DSA(lecture#8)

Prepared By Bal Krishna Subedi

Sorting

Sorting is among the most basic problems in algorithm design. We are given a sequence of items, each associated with a given key value. The problem is to permute the items so that they are in increasing (or decreasing) order by key. Sorting is important because it is often the first step in more complex algorithms. Sorting algorithms are usually divided into two classes, internal sorting algorithms, which assume that data is stored in an array in main memory, and external sorting algorithm, which assume that data is stored on disk or some other device that is best accessed sequentially. We will only consider internal sorting. Sorting algorithms often have additional properties that are of interest, depending on the application. Here are two important properties.

In brief the sorting is a process of arranging the items in a list in some order that is either ascending or descending order.

Bubble Sort:

The basic idea of this sort is to pass through the array sequentially several times. Each pass consists of comparing each element in the array with its successor (for example a[i] with a[i + 1]) and interchanging the two elements if they are not in the proper order. For example, consider the following array:

Initially:

25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7

After Pass 1:

25	48	37	12	57	86	33	92
0	1	2	3	4	5	6	7

After Pass 2:

25	37	12	48	57	33	86	92
0	1	2	3	4	5	6	7

After Pass 3:

25	12	37	48	33	57	86	92
0	1	2	3	4	5	6	7

After Pass 4:

12	25	37	33	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 5:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 6:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 7:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

Here, we notice that after each pass, an element is placed in its proper order and is not considered in succeeding passes. Furthermore, we need n – 1 passes to sort n elements.

Algorithm

```
BubbleSort(A, n)
{
        for(i = 0; i <n-1; i++)
        {
                for(j = 0; j < n-i-1; j++)
                {
                        if(A[j] > A[j+1])
                        {
                                temp = A[j];
                                A[j] = A[j+1];
                                A[j+1] = temp;
                        }
                }
        }
}
```

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + …………………………… +2 +1

= $O(n^2)$

There is no best-case linear time complexity for this algorithm.

Space Complexity:

Since no extra space besides 3 variables is needed for sorting

Space complexity = O(n)

Selection Sort:

Idea: Find the least (or greatest) value in the array, swap it into the leftmost(or rightmost) component (where it belongs), and then forget the leftmost component. Do this repeatedly.
Let a[n] be a linear array of n elements. The selection sort works as follows:

pass 1: Find the location loc of the smallest element int the list of n elements a[0], a[1], a[2], a[3], ……...,a[n-1] and then interchange a[loc] and a[0].

Pass 2: Find the location loc of the smallest element int the sub-list of n-1 elements a[1], a[2], a[3], ……...,a[n-1] and then interchange a[loc] and a[1] such that a[0], a[1] are sorted.

…..................... and so on.

Then we will get the sorted list a[0]<=a[1]<= a[2]<=a[3]<= ……..<=a[n-1].

Algorithm:

```
SelectionSort(A)
{
        for( i = 0;i < n ;i++)
        {
                least=A[i];
                p=i;
                for ( j = i + 1;j < n ;j++)
                {
                        if (A[j] < A[i])
                        least= A[j]; p=j;
                }
        }
        swap(A[i],A[p]);
}
```

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + ………………………… +2 +1

$= O(n^2)$

There is no best-case linear time complexity for this algorithm, but number of swap operations is reduced greatly.

Space Complexity:

Since no extra space besides 5 variables is needed for sorting

Space complexity = O(n)

Insertion Sort:

Idea: like sorting a hand of playing cards start with an empty left hand and the cards facing down on the table. Remove one card at a time from the table, and insert it into the correct position in the left hand. Compare it with each of the cards already in the hand, from right to left. The cards held in the left hand are sorted

Suppose an array a[n] with n elements. The insertion sort works as follows:

pass 1: a[0] by itself is trivially sorted.

