center );
         mergeSort( a, tmpArray, center + 1, right );
         merge( a, tmpArray, left, center + 1, right ); }}
```

Tidsforbrug:

Værst = Bedst = Gennemsnit = O(n log n)

Problemer: Kræver et ekstra array + kopiering frem og tilbage

En anden, Quicksort, = O(n log n) ca. 3 gange så hurtig (?)

Quicksort. Grundrincippet er meget enkelt, del-og-hersk

At sortere en sekvens af elementer S:

1. Hvis længden af S er 0 eller 1, så er vi færdige, ellers
2. Vælg et element v i S kaldet omdrejningspunkt ("pivot")
3. Flyt rundt på elementerne i S, så S kommer til at se ud som:

   x1, x2, ..., xk, v, y1, y2, ... ym

   hvor alle x'erne ≤ v og y'erne ≥ v
4. Sortér x1, x2, ..., xk på deres pladser i S
5. Sortér y1, y2, ... ym på deres pladser i S

For at vurdere tidsforbrug. Skridt 3 er kritisk: vi påstår O(n)

Princip:
Kør en tæller ind fra højre eller venstre, så snart "fejl" opdages, byt om;
Blive ved med det til tællerne mødes;

I Java (lidt enklere end i bogen)**:**

```
private static void partition(Comparable [ ] a,
                                   int low, int pivotIndex, int high )
  { Comparable pivot = a[pivotIndex];
    int i = low; int j = high;
    while(true)
      {while( a[ i ].compareTo( pivot ) < 0 ) i++;
       while( pivot.compareTo( a[ j ] ) < 0)  j--;
       if( i > j ) break;
       swapReferences( a, i, j );i++;j--; } }
```

Optælling:
i og j flyttes tilsammen a.length gange, og op til n swaps

Værst=Bedst=Gennemsnit = O(n).

Quicksort. Grundrincippet er meget enkelt, del-og-hersk
(gentaget)
At sortere en sekvens af elementer S:

1. Hvis længden af S er 0 eller 1, så er vi færdige, ellers
2. Vælg et element v i S kaldet omdrejningspunkt ("pivot")
3. Flyt rundt på elementerne i S, så S kommer til at se ud som:

   x1, x2, ..., xk, v, y1, y2, ... ym

   hvor alle x'erne ≤ v og y'erne ≥ v
4. Sortér x1, x2, ..., xk på deres pladser i S
5. Sortér y1, y2, ... ym på deres pladser i S

Vi ved nu skridt 3 er O(længde S).

Definition: *Medianen* for z1, z2, ..., zn er et zi,
så ca. halvdelen af z'er ≤ zi og ca. halvdelen af z'er ≥ zi

Obs: I de tilfælde, hvor vi på magisk vis er heldige at ramme (ca.) medianen som omdrejningspunkt, har vi O(n log n):
et balanceret binært træ som kaldetræ med dybde n og halvering af arbejde hver gang.
(eller check generel formel, argument som ved maximum contiguous subsequence sum)

Obs: Værste tilfælde $O(n^2)$ er hvis vi altid er uheldige og rammer det største element som omdrevningspunkt:
et skævt træ af dybde n og n-1, n-2, ..., 1, 0 arbejde på hvert niveau

Det kan bevises (vi gider ikke):
gennemsnit O(n log n)

Tommelfingerregel: Hvis omdrejningspunktet vælges ca. midt går det ikke galt med i forvejen næsten sorterede data.

Obs: Vi kunne i princippet godt beregne medianen hvergang, men tung beregning og ikke sikkert det giver noget...

Her en version af Quicksort som er enklere end bogens:

Tilpasset efter N.Wirth (1976)

Opsplitning som særskilt metode:

```
public static int partition(Comparable [ ] a, int low, int pivotIndex, int high )
//return new position for pivot element
{ Comparable pivot = a[pivotIndex];
    int i = low; int j = high;
    while(true)
      {while( a[ i ].compareTo( pivot ) < 0 ) i++;
       while( pivot.compareTo( a[ j ] ) < 0)  j--;
       if( i > j ) break;
       swapReferences( a, i, j );i++;j--;}
    return i; }
```

NBNB: FEJL I OVENSTÅENDE; SE
http://www.dat.ruc.dk/~henning/DatC_DsAlg/QSort.java
FOR RETTE VERSION!!!

Som i bogen, sæt nedre grænse på hvor vi går over til en anden metode; forenkle ved at vælge midterste element som omdrejningspunkt:

```
private static final int CUTOFF = 10;

private static void quicksort( Comparable [ ] a, int low, int high )

{ if( low + CUTOFF > high ) insertionSort( a, low, high );
  else
    { int middle = ( low + high ) / 2;
      int newPosForOldMiddleElt = partition(a, low, middle, high);
      quicksort( a, low, newPosForOldMiddleElt - 1 );
      quicksort( a, newPosForOldMiddleElt + 1, high ); }}
```

Bogens version lidt sværere at overskue:

- — ikke lagt “partition” ud i særskilt metode
- — som omdrejningspunkt vælges median af a[low], a[mid], a[high]
- — optimeringer:
  - o beregning af omdrejningspunkt kombineres med sortering af a[low], a[mid], a[high].
  - o man undgår allokering af variabel svarende til “pivot” i “partition”

```
private static void quicksort( Comparable [ ] a, int low, int high )
{ if( low + CUTOFF > high ) insertionSort( a, low, high );
  else
   { // Sort low, middle, high
     int middle = ( low + high ) / 2;
     if(a[middle].compareTo(a[low])<0) swapReferences(a,low,middle);
     if(a[high].compareTo(a[low])<0) swapReferences(a,low,high);
     if(a[high].compareTo(a[middle])<0) swapReferences(a,middle,high);
     // Place pivot at position high - 1
     swapReferences( a, middle, high - 1 );
     Comparable pivot = a[ high - 1 ];
     // Begin partitioning
     int i, j;
     for( i = low, j = high - 1; ; )
     { while( a[ ++i ].compareTo( pivot ) < 0 ) ;
       while( pivot.compareTo( a[ --j ] ) < 0 );
       if( i >= j ) break;
       swapReferences( a, i, j );}

     // Restore pivot
    swapReferences( a, i, high - 1 );

    quicksort( a, low, i - 1 );    // Sort small elements
    quicksort( a, i + 1, high );   // Sort large elements
      }
   }
```

Afsluttende øvelse: QuickSelect

At finde det k'te største element i array a ≈

1. Sortér
2. Find svaret i a[k-1]   ("-1" fordi vi starter med nul)

Benyt quicksort, men drop det ene rekursive kald:

```
private static final int CUTOFF = 10;

private static void quick~~sort~~select( Comparable []a, int low, int high, int k)

{ if( low + CUTOFF > high ) insertionSort( a, low, high );
  else
    { int middle = ( low + high ) / 2;
      int newPosForOldMiddleElt = partition(a, low, middle, high);
      if(k<i-1) quick~~sort~~select( a, low, newPosForOldMiddleElt - 1 );
      if(k>i-1) quick~~sort~~select( a, newPosForOldMiddleElt + 1, high ); }}
```

Dvs. gennemsnitsopførsel falder fra O(N log N) til O(log N)

— check evt. generel formel, argument nu som binær søgning og
  - og ikke som ved "maximum contiguous subsequence sum"

  Dvs. parametren A falder fra 2 til 1

S L U T