

# Sortering

- Varför sortering?
- Hur jämföra?
- Bubblesort
- Selectionsort
- Insertionsort
- Mergesort
- Quicksort
- Shellsort
- Jämföra objekt med Comparator och Comparable i Java

# Varför sortera?

- För att hitta lättare, findability
- I en sorterad lista kan man hitta saker på en konstant tid eller iallafall på  $O(\log n)$  tid.
- Man kan ordna på många olika sätt, men en sorteringsalgorithm behöver bara veta att varje element (nyckel) går att jämföra med  $>$ ,  $<$ ,  $=$ .
- Det finns många olika algoritmer, en del är enkla och bör användas på små problem, andra komplicerade och har mycket overhead men är effektiva för mycket stora datamängder.
- För att mäta effektiviteten används jämförelse mellan två nycklar (det har en tids enhet) och att byta plats på två element.

## Bubblesort

Lista 54, 26, 93, 17, 77, 31, 44, 55, 20

Sedan tittar vi på 54, 26 och byter plats på dem om det om de inte är sorterade d.v.s. 26, 54. Dessa två är nu sorterade.


Sedan tittar vi på 54, 93, inget byte.

Sedan tittar vi på 93, 17 de byter plats.....

Listans största värde hamnar sist, det vill säga den ligger på rätt plats.

Nu bublar vi upp det näst största på samma sätt.

# Bubble sort, första passet

First pass									
									
54	26	93	17	77	31	44	55	20	Exchange
26	54	93	17	77	31	44	55	20	No Exchange
26	54	93	17	77	31	44	55	20	Exchange
26	54	17	93	77	31	44	55	20	Exchange
26	54	17	77	93	31	44	55	20	Exchange
26	54	17	77	31	93	44	55	20	Exchange
26	54	17	77	31	44	93	55	20	Exchange
26	54	17	77	31	44	55	93	20	Exchange
26	54	17	77	31	44	55	20	93	93 in place after first pass

# Bubblesort

## Implementering

```
public static void bubblesort(int[] array) {  
    for( int i=0; i < array.length - 1; i++ ) {  
        for( int j = array.length - 1; j > i; j-- ) {  
            if( array[ j ] < array[ j - 1 ] ) {  
                Utility.swap( array, j, j - 1 );  
            }  
        }  
    }  
}
```

Först görs n-1 jämförelser

I värsta fallet n-1 byten

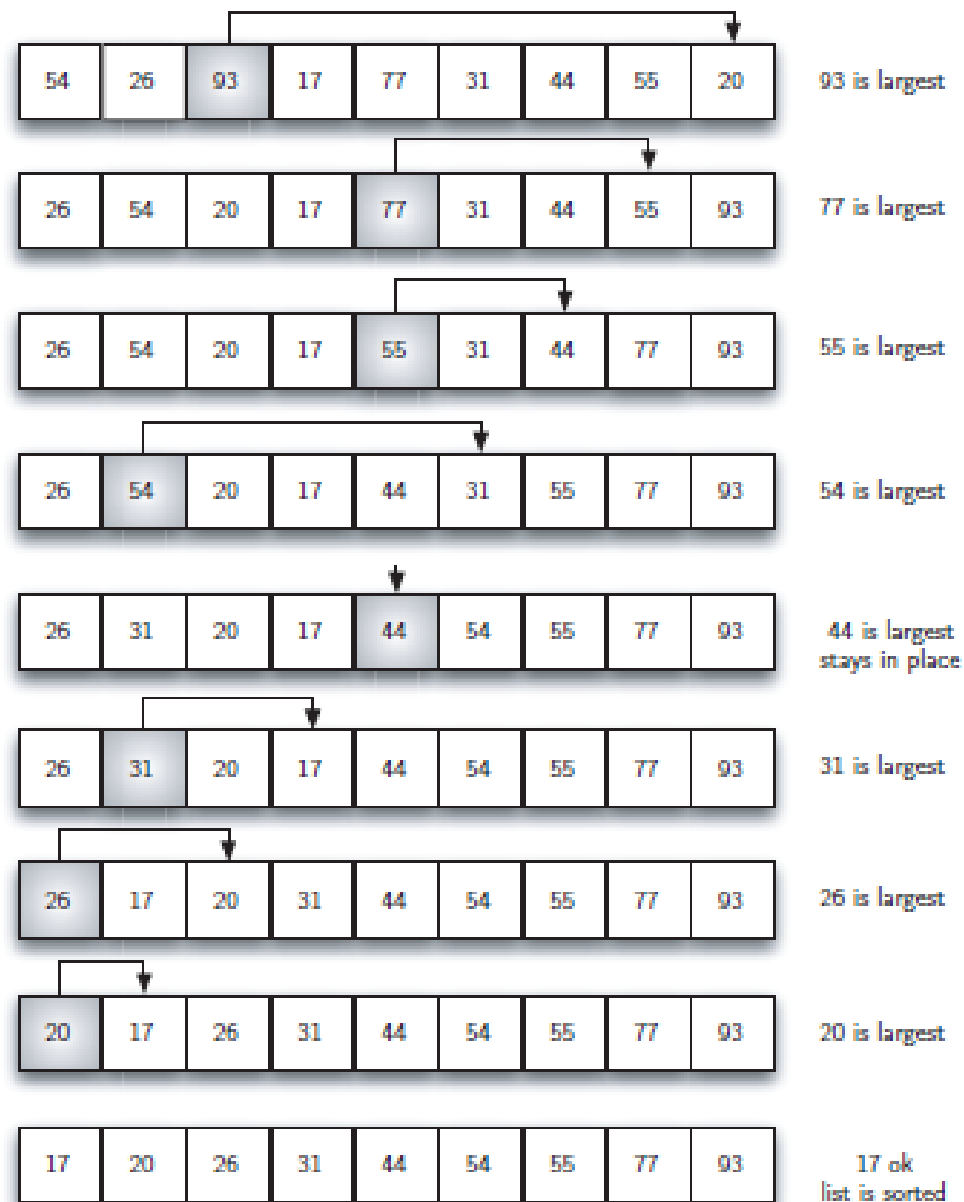
Sedan n-2 jämförelser....

