Sorting

- Sorting is arranging items into ascending or descending order.
- Various algorithms are available
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Merge sort
  - Quick sort

Bubble Sort

- Simplest of all sorting algorithm
  - Scan array by exchange adjacent elements that are out of order
  - Continue this for several passes until the array is sorted
- Easy to implement

Steps

- On each pass, compares each pair of adjacent items and swap them if they are in wrong order.
- At the end of first round largest value is at last position.
- After that, next round is with the remaining elements.
- At the end of second round , the next largest value comes to the second last position (one before the last position)
- Going to continue like this until no swaps are needed.

Pseudo code

```
bubblesort(data[],n)
      for(i=0; i<n-1; i++)
            for(j=n-1; j>i; --j)
            {
                  Swap elements in position j and j-1
                  If they are out of order;
            }
```

C Program code

```
#include<stdio.h>
int main()
{
    int i, m[9] = {1,8,6,4,10,5,3,2,25}, temp, swap=1;
    printf("Unsorted Array \n");
    for(i=0; i<=8; i++)
    {
        printf("%d ", m[i]);
    }

    while(swap>0)
    {
        swap=0;
        for(i=0; i<=8; i++)
        {
            if(m[i]>m[i+1])
            {
                temp = m[i];
                m[i]=m[i+1];
                m[i+1]=temp;
                swap++;
            }
        }
    }
    printf("Sorted Array \n");
    for(i=0; i<=8; i++)
    {
        printf("%d ", m[i]);
    }

}
```

Complexity

Big O of this algorithm : n^2

- What is the worst case scenario?
- Inner loop has n-1 turns
- Outer loop also runs for n-1 turn for the worst case
- Body of the inner loop (a single turn) is constant time
- Complexity a (n-1)*(n-1)*k
- Complexity is $O(n^2)$

Example 1:

(5 1 4 2 8)

Pictorial Representation

1st round (1st pass)

(5 1 4 2 8) -----> (1 5 4 2 8)

(1 5 4 2 8) -----> (1 4 5 2 8)
(1 4 5 2 8) -----> (1 4 2 5 8)
(1 4 2 5 8) -----> (1 4 2 5 8)
swaps=3

2nd round
(1 4 2 5 8) -----> (1 4 2 5 8)
(1 4 2 5 8) -----> (1 2 4 5 8)
(1 2 4 5 8) -----> (1 2 4 5 8)
(1 2 4 5 8) -----> (1 2 4 5 8)
swaps=1

Already sorted, but it doesn't know it, because the value in swaps variable is not zero. So have to do another pass without a swap.

3rd round
(1 2 4 5 8) -----> (1 2 4 5 8)
(1 2 4 5 8) -----> (1 2 4 5 8)
(1 2 4 5 8) -----> (1 2 4 5 8)
(1 2 4 5 8) -----> (1 2 4 5 8)
swaps=0

End of the algorithm. Now we have a sorted array.

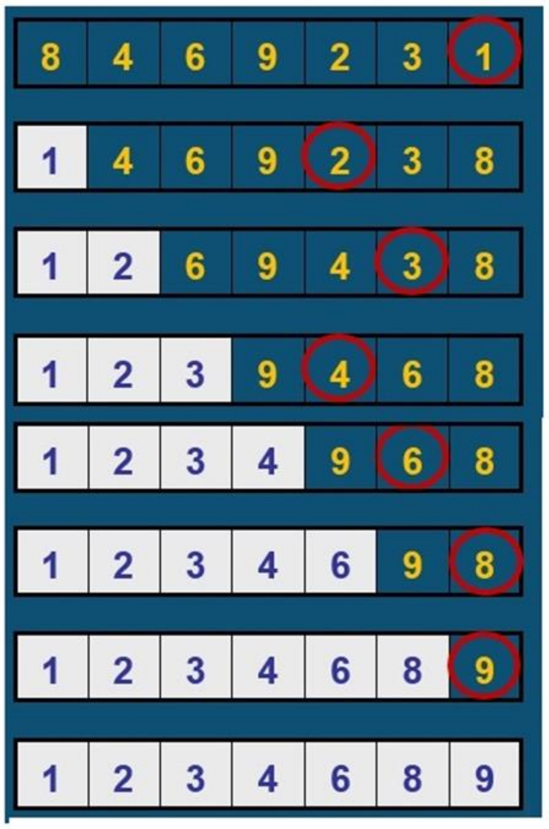
Example 2 :
(1 8 6 4 10 5 3 2 25)

Selection Sort (Opposite of Bubble)

Find first smallest value → put it at first position

Find second smallest value → put it at second position

Continue until all elements are arranged in order.
Same number of comparisons as bubble sort but less number of swaps.