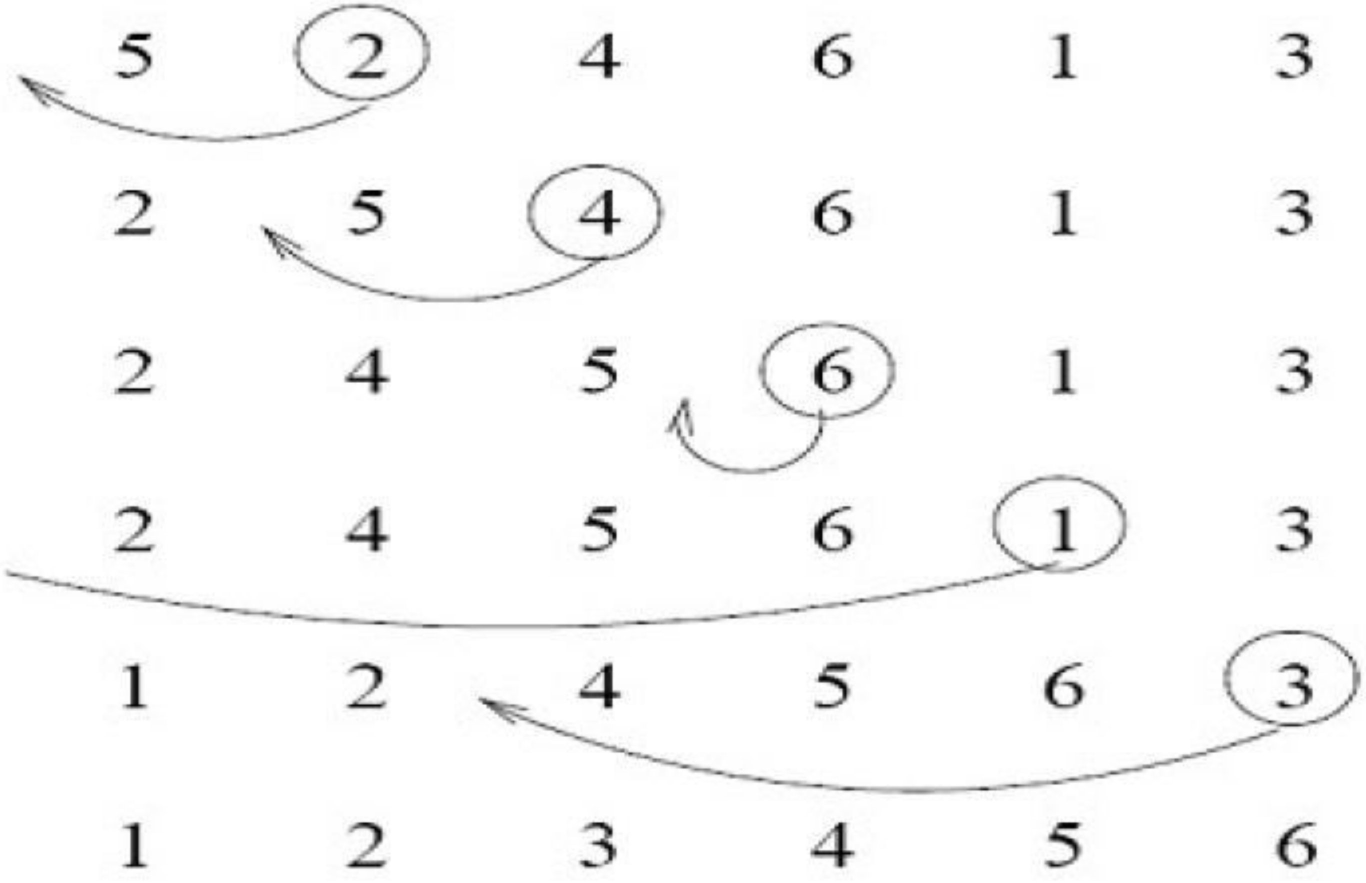
Pass 2: a[1] is inserted either before or after a[0] so that a[0], a[1] is sorted.

Pass 3: a[2] is inserted into its proper place in a[0],a[1] that is before a[0], between a[0] and a[1], or after a[1] so that a[0],a[1],a[2] is sorted.

……………………………………………

pass N: a[n-1] is inserted into its proper place in a[0],a[1],a[2],........,a[n-2] so that a[0],a[1],a[2],...........,a[n-1] is sorted with n elements.