$$n-1 + n-2 + n-3 + \dots + 3 + 2 + 1 = n((n-1)/2) = \\ 1/2n^2 - 1/2n = O(n^2)$$

## Selectionsort

- Är en modifiering av Bubblesort.
- Precis som Bubblesort letar den upp det största elementet och lägger den på rätt plats. Sedan det näst största elementet....o.s.v.
- Skillnaden är att den bara gör högst ett byte per pass.
- Den letar upp det största elementet sedan byter den plats med elementet på den sista platsen.
- $O(n^2)$  men i praktik snabbare än Bubblesort, lika många jämförelser men färre byten.

# Selektion Sort



## Insertionsort

- Denna sortering sorterar in elementen en efter en i en redan sorterad lista.

Lista 54, 26, 93, 17, 77, 31, 44, 55, 20

- 54 sorterad.
- Sedan sorteras 26 in i listan 54.  
26, 54 är nu sorterad.
- Sedan sorteras 93 in i listan.  
Ger 26, 54, 93

Så fortsätter det tills listan är klar

$O(n^2)$  jämförelser, Men i bästa fallet (en sorterad lista) så behövs bara 1 jämförelse varje pass D.v.s.  
En sorterad lista ger  $O(n)$



# Insertion sort

54	26	93	17	77	31	44	55	20
----	----	----	----	----	----	----	----	----

Assume 54 is a sorted list of 1 item

26	54	93	17	77	31	44	55	20
----	----	----	----	----	----	----	----	----

inserted 26

26	54	93	17	77	31	44	55	20
----	----	----	----	----	----	----	----	----

inserted 93

17	26	54	93	77	31	44	55	20
----	----	----	----	----	----	----	----	----

inserted 17

17	26	54	77	93	31	44	55	20
----	----	----	----	----	----	----	----	----

inserted 77

17	26	31	54	77	93	44	55	20
----	----	----	----	----	----	----	----	----

inserted 31

17	26	31	44	54	77	93	55	20
----	----	----	----	----	----	----	----	----

inserted 44

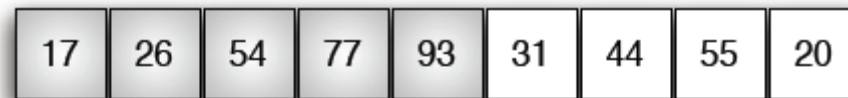
17	26	31	44	54	55	77	93	20
----	----	----	----	----	----	----	----	----

inserted 55

17	20	26	31	44	54	55	77	93
----	----	----	----	----	----	----	----	----

inserted 20

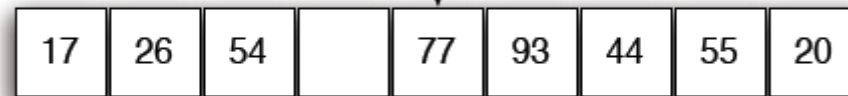
## Den femte insorteringen, detaljerat



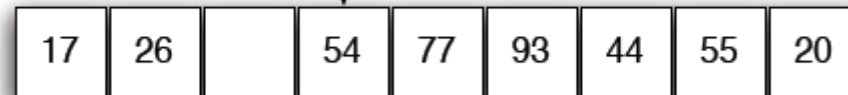
Need to insert 31  
back into the sorted list



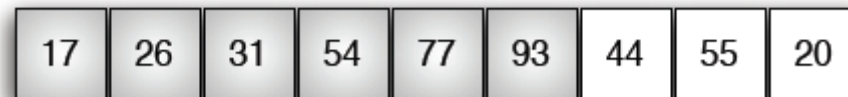
$93 > 31$  so shift it  
to the right



$77 > 31$  so shift it  
to the right



$54 > 31$  so shift it  
to the right



$26 < 31$  so insert 31  
in this position

# Insertionsort

## Implementering ( ungefär suns version )

// Används på små arrayer i java (mindre än 7 element)

```
public static void insertionsort( int[] array ) {  
    for( int i = 1; i < array.length; i++ ) {  
        for ( int j = i; ( j > 0 ) && ( array[ j - 1 ] > array[ j ] ) ; j-- ) {  
            Utility.swap( array, j, j - 1 );  
        }  
    }  
}
```

# Snabbare sorteringsalgoritmer

Sortera snabbare än  $O(n^2)$  tid i värsta fallet!

