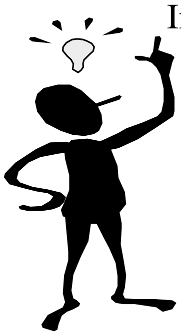
Sorting Techniques

Some of the sorting techniques used to arrange data are as follows :

- Insertion Sort
- Selection Sort
- Bubble Sort
- Quicksort
- Heapsort
- Mergesort

Insertion Sort



The general idea of the insertion sort method is that for each element, find the slot where it belongs.

Example

Array	**G**	**D**	**Z**	**F**	**B**	**E**
Index	0	1	2	3	4	5

The element in position Array[0] is certainly sorted.

Thus, move on to insert the second character, D,

into the appropriate location to maintain the

alphabetical order.

How does it work?

- Each element Array[j] is taken one at a time from j = 0 to n-1.
- Before insertion of Array[j], the subarray from Array[0] to Array[j-1] is sorted, and the remainder of the array is not.
- After insertion, Array[0…j] is correctly ordered, while the subarray with elements Array[j+1]…..Array[n-1] is unsorted.

Insertion Sort technique.

Array	**D**	**G**	**Z**	F	B	E
Index	0	1	2	3	4	5
Array	**D**	**F**	**G**	**Z**	B	E
Index	**0**	**1**	**2**	**3**	**4**	**5**
Array	**B**	**D**	**F**	**G**	**Z**	E
Index	0	1	2	3	4	5
Array	**B**	**D**	**E**	**F**	**G**	**Z**
Index	**0**	**1**	**2**	**3**	**4**	**5**

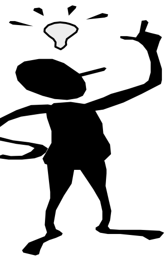
Insertion Sort Algorithm

```
for i = 1 to n-1
    temp = a[i]
    loc = i
    while(loc>0  &&  (a[loc-1]>
  temp)
        a[loc] = a[loc-1]
        loc = loc -1
    a[loc] = temp
```

Complexity

- Best situation: the data is already sorted. The inner loop is never executed, and the outer loop is executed n – 1 times for total complexity of O(n).
- Worst situation: data in reverse order. The inner loop is executed the maximum number of times. Thus the complexity of the insertion sort in this worst possible case is quadratic or $O(n^2)$.

Selection Sort



```
The general idea of
the  selection sort
is that for each
slot, find the
element that
belongs there.
```

Selection Sort Algorithm

```
for i = 0 to n-2
  temp = a[i]
  loc = i;
  for j = i+1 to n-1
    if a[j] < a[loc]
        loc = j
  a[i] = a[loc]
  a[loc] = temp
```

- The inner loop in the above algorithm finds the location of the next smallest element in the array (the location of the next smallest element in the array).
- The outer loop moves along the array as the elements are sorted.
- What is the complexity of selection sort?

Bubble Sort

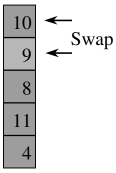
- Assume we are sorting a list represented by an array A of n integer elements
- Bubble sort algorithm in pseudo-code

