SORTING

Introduction

- Sorting data
  - simple algorithms have poor performance
  - To maximise performance more complex algorithms needed
- We consider two well known techniques for sorting data
- Bubble Sort
- Quick Sort

Bubble Sort

- Assume we are sorting a list represented by an array A of n integer elements
- Bubble sort algorithm in pseudo-code

```
FOR every element in the list,
    proceeding from the first to the last
DO
   WHILE list element > next list element
      bubble next element back (up) the list
      by successive swapping with
      the element just above/prior it
```

- The smallest values bubble up to the top

Bubble Sort

10	9	8	11	4

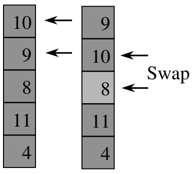
Bubble Sort

10	←
9	← Swap
8	
11	
4	

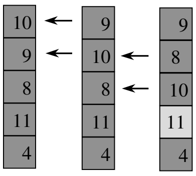
Bubble Sort

10	←	9
9	←	10
8		8
11		11
4		4

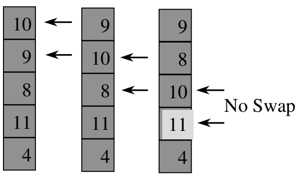
Bubble Sort



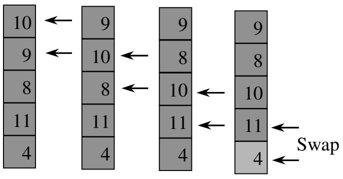
Bubble Sort



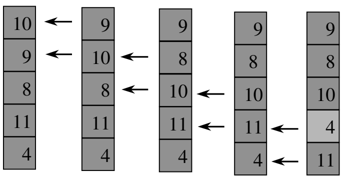
Bubble Sort



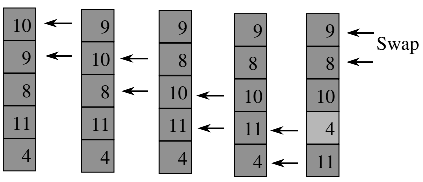
Bubble Sort



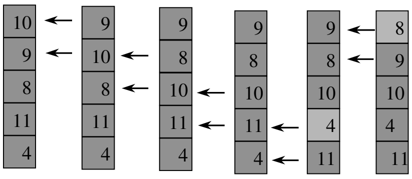
Bubble Sort



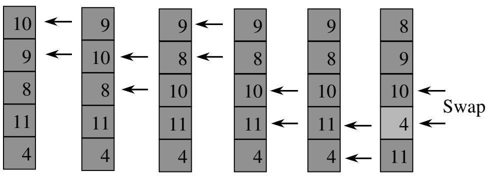
Bubble Sort



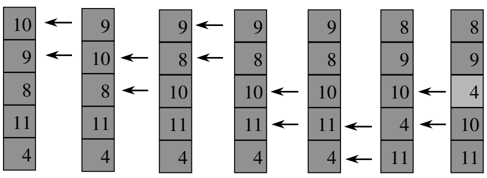
Bubble Sort



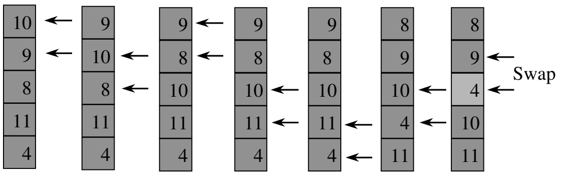
Bubble Sort



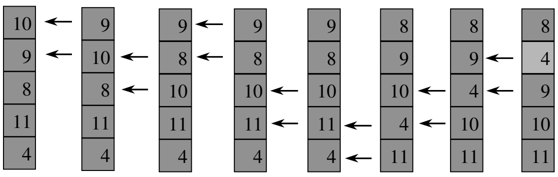
Bubble Sort



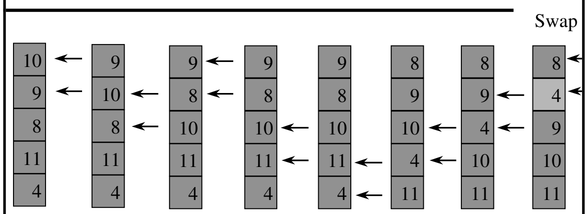
Bubble Sort



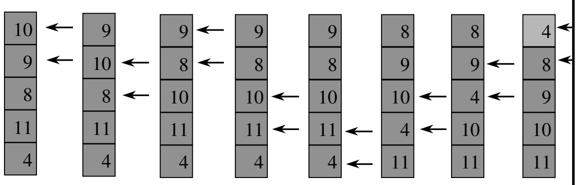
Bubble Sort



Bubble Sort



Bubble Sort



Implementation of bubbleSort()

```
public void bubbleSort(Comparable [] theArray, int n)
   {
```