Två sorteringsalgoritmer:

- Mergesort
- Quicksort

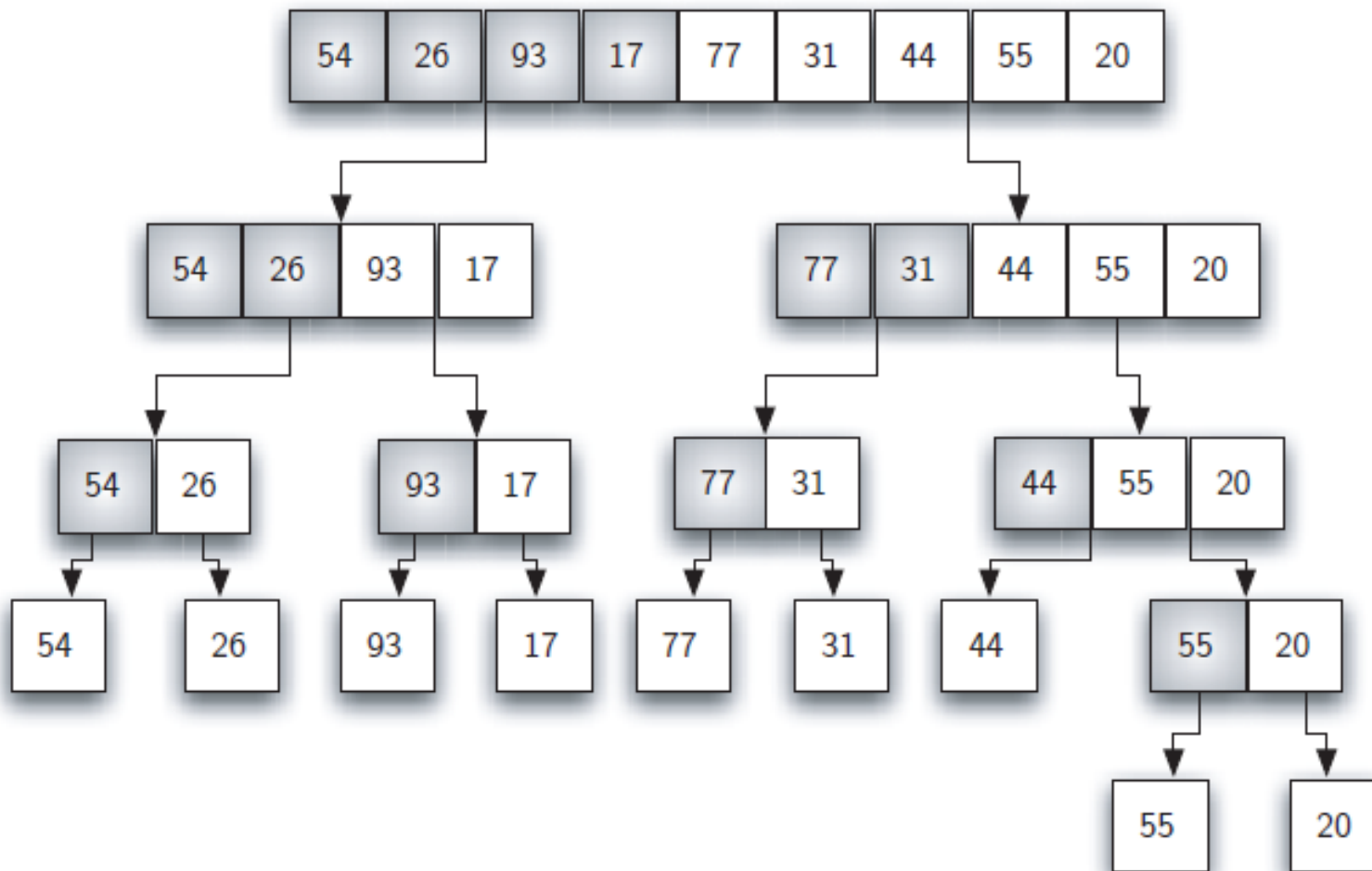
Sorterar på  $O(n \log n)$  tid.

Sortering som bygger på jämförelser kan inte vara snabbare än  $O(n \log n)$ .