Pseudocode

```
selectionSort(A[], n)
      for(i=0; i<n-1; i++)
            Select the smallest element among A[i],...,A[n-1]
            Swap it with A[i]
```

C programming code

```
      #include <stdio.h>
      int minIndex(float d[], int size)
      {
          int i,minIn=0;
          float min=d[0];
          for(i=1; i<size; i++)
              if(d[i]<min)
              {
                  minIn=i;
                  min=d[i];
              }
```

```
        return minIn;
    }
    void swap(float *p1, float* p2)
    {
        float temp;
        temp=*p1;
        *p1=*p2;
        *p2=temp;
    }

    void selectionSort(float d[], int size)
    {
        if(size>1)
        {
            swap(&d[0],&d[minIndex(&d[0],size)]);
            selectionSort(&d[1],size-1);
        }
    }
    int main()
    {
        int i;
        float no[6] = {2,4,5,3,1,9};
        printf("Unsorted Array \n");
        for(i=0; i<6; i++)
        {
            printf("%0.0f ", no[i]);
        }
        printf("\nSorted Array \n");
        selectionSort(no,6);
        for(i=0; i<6; i++)
        {
            printf("%0.0f ", no[i]);
        }
        return 0;
    }
```

Complexity

Big O of this algorithm : n^2

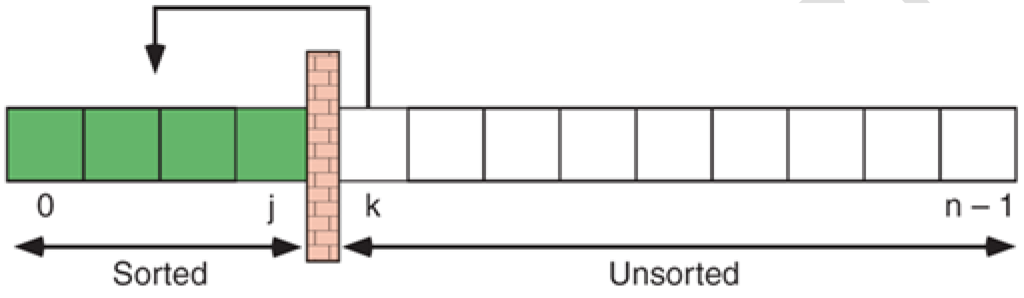
- swap() is constant time → O(1)
- minIndex() → O(n)