Example:



Algorithm:

```
InsertionSort(A)
        {
                for (i=1;i<n;i++)
                {
                        key = A[ i]
                        for(j=i; j>0 && A[j] >key; j--)
                        {
                                A[j + 1] = A[j]
                        }
                        A[j + 1] = key

                }
        }
```

Time Complexity:

Worst Case Analysis:

Array elements are in reverse sorted order

Inner loop executes for 1 times when i=1, 2 times when i=2... and n-1 times when i=n-1:

Time complexity = 1 + 2 + 3 + ... +(n-2) +(n-1)

= $O(n^2)$

Best case Analysis:

Array elements are already sorted

Inner loop executes for 1 times when i=1, 1 times when i=2... and 1 times when i=n-1:

Time complexity = 1 + 2 + 3 + ... +1 +1

= $O(n)$

Space Complexity:

Since no extra space besides 5 variables is needed for sorting

Space complexity = $O(n)$

Quick Sort:

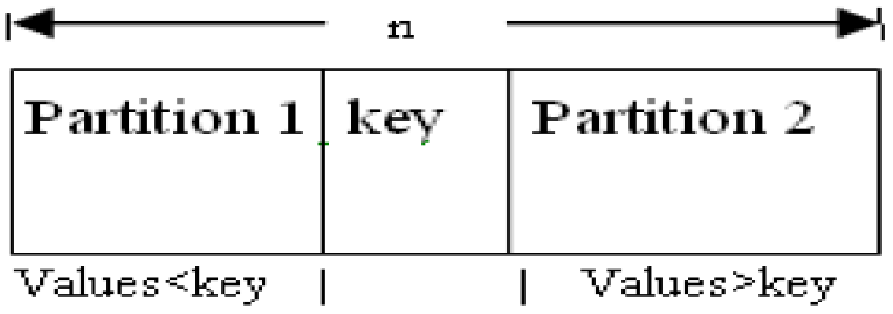
Quick sort developed by C.A.R Hoare is an unstable sorting. In practice this is the fastest sorting method. It possesses very good average case complexity among all the sorting algorithms. This algorithm is based on the divide and conquer paradigm. The main idea behind this sorting is partitioning of the elements.

Steps for Quick Sort:

Divide: partition the array into two nonempty sub arrays.

Conquer: two sub arrays are sorted recursively.

Combine: two sub arrays are already sorted in place so no need to combine.



Example: a[]={5, 3, 2, 6, 4, 1, 3, 7}

5	3	2	6	4	1	3	7	
x							y	
5	3	2	6	4	1	3	7	
			x			y	{swap x & y}	
5	3	2	3	4	1	6	7	
					y	x	{swap y and pivot}	
1	3	2	3	4	5	6	7	
					p			

(1 3 2 3 4) 5 (5 7)
and continue this process for each sub-arrays and finally we get a sorted array.

Algorithm:

```
QuickSort(A,l,r)
{
        f(l<r)
        {
                p = Partition(A,l,r);
                QuickSort(A,l,p-1);
                QuickSort(A,p+1,r);
        }
}

Partition(A,l,r)
{
        x =l;
        y =r ;
        p = A[l];
        while(x<y)
        {
                while(A[x] <= p)
                        x++;
                while(A[y] >=p)
                        y--;
                if(x<y)
                        swap(A[x],A[y]);
        }
        A[l] = A[y];
        A[y] = p;
        return y;        //return position of pivot
}
```

Time Complexity:

Best Case:

Divides the array into two partitions of equal size, therefore

$T(n) = 2T(n/2) + O(n)$, Solving this recurrence we get,

$T(n)=O(n\log n)$

Worst case:

when one partition contains the n-1 elements and another partition contains only one element.
Therefore its recurrence relation is:
$T(n) = T(n-1) + O(n)$, Solving this recurrence we get
$T(n)=O(n^2)$

Average case:

Good and bad splits are randomly distributed across throughout the tree
$T1(n)= 2T'(n/2) + O(n)$ Balanced
$T'(n)= T(n-1) + O(n)$ Unbalanced
Solving:

$B(n)= 2(B(n/2 -1) + \Theta(n/2)) + \Theta(n)$
$= 2B(n/2 -1) + \Theta(n)$
$= O(n\log n)$
$\Rightarrow T(n)=O(n\log n)$

Merge Sort

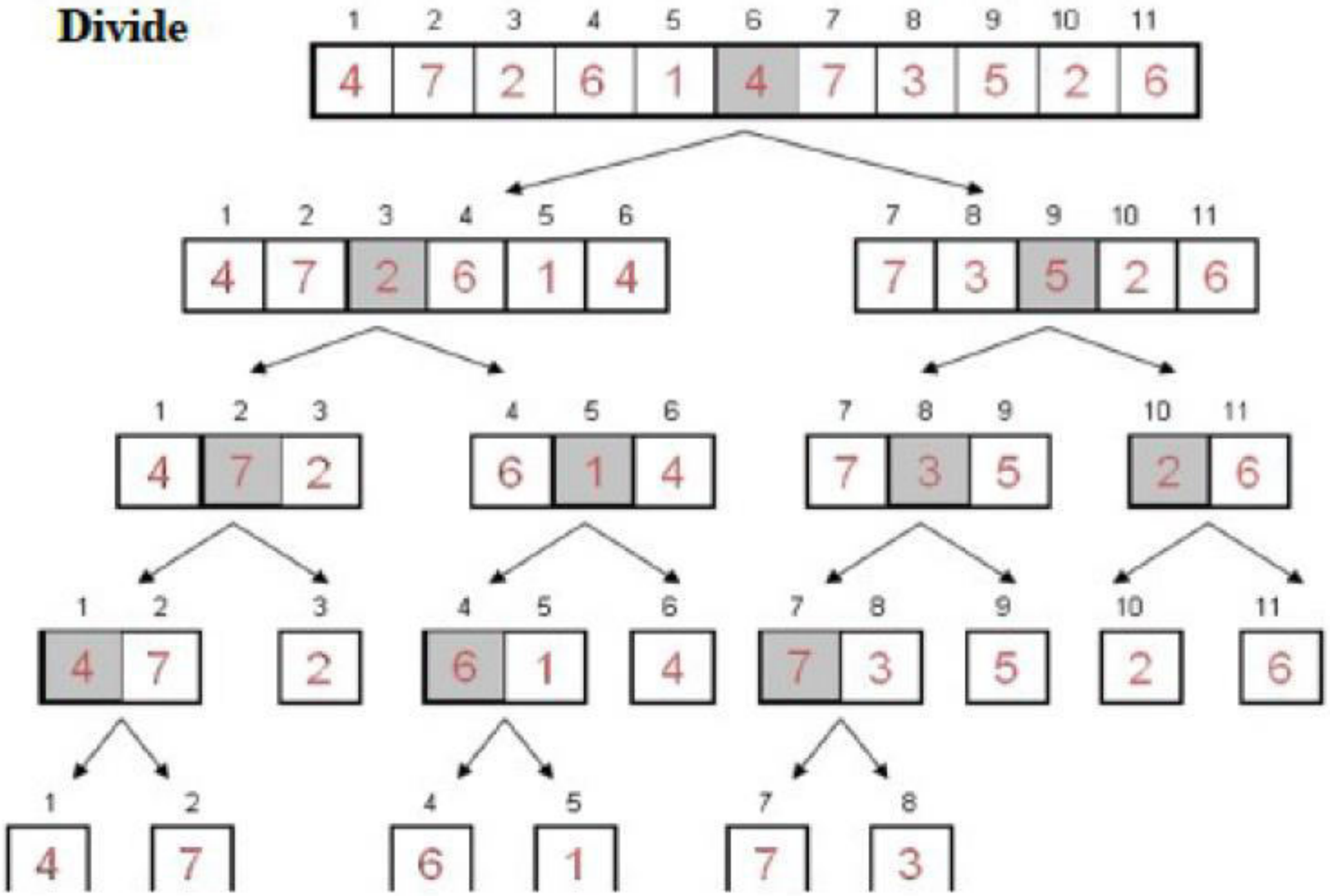
To sort an array A[l . . r]:

- **Divide**
  - – Divide the n-element sequence to be sorted into two sub-sequences of n/2 elements
- **Conquer**
  - – Sort the sub-sequences recursively using merge sort. When the size of the sequences is 1 there is nothing more to do
- **Combine**

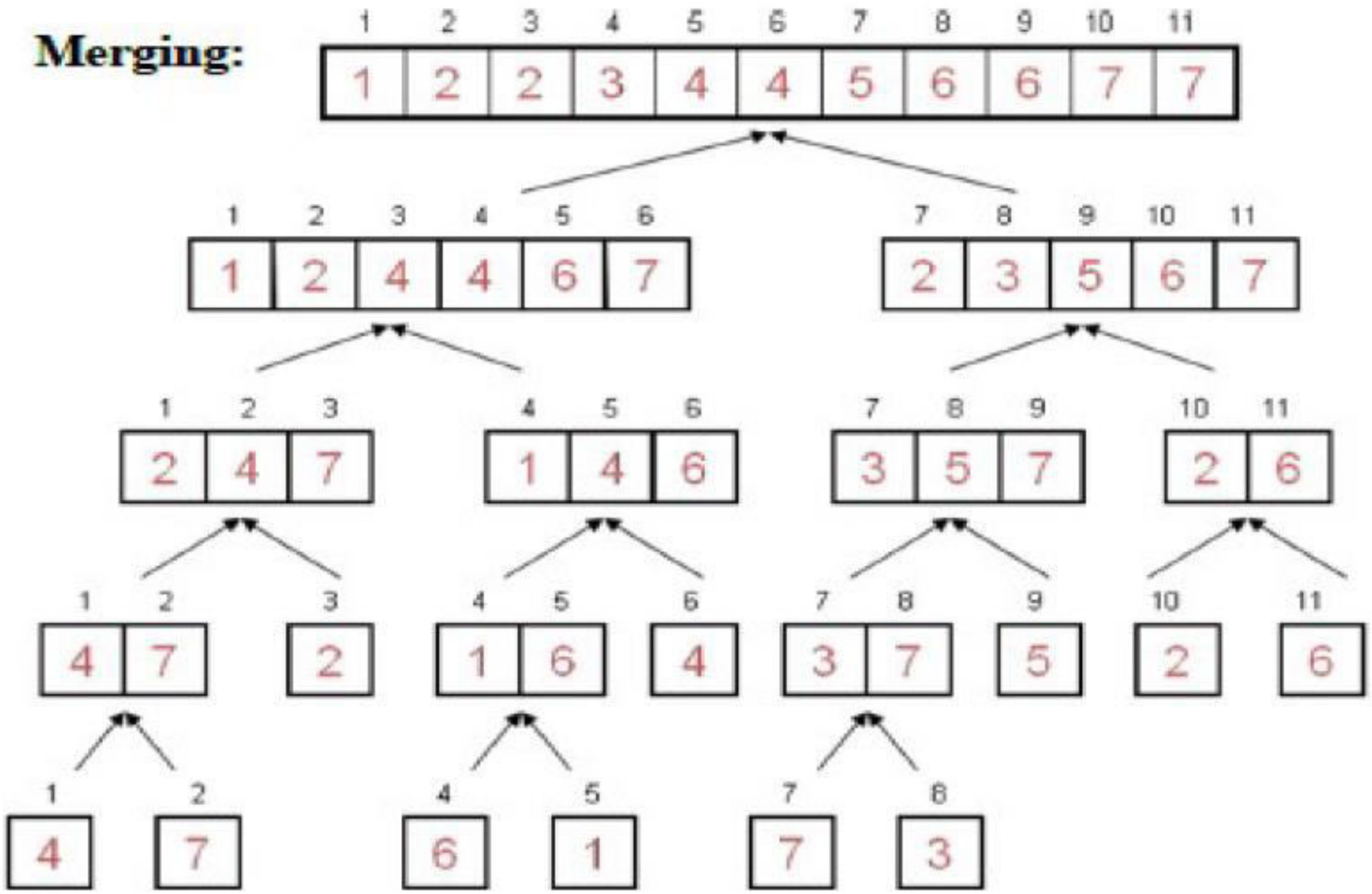
  Merge the two sorted sub-sequences

  Example: a[]={4, 7, 2, 6, 1, 4, 7, 3, 5, 2, 6}

Divide


1 2 3 4 5 6 7 8 9 10 11
4 7 2 6 1 4 7 3 5 2 6
1 2 3 4 5 6
4 7 2 6 1 4
7 8 9 10 11
7 3 5 2 6
1 2 3
4 7 2
4 5 6
6 1 4
7 8 9
7 3 5
10 11
2 6
1 2
4 7
3
2
4 5
6 1
6
4
7 8
7 3
9
5
10
2
11
6
1
4
2
7
4
6
5
1
7
7
8
3

Merging:

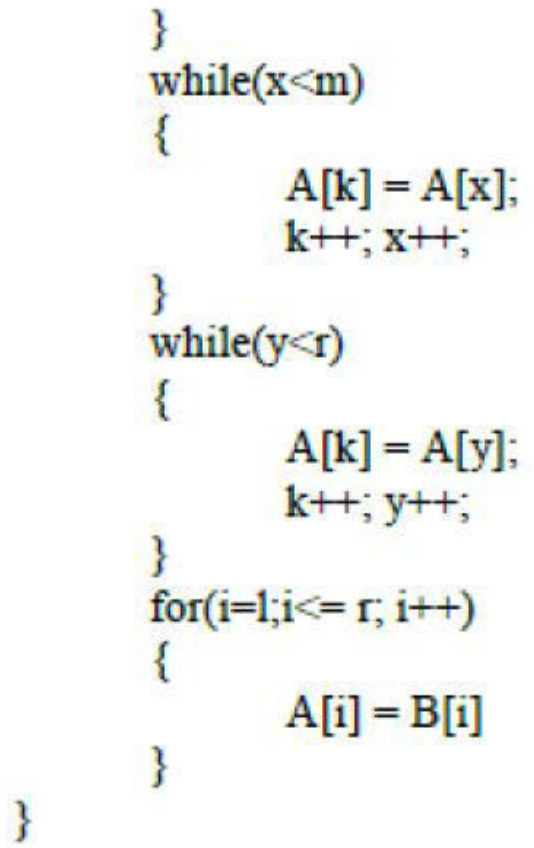


1 2 3 4 5 6 7 8 9 10 11
1 2 2 3 4 4 5 6 6 7 7
1 2 3 4 5 6
1 2 4 4 6 7
7 8 9 10 11
2 3 5 6 7
1 2 3
2 4 7
4 5 6
1 4 6
7 8 9
3 5 7
10 11
2 6
1 2
4 7
3
2
4 5
1 6
6
4
7 8
3 7
9
5
10
2
11
6
1
4
2
7
4
6
5
1
7
7
8
3

Algorithm:

```
MergeSort(A, 1, r)
{
        If (1 < r)
        {                                           //Check for base case
          m = $\lfloor(1 + r)/2\rfloor$                         //Divide
              MergeSort(A, 1, m)                    //Conquer
              MergeSort(A, m + 1, r)                //Conquer
              Merge(A, 1, m+1, r)                   //Combine
        }
}
```

```
Merge(A,B,l,m,r)
{
        x=l, y=m;
        k=l;
        while(x<m && y<r)
        {
                if(A[x] < A[y])
                {
                        B[k]= A[x];
                        k++; x++;
                }
                else
                {
                        B[k] = A[y];
                        k++; y++;
                }
```



```
        }
        while(x<m)
        {
                A[k] = A[x];
                k++; x++;
        }
        while(y<r)
        {
                A[k] = A[y];
                k++; y++;
        }
        for(i=l;i<= r; i++)
        {
                A[i] = B[i]
        }
}
```

Time Complexity:

Recurrence Relation for Merge sort:

$T(n) = 1$ if $n=1$

$T(n) = 2\,T(n/2) + O(n)$ if $n>1$

Solving this recurrence we get

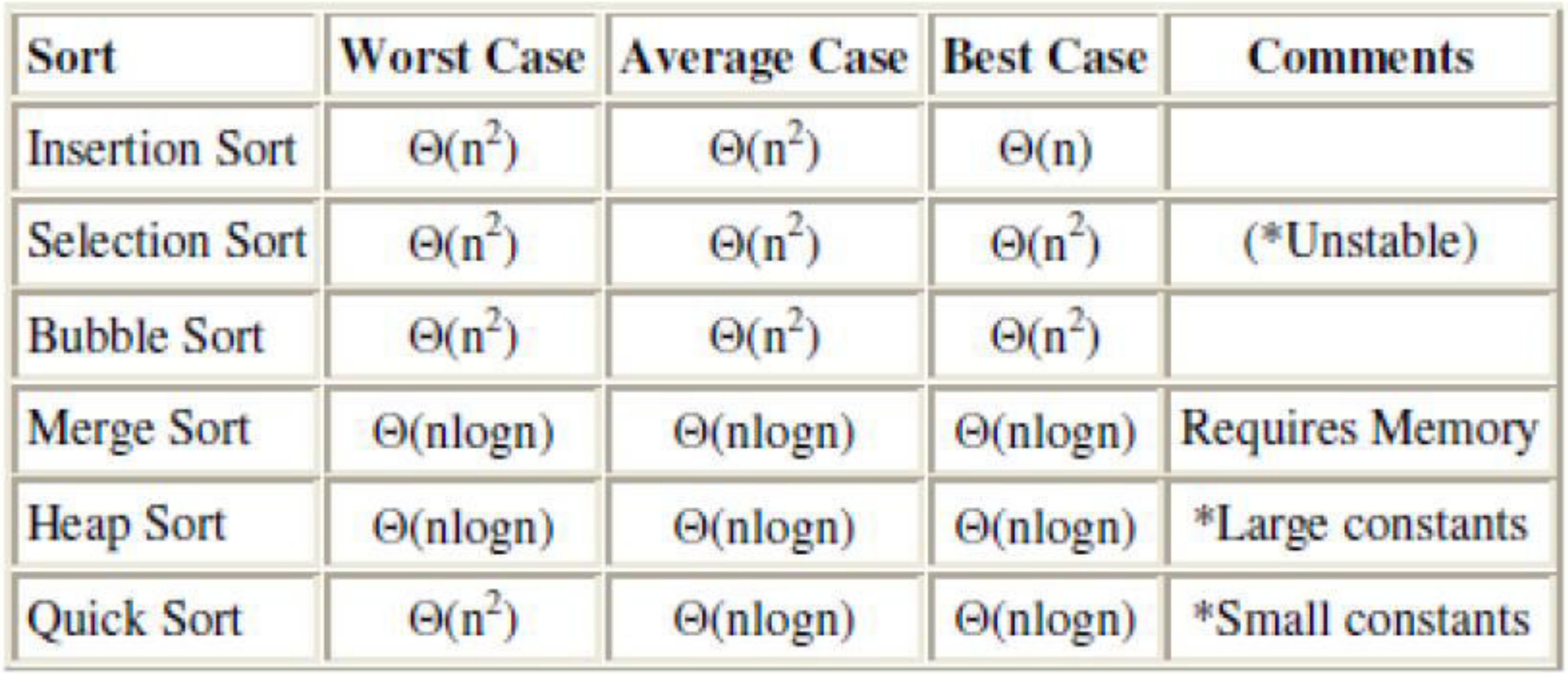
$T(n) = O(n\log n)$

Space Complexity:

It uses one extra array and some extra variables during sorting, therefore

Space Complexity$= 2n + c = O(n)$

Sorting Comparison:



Sort	Worst Case	Average Case	Best Case	Comments
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	(*Unstable)
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	
Merge Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	Requires Memory
Heap Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	*Large constants
Quick Sort	$\Theta(n^2)$	$\Theta(n\log n)$	$\Theta(n\log n)$	*Small constants

Thanks You

Any Queries?