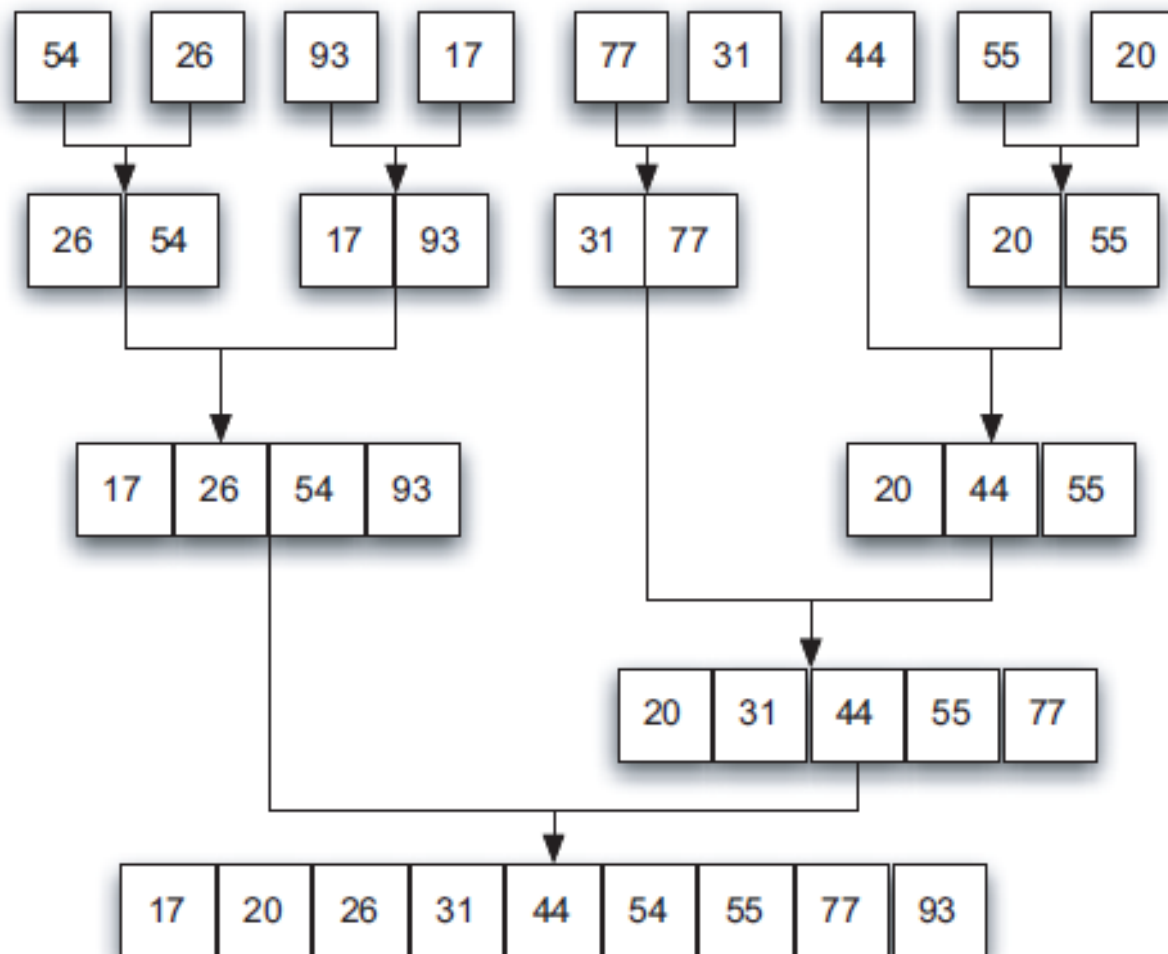
# Mergesort

- En divide and conquer strategi. (söndra och härska, dela och härska).
- Algoritmen använder rekursion.
- Delar listan på hälften, delar sedan första halvan på hälften o.s.v. tills bara ett element återstår.
- "merga" sedan ihop två sorterade listor till en sorterad lista.

## Att dela.....tills listan har längden 1



## Att smälta samman (merga) två listor till en



# Mergesort

## Implementering

```
public static void mergesort( int [] array ) {  
    mergesort( array, 0, array.length );  
}
```

```
private static void mergesort( int[] array, int first, int n ) {  
    int n1,n2;  
    if( n > 1 ) {  
        n1 = n / 2;  
        n2 = n - n1;  
        mergesort( array, first, n1 );  
        mergesort( array, first + n1, n2);  
        merge( array, first, n1, n2 );  
    }  
}
```

```
private static void merge( int[] array, int first, int n1, int n2 ) {...}
```



## Analys

Splittringen sker på  $O(\log n)$  nivåer.

Varje element i listan måste sammanfogas med en annan lista. Inte en gång utan  $\log n$  gånger på vägen upp i rekursionen.

Alltså  $O(n \log n)$ .