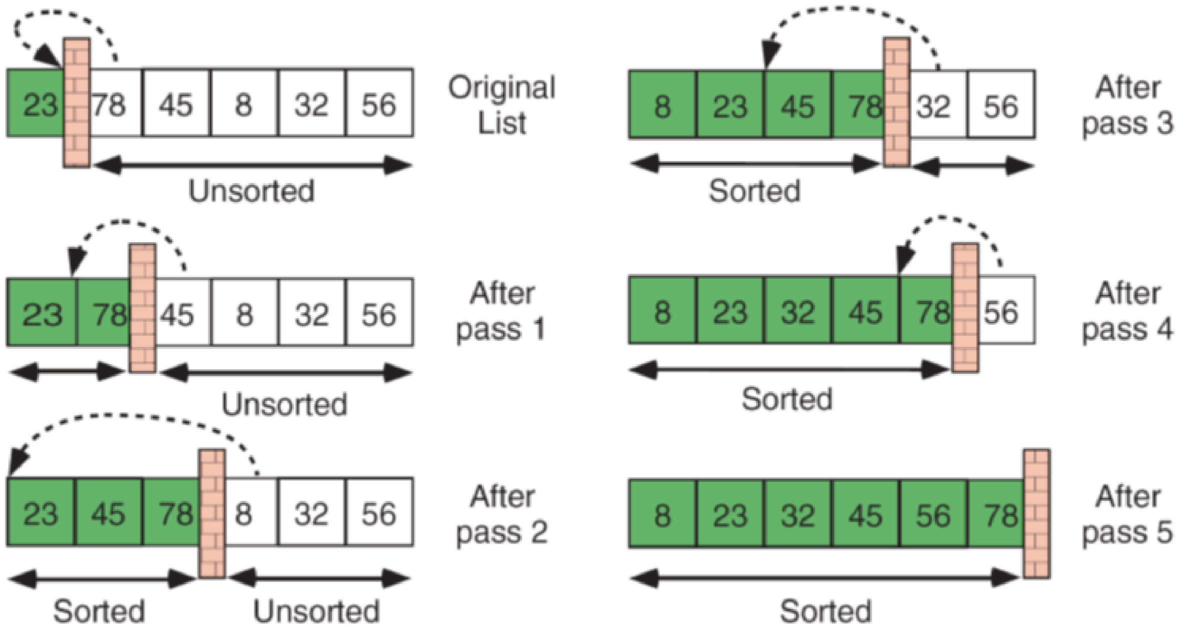
- selectionSort is called n times for a prob. of size n, and in each minIndex is called once.
    - i.e. Complexity is O(n^2)
- minIndex is actually called for degreasing array sizes; n, (n-1), (n-1)…
    - Average array/loop size is still n/2 à O(n/2)=O(n)

Direct Insertion Sort

- Pick up and drop in the correct place.

Example 1 :





Pseudocode

```
insertionSort(A[],n)
      for(i=1; i<n; i++)
```

```
            Move all elements A[j] greater than A[i] by 1 position
            Place A[i] in its proper position.
```

C programming code

```
#include<stdio.h>
int main()
{
    int i, j, m[6]= {23,78,45,8,32,56}, temp;
    printf("Unsorted Array \n");
    for(i=0; i<6; i++)
    {
        printf("%d ", m[i]);
    }

    for(i=1; i<6; i++)
    {
        temp=m[i];
        j=i-1;
        while((temp<m[j])&&(j>=0))
        {
            m[j+1]=m[j];
            j=j-1;
        }
        m[j+1]=temp;
    }
    printf("\nSorted Array \n");
    for(i=0; i<6; i++)
    {
        printf("%d ", m[i]);
    }
}
```

Example 2 :

(5 3 1 4 2)

Complexity

Big O of this algorithm : n^2

Merge Sort

Divide array

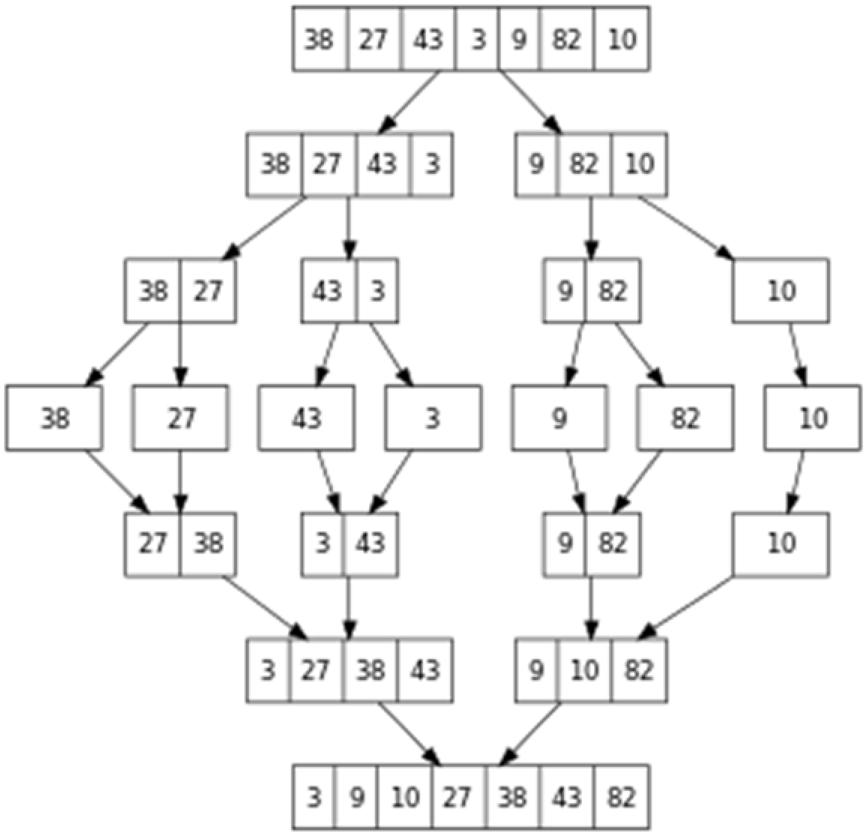
Left half : from (firstIndex) to (midIndex)

