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, emne, længde, ...
- Et af de ældste og bedst undersøgte områder i datalogien
- Et lærestykke i algoritmekonstruktion & -analyse

Indhold

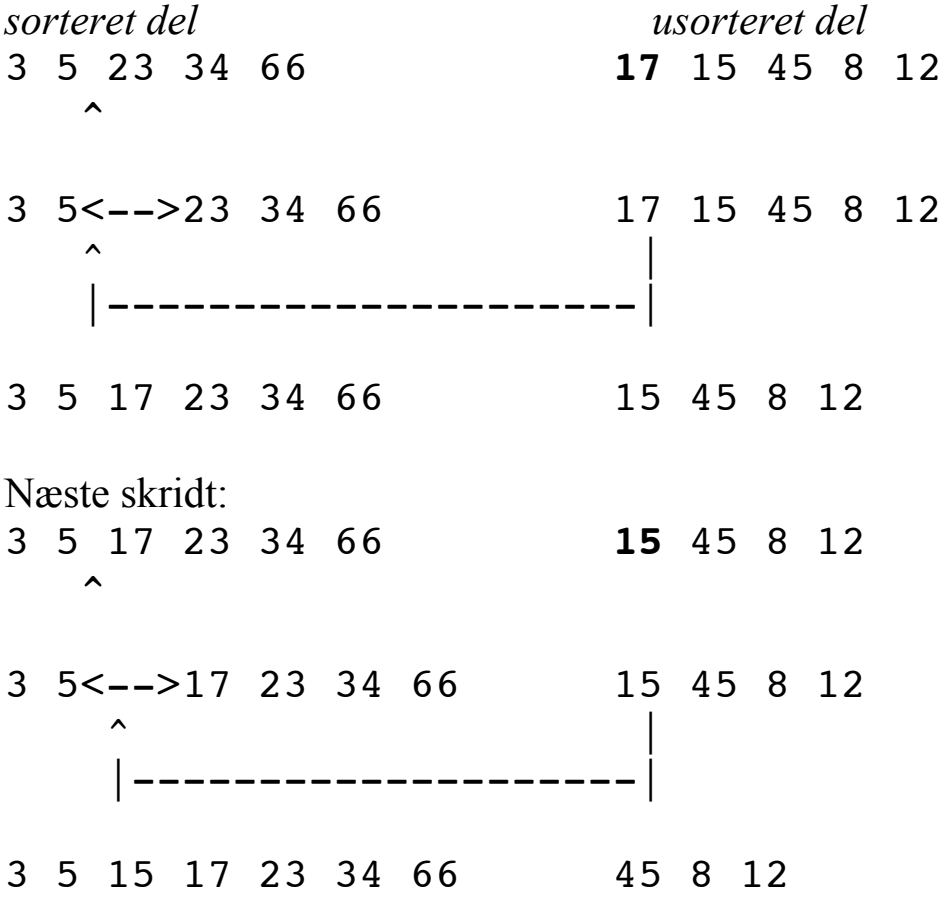
- Insertionsort, sortering ved indsættelse
  - m. kompleksitetsanalyse
- Shellsort — interessant optimering af Insertionsort
- Mergesort — eksempel på del-og-hersk m. stort lagerforbrug
- Quicksort — del-og-hersk
  - sorteringsalgoritm***en***
  - eksempel på meget elegant algoritme