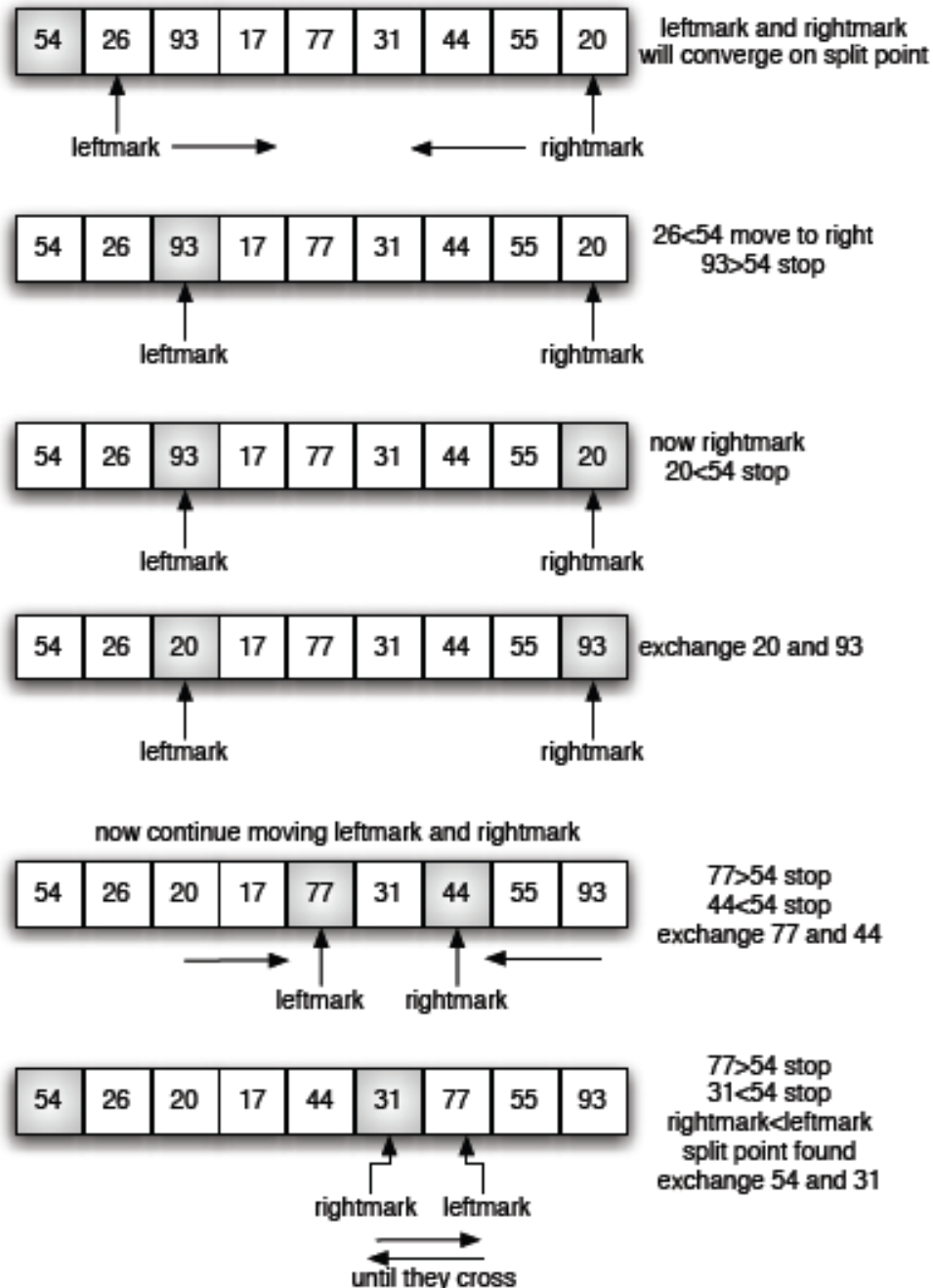
# Quicksort

- Att dela sedan härska på ett annat sätt.
- Välj ett pivot element, t.ex. det första i listan.
- Sätt in det på rätt plats
- Det vill säga lägg alla mindre värden framför och alla större värden bakom i listan
- I fallet nedan ska 54 flytta till plats 5 i listan. 26, 17, 31, 44 och 20 framför och 93, 77 och 55 bakom.

54	26	93	17	77	31	44	55	20
----	----	----	----	----	----	----	----	----

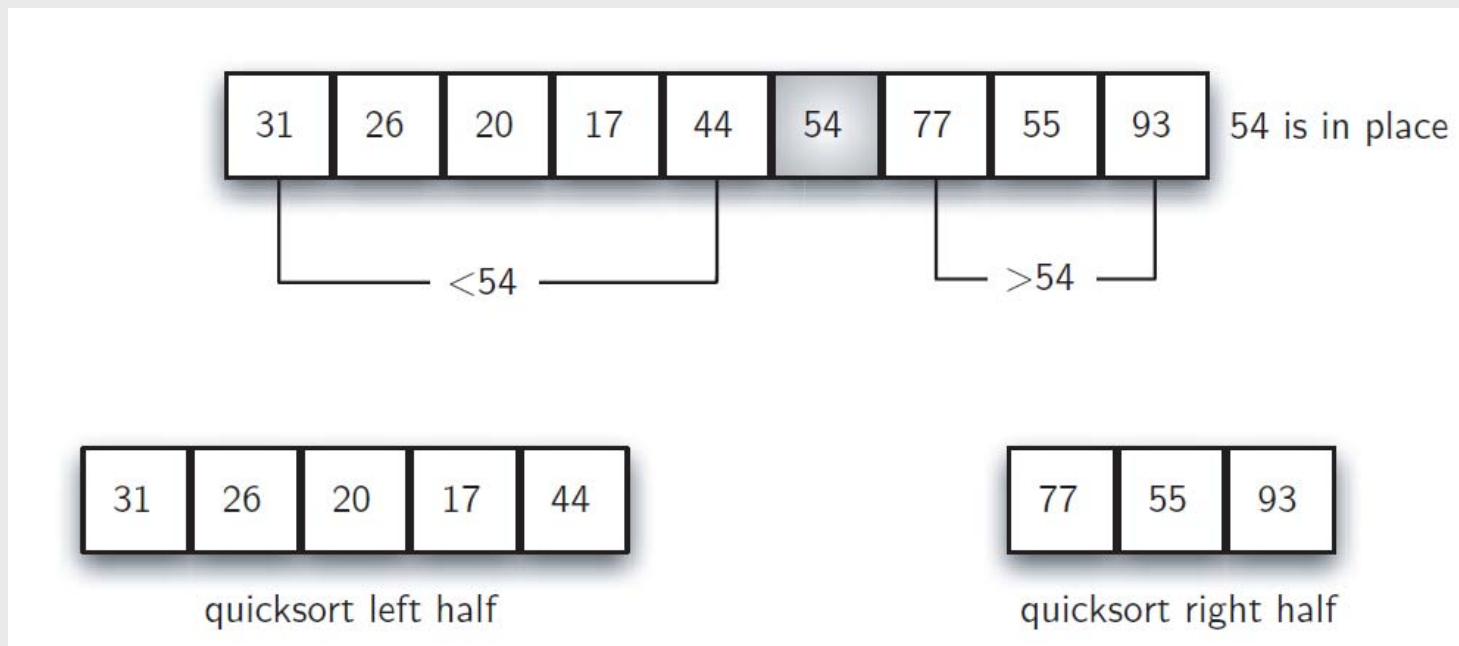
54 will be the first pivot value

Hitta platsen där  
54 ska vara  
och samtidigt  
flytta alla som  
är på "fel"  
sida.