Right half : from ((mid+1) index) to (lastIndex)

How to find midIndex
midIndex=(firstIndex+lastIndex)/2

When firstIndex<lastIndex , still we have to partition
When firstIndex==lastIndex , end of partition
Then we come to merge.
Sort and merge each part.
Once you merge one, then you go to the right of it.

Example 1 :



Example 2 :
(1 8 6 4 10 5 3 2 22)

Pseudo Code

```
mergeSort(A)
If A have at least 2 elements
      mergeSort(left half of A)
      mergeSort(right half of A)
```

```
    merge(both halves into a sorted list)

merge(array1, array2, array3)
    i1, i2, i3 are properly initialized
    While both array2 and array3 contain elements
        If array2[i2] < array3[i3]
            array[i1++] = array2[i2++]
        Else
            array[i1++] = array3[i3++]
        Load into array1 the remaining elements of either array2
        array3.
```

Complexity

Big O of this algorithm : nlogn

A small list take few steps to sort than a large list.
Fewer steps goes to construct a sorted list from two sorted lists than from two unsorted lists.

Question

PNG Co. receives approximately around 1000 files from branches around the world which contains exactly five integers in sorted order. (Four of these files are given below.) The management requires all the files to be put into single large sorted file so it is easier for processing. Explain using a standard sorting algorithm how to put these 1000 files into a single sorted file. (Clearly state your assumption if any.)

File A : 55, 105, 2683, 7896, 10000
File B : 52, 100, 5653, 7866, 12500
File C : 26, 107, 4683, 7896, 10080
File D : 454, 4105, 5689, 7899, 15768

Quick Sort

Pick one element as pivot.
Re-order the list;

element values > pivot → put after pivot

element values < pivot → put before pivot

element values = pivot → can go either way

After partitioning; pivot is at its final position
Recursively sort the sub list of the lesser and greater elements.

Selecting Pivot

1. Choosing first index
   Less number of elements come its final position at a time.
2. Choosing middle index
   More cautious way, since many arrays to be sorted, already have many elements their proper position.

Question

Quicksort relies heavily on the **partition process** and there are many algorithms designed to partition a given array. If the **middle value** of all elements were chosen as a **pivot value** it would partition the array into **two consistent halves required for a quick sort to be efficient.** Provide your views sighting **running time complexity** of selecting the pivot as mentioned above by using an array of {2, 8, 6, 1, 10, 15, 3, 12, 25, 22} as an example.

Quicksort method 1 : take first index as pivot

hi	:	go backward from right until you get a **num <= pivot**
lo	:	go forward from left until you get a **num > pivot**

If **lo>=hi** , swap pivot and hi (out of the loop)
If **lo<hi** , swap lo and hi

Quicksort method 2 : take mid index as pivot

hi	:	go backward from right until you get a **num <= pivot**
lo	:	go forward from left until you get a **num > pivot**

If **lo>=hi** , swap pivot and hi (out of the loop)
If **lo<hi** , swap lo and hi

How to find the mid , if there are even number of elements?
First of all find the largest number of the array and put it in the last place. Do the quicksort to the rest of the elements.

Complexity

Big O of this algorithm : n^2