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

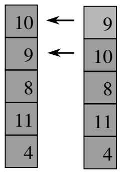
Bubble Sort

10	9	8	11	4

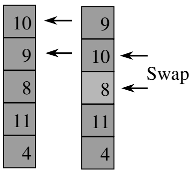
Bubble Sort



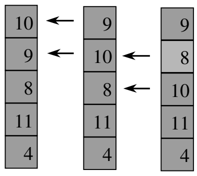
Bubble Sort



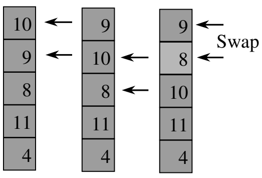
Bubble Sort



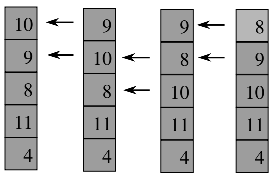
Bubble Sort



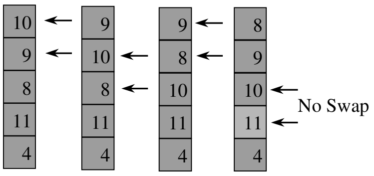
Bubble Sort



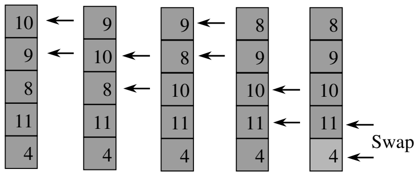
Bubble Sort



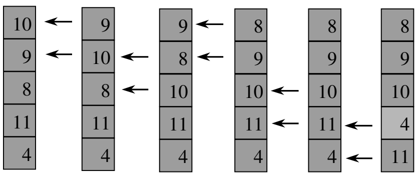
Bubble Sort



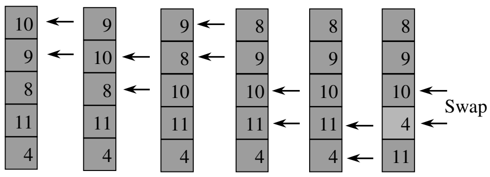
Bubble Sort



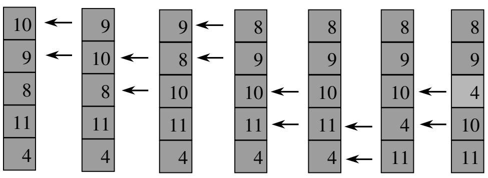
Bubble Sort



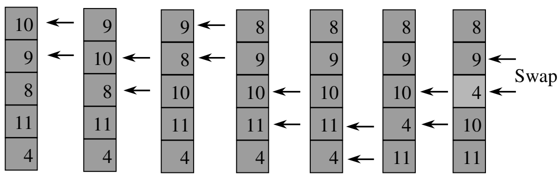
Bubble Sort



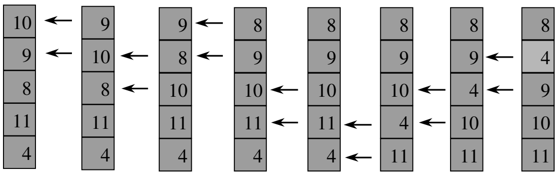
Bubble Sort



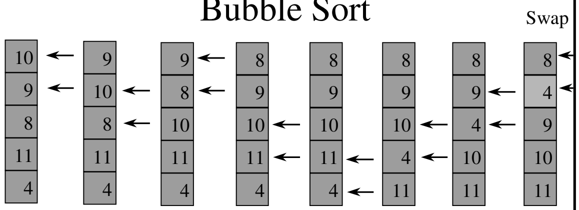
Bubble Sort



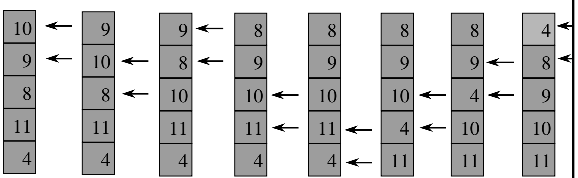
Bubble Sort



Bubble Sort



Bubble Sort



Bubble Sort Algorithm

```
FOR i = 1 to n-1 DO
   j=i; (*look at current pair & see if we can swap*)
   WHILE (A[j] > A[j+1])DO
      temp := A[j];  (* swap *)
      A[j] := A[j+1];
      A[j+1] := temp;

      IF j<> 0  (* now look at next pair back *)
         THEN
            j := j-1;
      END
   END
END
```

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: $O(n^2)$

Quicksort

Given a list of elements,
- take a partitioning element
- and create a (sub)list
  - such that all elements to the left of the partitioning element are less than it,
  - and all elements to the right of it are greater than it.
- Now repeat this partitioning effort on each of these two sublist

Quicksort

- And so on in a recursive manner until all the sublists are empty, at which point the (total) list is sorted
- Partitioning can be effected simultaneously, scanning left to right and right to left, interchanging elements in the wrong parts of the list
- The partitioning element is then placed between the resultant sublists (which are

Implementation of Quicksort

```
If anything to be partitioned
THEN
   choose a pivot
   REPEAT
      scan from left to right until we find an element
      > pivot: i points to it

      scan from right to left until we find an element
      < pivot: loc points to it

      IF  i < loc
         THEN
            exchange pivot and locth element
   UNTIL i > loc
```