## Första delningen är avslutad

- 54 är på rätt plats
- Gör sedan Quicksort på de osorterade dellistorna på varje sida med hjälp av rekursion



# Quicksort

## Implementering

```
public static void quicksort( int [] array ) {  
    quicksort( array, 0, array.length-1 );  
}
```

```
private static void quicksort( int[] array, int left, int right ) {  
    int pivotIndex;  
    if( left < right ) { // minst två element  
        pivotIndex = partition(array, left, right, (left+right)/2); // flytta elementen  
        quicksort( array, left, pivotIndex-1);  
        quicksort( array, pivotIndex+1, right);  
    }  
}
```

```
private static void partition( int[] array, int left, int right, int pivotIndex) {...}
```

## Pseudokod, quicksort

```
quickSort (A,p,r)
  if p<r
    then q ← partition(A,p,r)
         quickSort (A,p,q)
         quickSort (A,p+1,r)
partition(A,p,r)
  x ← A[p], i ← p-1, j ← r+1
  while True
    do repeat j ← j-1
       until A[j] ≤ x
    do repeat i ← i+1
       until A[i] ≥ x
    if i < j
      exchange A[i] ↔ A[j]
    else return j
```

# Analys

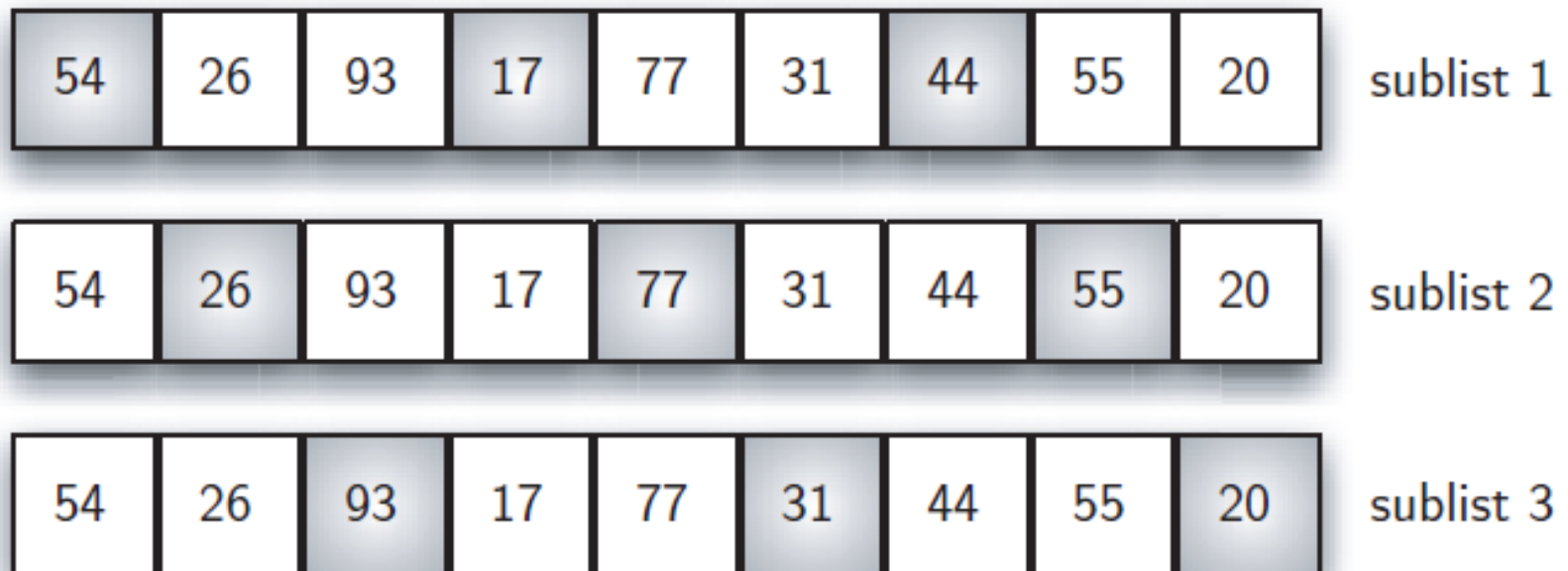
- Quicksort kan sortera i en och samma lista så elementen måste inte kopieras många gånger som i MergeSort. Kallas Quick Sort inplace.
- $O(n \log n)$  i medelfallet MEN
- Om listan redan är sorterad så tar algoritmen  $O(n^2)$  tid. Delnings elementet (Pivot elementet) ligger långt åt höger eller långt åt vänster hela tiden.
- Detta kan man försöka undvika genom att slumpa fram vilket element som ska vara pivot elementet och inte automatiskt ta det första. Man kan även ta det mittersta av tre (först, mitten, sist).
- Om pivot elementet delar listan på mitten varje gång så är Quick Sort optimal.