Implementation of bubbleSort()

```
for(int index=0; index < n-pass; ++index) {



  if(theArray[index].compareTo(theArray[nextIndex]) > 0)
 {



}
 }
}
}
```

Performance of Bubblesort

- For an n element array the bubble sort loop is executed approximately n^2 times. We say that the running time is of the order of n^2 and write it as $O(n^2)$.
  - The analysis of running time is part of complexity theory see next section.
  - This gives us a method of comparing algorithms for efficiency.
- Are there any better sorting algorithms??
  - Quicksort

Quicksort

- The Quicksort algorithm was developed by C.A.R. Hoare. It has the best average behaviour in terms of complexity:

  Average case: $O(n \log_2 n)$
  Worst case: $O(n^2)$
- Worst case occurs for sequences that are already sorted or nearly sorted.

Quicksort

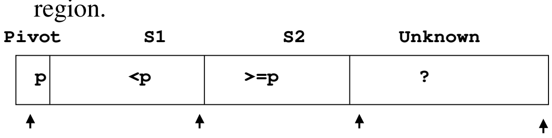
- Given a list of elements,
- take a partitioning element (pivot)
- and create two regions S1 and S2
  - such that all elements less than the pivot are in S1,
  - and all elements greater than the pivot are in S2.
- Now repeat this partitioning effort on each of these two regions

Quicksort

- The partitioning element is then placed between the resultant sublists (which are then partitioned in the same manner)
- The partitioning arranges the array so that `S1` is `theArray[first..pivotIndex-1]` and `S2` is `theArray[pivotIndex+1..last]`.

Quicksort

- We place the pivot in **`theArray[first]`** before partitioning and items which have yet to be placed in regions S1 or S2 are in the *unknown* region.



Quicksort

- Throughout the entire partitioning process:
  - The items in S1 are less than the pivot, and those in S2 are all greater than the pivot.
- We initialise the indices at the start to:
  - **`lastS1 = first`**
  - **`firstUnknown = first + 1`**

Quicksort

- At each step of the partitioning, you examine one item of the unknown region, determine whether it belongs to S1 or S2, and place it there.

Pivot **Unknown**

P	?

Quicksort

- To move `theArray[firstUnknown]` into `S1:`
  - `swap theArray[firstUnknown] with theArray[lastS1+1];`
  - `Increment lastS1;`
  - `Increment firstUnknown;`
- To move `theArray[firstUnknown]` into `S1:`
  - `Increment firstUnknown;`

Partition Algorithm - (pseudo code)

```
partition)theArray, first, last)
//Returns index of pivot after partitioning
  theArray[first..last]
pivot = theArray[first]; lastS1=first; firstUnknown=first+1;

   WHILE (firstUnknown <=last)  {
         if (theArray[firstUnknown] < pivot)
              Move theArray[firstUnknown] into S1
         else
              Move theArray[firstUnknown] into S2
      } //end while
   // place pivot in proper position
   swap theArray[firs] with theArray[lastS1]
   return lastS1  // returns index of the pivot
```

Quicksort, an example

Original Array (Pivot ↓ at first element)

27	38	12	39	27	16

Pivot | Unknown

27	38	12	39	27	16

```
firstUnknown=1
(points to 38)
```

```
38 belongs in
S2
```


Quicksort, an example

```
S1 is empty
```

```
12 belongs in
S1
```

```
so swap 38 and
12
```

Pivot | S2 | Unknown

27	38	12	39	27	16

Pivot | S1 | S2 | Unknown

27	12	38	39	27	16

```
39 belongs in
S2
```

Pivot | S1 | S2 | Unknown

27	12	38	39	27	16

```
27 belongs in
S2
```


Quicksort, an example

Pivot | S1 | S2 | Unknown

27	12	38	39	27	16

```
16 belongs in S1
```

```
so swap 38 and
16
```

Pivot | S1 | S2

27	12	16	39	27	38

```
S1 and S2 are
determined
```

S1 | Pivot | S2

27	12	16	39	27	38

```
Place pivot
between S1 and
S2
```

Implementation of Partition

```
private static int partition(Comparable
  [] theArray, int first, int last) {
```



```
for(int firstU =
  first+1;firstU<=last;firstU++) {



     }
}
```



```
tempItem = theArray[first];
theArray[first] = theArray[lastS1];
theArray[lastS1] = tempItem;

return lastS1;
}
```

How to Implement Quicksort

- We now must sort the sublists so recursively call the quicksort function
  - firstly the left sublist
  - QuickSort(theArray, first, pivotIndex-1);
  - Then the right sublist
  - QuickSort(theArray, pivotIndex+1, last);


```
public void quickSort(Comparable []
  theArray, int first, int last) {
int pivotIndex;

if(first < last) {




}
}
```