Implementation of Quicksort()

```
exhange pivot and loc element

partition from 1st to loc-1th elements
(* i.e. quicksort 1st to locth *)

partition from locth to rth elements
(i.e. quicksort locth to rth *)
```

Example

```
G B  S    E    V    M    P    H    D    C
     ?
0 1  2    3    4    5    6    7    8    9
  10
```

```
i = 0
first = 0
loc = 10
last = 9
```

```
G B  C    E    V    M    P    H    D    S
     ?
0 1  2    3    4    5    6    7    8    9
  10

i = 2
first = 0
loc = 9
last = 9
```

- *i* stops at 4 (a[4] or V>= G), and *loc* at 8 (a[8] or D <= G). These values are then swapped.

```
G B  C    E    D    M    P    H    V    S
     ?
0 1  2    3    4    5    6    7    8    9
  10

i = 4
first = 0
```

- i and loc continue to move until i > loc
- This happens when i points to M and loc points to D

```
G B  C    E    D    M    P    H    V    S
     ?
0 1  2    3    4    5    6    7    8    9
  10

i = 5
first = 0
loc = 4
```

Algorithm for *partition(a,first,last,loc)*

```
i = first
loc = last + 1
pivot = a[first]
```

Recursive Quicksort Algorithm

```
if  first < last then
  partition(a,first,last,loc)
  quicksort(a,first,loc-1)
  quicksort(a,loc+1,last)
```

The pivot can be any of the array elements, such as the leftmost one.

```
Pivot = a[first]
```

- Two indicies need to be initialised, *i* and *loc*, to indicate the outside bounds, *first* and *last+1* , respectively of the elements to be divided into two collections.

```
i = first
loc = last +1
```

- Although `last+1` may be out of bounds for array `a`, there is no error because `a[last+1]` is never accessed, because `loc` is decremented first before referencing `a[loc]`.
- In the partition procedure, we wish to have the elements less than or equal to the pivot

Quicksort

Given a list of elements,

- take a partitioning element
- and create a (sub)list
  - such that all elements to the left of the partitioning element are less than it,
  - and all elements to the right of it are greater than it.
- Now repeat this partitioning effort on each of these two sublist

Quicksort

- And so on in a recursive manner until all the sublists are empty, at which point the (total) list is sorted
- Partitioning can be effected simultaneously, scanning left to right and right to left, interchanging elements in the wrong parts of the list
- The partitioning element is then placed between the resultant sublists (which are

The Partition

```
do      {do {i = i+1;} while (a[i] < pivot && (i <
   last));
               do {loc = loc-1;} while (a[loc] > pivot);
               if (i < loc)
               {        swap (a(i), a(loc))
               }
   } while (i < loc);
   swap (a(first), a(loc))
}
```

Recursive Quicksort Algorithm

```
if  first < last then
  partition(a,first,last,loc)
  quicksort(a,first,loc-1)
  quicksort(a,loc+1,last)
```

The pivot can be any of the array elements, such as the leftmost one.

```
Pivot = a[first]
```

Divide and Conquer

- Quicksort is a divide and Conquer Algorithm
- A divide-and-conquer algorithm is one that divides the problem into smaller problems upon which it performs the same process.

Quicksort

10	9	8	11	4	99

sentinel

Quicksort

10	9	8	11	4	99

```
QS(A, , )

L:
R:
i:
j:
pivot:
```

↑ ↑
i j

Quicksort

10	9	8	11	4	99

i j

```
QS(A,1,6)

L:      1
R:      6
i:      1
j:      6
pivot: 10
```

Quicksort

10	9	8	11	4	99

i j

```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4
j:      6 5
pivot: 10
```

Quicksort

10	9	8	4	11	99

i j

```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4
j:      6 5
pivot: 10
```

Quicksort

10	9	8	4	11	99

j i

```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4 5
j:      6 5 4
pivot: 10
```

Quicksort

4	9	8	10	11	99

j i

```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4 5
j:      6 5 4
pivot: 10
```