# Shellsort

- Lite mer komplicerad algoritm.
- Man gör insertionsort på delmängder av listan i olika omgångar.
- Man hoppar ett visst steg (ökning) och sorterar vart i'te element i listan mot varandra. Insertionsort används.
- En viss ordning har uppkommit.
- Sedan görs samma samma sak med ett mindre steg (insertionsort).
- Sist görs sorteringen med insertionsort med steg 1

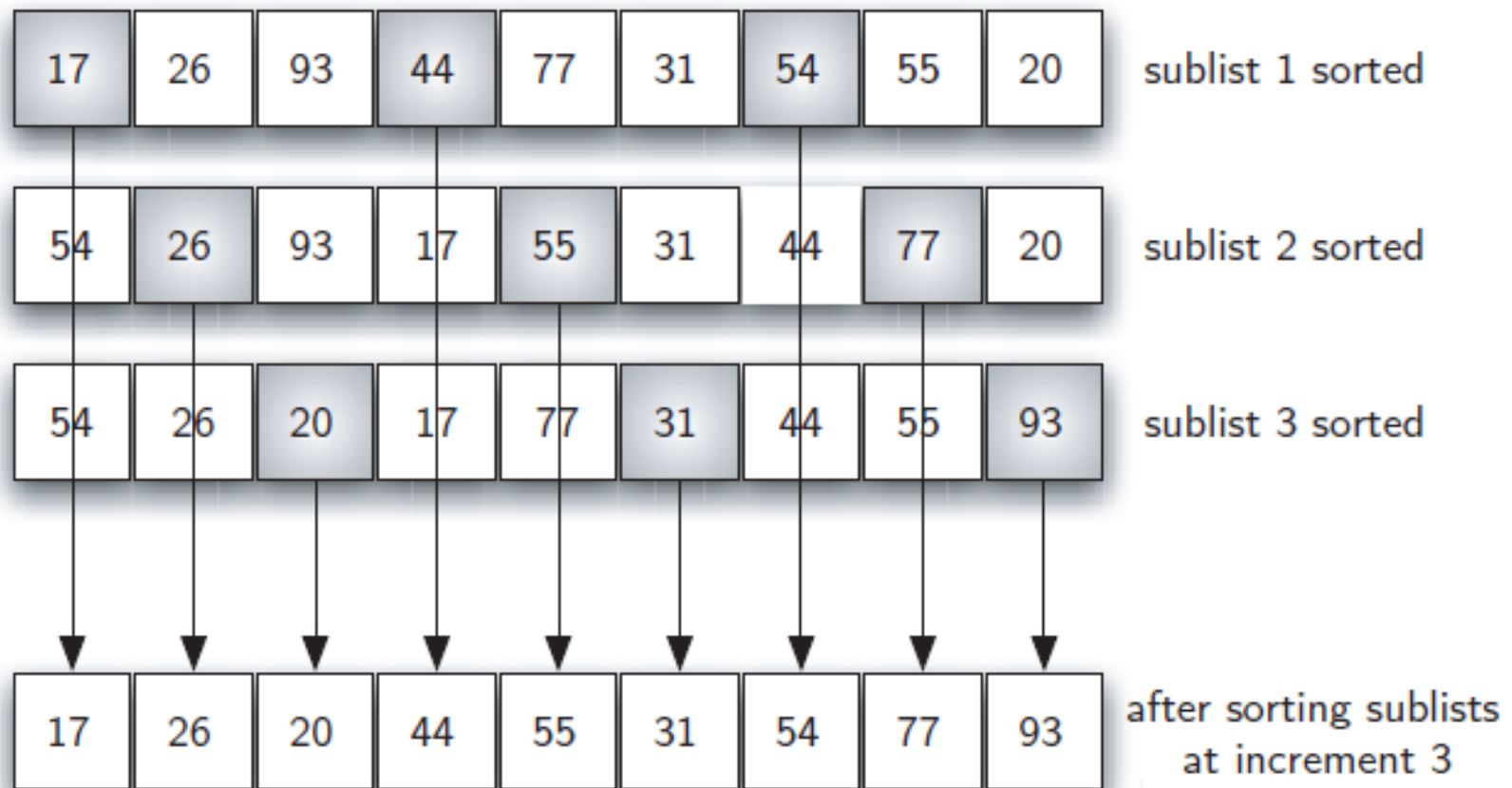


# Shellsort



- En lista dela in i tre listor som sorteras var och en för sig. Steget eller  $i=3$ , i detta fall.

# Shellsort



# Shellsort

- Tänk dig att ett lista förs sorteras med insertionsort med steg 4, sedan steg 2, sedan steg 1.

alist = [54, 26, 93, 17, 77, 31, 44, 55, 20]

shellSort(alist)

Listan efter sortering med steg 4 ( $3+1*3$  jämförelser)

[20, 26, 44, 17, 54, 31, 93, 55, 77]

Listan efter sortering med steg 2 ( $4+3$  jämförelser)

[20, 17, 44, 26, 54, 31, 77, 55, 93]

Listan efter sortering med steg 1

[17, 20, 26, 31, 54, 55, 77, 93] (7 jämförelser)

## Första indelningen med ökning 4

54	26	93	17	77	31	44	55	20	sublist 1
54	26	93	17	77	31	44	55	20	sublist 2
54	26	93	17	77	31	44	55	20	sublist 3
54	26	93	17	77	31	44	55	20	sublist 4