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....

```
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 ... (suk)

Shellsort: Optimering af insertionSort vha. insertionSort

Tidsforbrug for insertionSort

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

Generelt: ≈ 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

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

Varianter over Shellsort = forskellige sekvenser af h'er:

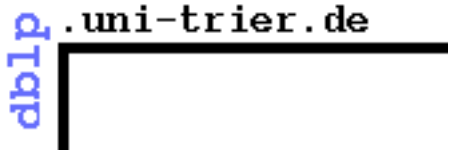
Forskellige strategier:

	worst case	gennemsnit
/2	$O(n^2)$	$O(n^{3/2}) = O(n^{1,5})$
/2 og +1 hvis lige	$O(n^{3/2}) = O(n^{1.5})$	??$O(n^{5/4}) = O(n^{1.25})$
/2.2	–	??$O(n^{7/6}) = O(n^{1.17})$

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**



Search Title

Keyword: shellsort Submit Reset

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

Resultat:

Robert S. Roos, Tiffany Bennett, Jennifer Hannon, Elizabeth Zehner: A Genetic Algorithm For Improved Shellsort Sequences. GECCO 2002: 694

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)

Robert T. Smythe, J. A. Wellner: Asymptotic analysis of (3, 2, 1)-shell sort. Random Structures and Algorithms 21(1): 59-75 (2002)

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

Baseret på fletning 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:

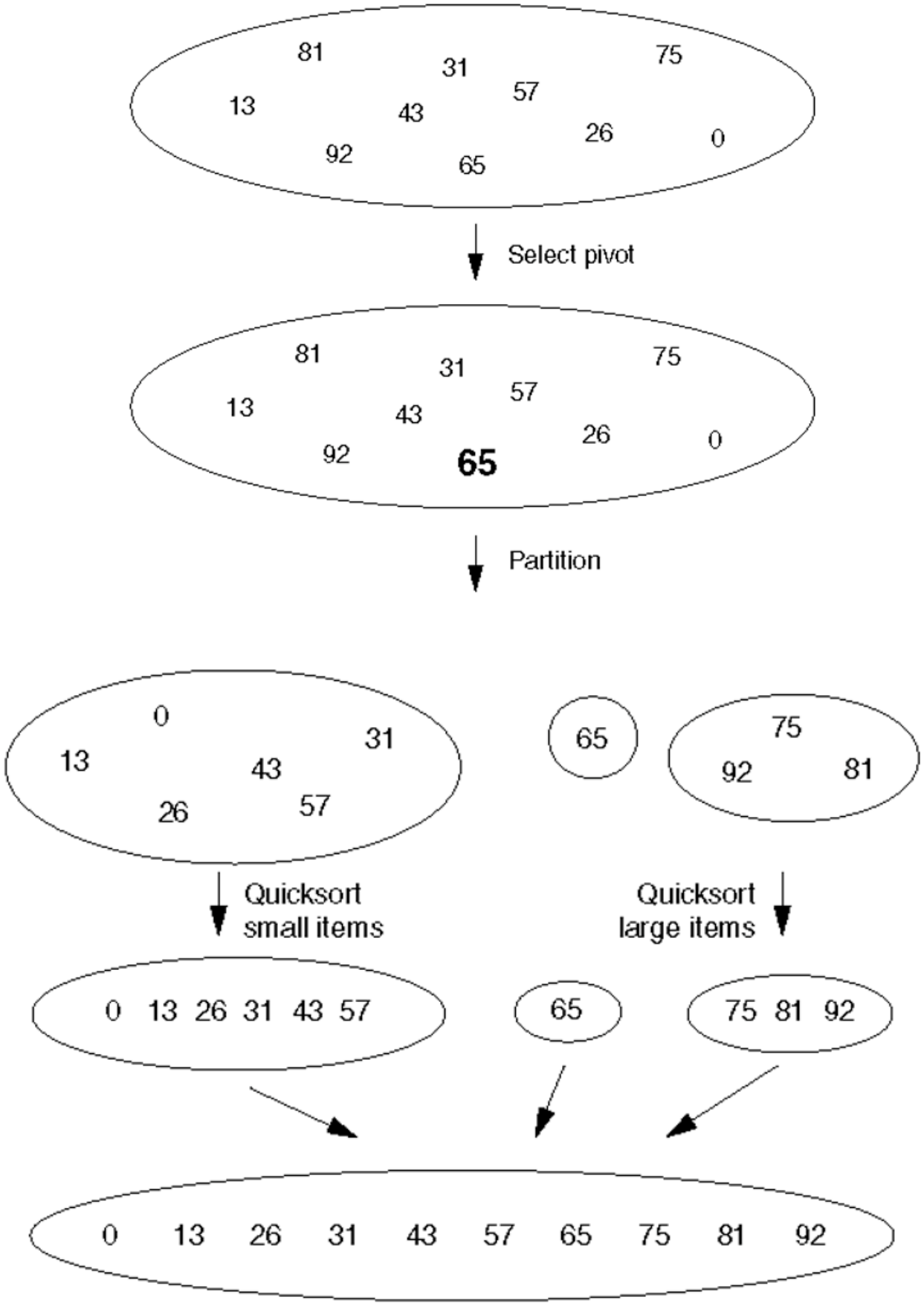
1. Hvis længden af S er 0 eller 1, så er vi færdige, ellers
2. Vælg et element v i S kaldet omdrejningspunkt ("pivot")
3. Flyt rundt på elementerne i S, så S kommer til at se ud som:

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

   hvor alle x'erne $\leq$ v og y'erne $\geq$ 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)

Tegninger fra bogen



Princip for »Pivotering«:

Kør tællere ind fra højre og venstre, så snart "fejl" opdages, byt om;
Blive ved med det til tællerne mødes;

Komplikation: Pivoteringselementet må typisk også flyttes

Trick: Pivoteringselementet flyttes hen i den ene ende, og på plads igen til sidst.

Antag følgende array, og 6 på magisk vis er valgt som pivot.element.

```
        8 1 4 6 0 3 5 2 7 9
              \-----------/

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

```
for( i = low, j = high; ; )
   {   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 );
```

```
        2 1 4 5 0 3 9 8 7 6
                    \-----/

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

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 $\leq$ v og y'erne $\geq$ 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 $\leq$ zi og ca. halvdelen af z'er $\geq$ 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 den sande median medianen hvergang, men tung beregning og ikke sikkert det giver noget...

Tommelfingerregel:

Vælg som pivoteringselement det medianen af
første, midterste og sidste arrayelement

F.eks.

```
8 x x x 6 x x x x 0
```

Disse »sorteres« ved test og ombytninger:

```
// 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 );
```

Dvs. vi får

```
0 x x x 6 x x x x 8
```

Medianen (6) står altså i midten, og første (0) og sidste (8) er på ret plads.

Ergo, se bort fra første og sidste i pivoteringen:

```
0 x x x x x x x 6 8
  ============
```

Og forsæt med pivotering som før.

Så mangler vi blot indsætte rekursive kald + optimere med en enklere metode for små del-arrays:

```
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 arrays starter med nul)

Benyt quicksort, men drop det ene rekursive kald:

```
private static void quickSELECTsort( Comparable [ ] a, int low, int high,
int Position_in_Order )
{ if( low + CUTOFF > high ) insertionSort( a, low, high );
  else
   { // Sort low, middle, high
     ....
     // Place pivot at position high - 1
     .....
     // Begin partitioning
     ....
     // Restore pivot
     ....

    if ( Position_in_Order  <= i)
              quickSELECTsort( a, low, i - 1, Position_in_Order  );
   else if ( Position_in_Order  > i+1)
              quickSELECTsort( a, i + 1, high, Position_in_Order  );
      }
   }
```

Dvs. gennemsnitsopførsel falder fra O(N log N) til O(N), men værste fald stadig $O(N^2)$ [Prøv at gennemføre argumenterne hvorfor]

S L U T