Quicksort

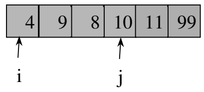
4	9	8	10	11	99

i j

```
QS(A,1,6)           QS(A,1,4)          QS(A,5,6)

L:      1           L:      1          L:      5
R:      6           R:      4          R:      6
i:      1 2 3 4 5   i:                 i:
j:      6 5 4       j:                 j:
pivot: 10           pivot:  4          pivot:  11
```

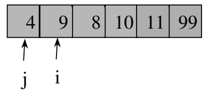
Quicksort




```
QS(A,1,4)                 QS(A,5,6)

L:      1                 L:      5
R:      4                 R:      6
i:      1                 i:      5
j:      4                 j:      6
pivot:  4                 pivot:  11
```

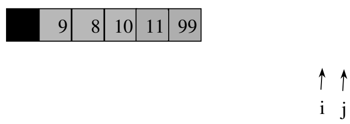
Quicksort




```
QS(A,1,4)                 QS(A,5,6)

L:      1                 L:      5
R:      4                 R:      6
i:      1 2               i:      5
j:      4 3 2 1           j:      6
pivot:  4                 pivot:  11
```

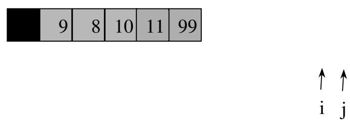
Quicksort




```
QS(A,1,4)       QS(A,1,1)     QS(A,2,4)     QS(A,5,6)

L:      1       L:      1     L:      2     L:      5
R:      4       R:      1     R:      4     R:      6
i:      1 2     i:            i:            i:      5
j:      4 3 2 1 j:            j:            j:      6
pivot:  4       pivot:  4     pivot:  9     pivot:  11
```

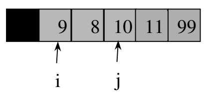
Quicksort




```
QS(A,1,1)                 QS(A,2,4)     QS(A,5,6)

L:      1                 L:      2     L:      5
R:      1                 R:      4     R:      6
i:                        i:            i:      5
j:                        j:            j:      6
pivot:  4                 pivot:  9     pivot:  11
```

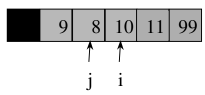
Quicksort




```
QS(A,2,4)                 QS(A,5,6)

L:      2                 L:      5
R:      4                 R:      6
i:      2                 i:      5
j:      4                 j:      6
pivot:  9                 pivot:  11
```

Quicksort




```
QS(A,2,4)                 QS(A,5,6)

L:      2                 L:      5
R:      4                 R:      6
i:      2 3 4             i:      5
j:      4 3               j:      6
pivot:  9                 pivot:  11
```

Quicksort

	8	9	10	11	99

i j

	QS(A,2,4)	QS(A,2,3)	QS(A,4,4)	QS(A,5,6)
L:	2	2	4	5
R:	4	3	4	6
i:	2 3 4			5
j:	4 3			6
pivot:	9	8	10	11

Quicksort

	8	9	10	11	99

i j

	QS(A,2,3)	QS(A,4,4)	QS(A,5,6)
L:	2	4	5
R:	3	4	6
i:	2		5
j:	3		6
pivot:	8	10	11

Quicksort

	8	9	10	11	99

j i

	QS(A,2,3)	QS(A,4,4)	QS(A,5,6)
L:	2	4	5
R:	3	4	6
i:	2 3		5
j:	3 2		6
pivot:	8	10	11

Quicksort

	8	9	10	11	99

i j

	QS(A,2,3)	QS(A,2,2)	QS(A,3,3)	QS(A,4,4)	QS(A,5,6)
L:	2	2	3	4	5
R:	3	2	3	4	6
i:	2 3				5
j:	3 2				6
pivot:	8	8	9	10	11

Quicksort

	8	9	10	11	99

i j

	QS(A,4,4)	QS(A,5,6)
L:	4	5
R:	4	6
i:		5
j:		6
pivot:	10	11

Quicksort

	8	9	10	11	99

i j

	QS(A,5,6)	QS(A,5,5)	QS(A,6,6)
L:	5	5	6
R:	6	5	6
i:	5		
j:	6		
pivot:	11	11	99