## Analys av Shellsort

- Beroende på hur man väljer stegen så tar Shellsort olika långt tid. Mellan  $O(n)$  och  $O(n^2)$ .
- Använder man stegen  $(2^k)-1$   $k=(\dots 5,4,3,2,1)$  så kan man få  $O(n^{1,5})$ .
- Där  $k$  från början är ungefär hälften av  $n$  och sedan minskar till 1 ( $\dots 31,15,7,3,1$ )

## Olika illustrationer av sortering

- Bubble-sort with Hungarian ("Csángó") folk dance(<https://www.youtube.com/watch?v=lyZQPjUT5B4>)
- Select-sort with Gypsy folk dance (<https://www.youtube.com/watch?v=Ns4TPTC8whw>)
- Insert-sort with Romanian folk dance (<https://www.youtube.com/watch?v=ROaIU379I3U>)
- Shell-sort with Hungarian (Székely) folk dance (<https://www.youtube.com/watch?v=CmPA7zE8mx0>)
- Merge-sort with Transylvanian-saxon (German) folk dance ([https://www.youtube.com/watch?v=XaqR3G\\_NVoo](https://www.youtube.com/watch?v=XaqR3G_NVoo))
- Quick-sort with Hungarian (Küküllőmenti legényes) folk dance (<https://www.youtube.com/watch?v=ywWBy6J5gz8>)

# Comparable

**Problem:** Sortera en array med objekt

Ofta är det **arrayer med objekt** som man sorterar. För att sortera arrayer måste objekten kunna jämföras.

Klasser som implementerar **Comparable** medger sortering

## Exempel

```
public class Commodity implements Comparable { // Comparable<Commodity>
    private String name;
    private double price;

    :

    public int compareTo( Object obj ) { // compareTo(Commodity c)
        Commodity c = ( Commodity )obj; //
        if( this.price < c.price )
            return -1;
        else if( this.price == c.price )
            return 0;
        else
            return 1;
    }
```

# Comparable – jämföra två Commodity-objekt

```
Commodity c1 = new Commodity("D", 12.25);  
Commodity c2 = new Commodity("B", 8.90);
```

```
Comparable comp = ( Comparable )c1;  
int res = comp.compareTo( c2 );  
if( res == -1 )  
    System.out.println( c1 + "\n" + c2);  
else if( res == 0 )  
    System.out.println("Lika stora");  
else  
    System.out.println( c2 + "\n" + c1);
```

## Resultat

B: 8.9

D: 12.25



# Comparable – i insertionsort

I *insertionsort* ser jämförelsen mellan två *int-element* ut så här:

```
public static void insertionsort(int[] array) {  
    for( int i = 1; i < array.length; i++ ) {  
        for ( int j = i; ( j > 0 ) && ( array[j] < array[j-1] ) ; j-- ) {  
            Utility.swap( array, j, j-1 );  
        }  
    }  
}
```

I *insertionsort* ser jämförelsen mellan *två objektreferenser* ut så här:

```
public static void insertionsort(Object[] array) {  
    Comparable comp;  
    for( int i = 1; i < array.length; i++ ) {  
        comp = (Comparable)array[i];  
        for ( int j = i; ( j > 0 ) && ( comp.compareTo( array[j-1] ) < 0 ) ; j-- ) {  
            Utility.swap( array, j, j-1 );  
        }  
    }  
}
```

eller kortare:

```
public static void insertionsort(Object[] array) {  
    for( int i = 1; i < array.length; i++ ) {  
        for ( int j = i; ( j > 0 ) && ( ( (Comparable)array[j] ).compareTo( array[j-1] ) < 0 ) ; j-- ) {  
            Utility.swap( array, j, j-1 );  
        }  
    }  
}
```

# Comparator – sortera objekt

En klass som implementerar interfacet Comparator kan hjälpa till vid sortering. Detta är aktuellt om:

Den naturliga sorteringsordningen (Comparable) inte ger den sortering som man önskar.

Objekten som ska sorteras implementerar inte Comparable.

Att sortera Commodity-objekt avtagande med avseende på priset:

```
import java.util.Comparator;
```

```
public class PriceDescending implements Comparator { // ... implements Comparator<Commodity> {  
    public int compare( Object obj1, Object obj2 ) { // ... compare(Commodity c1, commodity c2) {  
        Commodity c1 = ( Commodity )obj1;  
        Commodity c2 = ( Commodity )obj2;  
        double res = c1.getPrice() - c2.getPrice();  
        if( res < 0 )  
            return 1;  
        else if( res == 0 )  
            return 0;  
        else  
            return -1;  
    }  
}
```

# Comparator – i bubblesort

## Bubblesort för int-array

```
public static void bubblesort(int[] array) {  
    for( int i=0; i < array.length - 1; i++ ) {  
        for( int j = array.length - 1; j > i; j-- ) {  
            if( array[ j ] < array[ j - 1 ] )  
                Utility.swap( array, j, j - 1 );  
        }  
    }  
}
```

## Bubblesort för Object-array med hjälp av comparator-implementering

```
public static void bubblesort(Object[] array, Comparator comp) {  
    for( int i=0; i < array.length - 1; i++ ) {  
        for( int j = array.length - 1; j > i; j-- ) {  
            if( comp.compare( array[ j ], array[ j - 1 ] ) < 0 )  
                Utility.swap( array, j, j - 1 );  
        }  
    }
```