Sorting

Sorting is the process of arranging the data either in ascending or descending order. The term **sorting** came into picture, as humans realized the importance of searching quickly.
Sorting refers to various methods of arranging or ordering things based on criteria (numerical, chronological, alphabetical, hierarchical, etc)
In internal sorting all the data to sort is stored in memory at all times while sorting is in progress. In external sorting data is stored outside memory (like on disk) and only loaded into memory in small chunks. External sorting is usually applied in cases when data can't fit into memory entirely.
An **internal sort** is any data sorting process that takes place entirely within the main memory of a computer. This is possible whenever the data to be sorted is small enough to all be held in the main memory.
External sorting is a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory,
Bubble sort

In bubble sort, we make several passes of the file, which is to be sorted.

On each pass, we compare ith element with (i+1)th element and then exchange them if they are out of order.

Here we bubble the highest value among the unsorted elements to the end using pairwise comparison and swapping

Algorithm

```
start
Step 1: Repeat Step 2 For i = 0 to N-1
Step 2: Repeat For J = 0 to N – i-1
Step 3: IF A[J] > A[J+1] SWAP (A[J] and A[i])
 [END OF INNER LOOP]
[END OF OUTER LOOP]
Step 4: End
```

C module

```
void bubbleSort(int arr[], int n)
{
for(i=0;i<n;i++)
        {
        for(j=0;j<n-i-1;j++)
                {
                if(arr[j]<arr[j+1])
                        {
                        temp=arr[j];
                        arr[j]=arr[j+1];
                        arr[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
=n(n+1)/2
= $O(n^2)$
Space Complexity:
The **space complexity** for **Bubble Sort** is O(1), because only a single additional memory **space** is required i.e. for temp variable for swapping.

Selection sort:

- ✔ Each pass selects the smallest data element from the unsorted set and move it to its position
- ✔ It works as follows:

  First find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.

Algorithm:

- ✔ start
- ✔ Search the smallest element in the array, and replace it with the element in the first position.
- ✔ We then move on to the second position, and look for smallest element present in the sub array, starting from index 1, till the last index.
- ✔ Replace the element at the **second** position in the original array, with the second smallest element.
- ✔ Repeat Until the array is completely sorted
- ✔ End

C module
```
void selectionSort(int arr[], int n)
{
        for( i =0;i< n;i++)
                {
                p=i;
                for ( j=i+1;j<n;j++)
                        {
                        if (arr[j]< arr[p])
                                {
                                p=j;
                                }
                        }
        temp=arr[i];
        arr[i]=arr[p];
        arr[p]=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
=n(n+1)/2
= O(n2)
Space Complexity:
The **space complexity** for **Selection Sort** is O(1), because only a single additional memory **space** is required

Insertion Sort:

- ✔ Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.
- ✔ For insertion, we compare A[j] with the elements of A[0,1,2,..j-1] and during that time, we move contents to the right during the search.

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.

C module
```
void selectionSort(int arr[], int n)
        {
        for(j=1;j<n;j++)
                {
                key=arr[j];
                i=j-1;
                while((i>-1) && (arr[i]>key))
                        {
                        arr[i+1]=arr[i];
                        i--;
                        }
                        arr[i+1]=key;
                }
        }
```

Shell sort
The basic idea in shellsort is to exchange every hth element in the array. Now this can be confusing so we'll talk more about this, *h(gap)* determines how far apart element exchange can happen, say for example take *h* as 13, the first element (index-0) is exchanged with the 14*th* element(index-13) if necessary (of course). The second element with the 15*th* element, and so on. Now if we take has 1, it is exactly the same as a regular insertion sort.
Shell sort is the generalization of insertion sort which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

Example:
23,29,15,19,31,7,9,5,2

Gap=n/2 in first pass; n/4 in second and n/8 in third pass
First pass:(gap=4)

23	29	15	19	**31**	7	9	5	2
23	**29**	15	19	31	**7**	9	5	2
23	7	**15**	19	31	29	**9**	5	2
23	7	9	**19**	31	29	15	**5**	2
23	7	9	5	**31**	29	15	19	**2**
23	7	9	5	**2**	29	15	19	31
2	7	9	5	23	29	15	19	31

Second pass(gap=2)

2	7	**9**	5	23	29	15	19	31
2	**7**	9	**5**	23	29	15	19	31
2	5	**9**	7	**23**	29	15	19	31
2	5	9	**7**	23	**29**	15	19	31
2	5	9	7	**23**	29	**15**	19	31
2	5	9	7	15	**29**	23	**19**	31
2	5	9	7	15	19	**23**	29	**31**
2	5	9	7	15	19	23	29	31

Third Pass:(gap=1)

2	**5**	9	7	15	19	23	29	31
2	**5**	**9**	7	15	19	23	29	31
2	5	**9**	**7**	15	19	23	29	31
2	5	7	**9**	**15**	19	23	29	31
2	5	7	9	**15**	**19**	23	29	31
2	5	7	9	15	**19**	**23**	29	31
2	5	7	9	15	19	**23**	**29**	31
2	5	7	9	15	19	23	**29**	**31**

C module:

```
void shellsort(int A[],n)
{
int gap;
for(gap=n/2;gap>=1;gap/2)
        {
        for(j=gap;j<n;j++)
                {
                for(i=j-gap;i>=0;i-gap)
                        {
                        if(A[i+gap]<A[i])
                                {
                                swap(A[i+gap],A[i]);
                                }
                        else
                                {
                                break;
                                }
                        }
                }
        }
}
```

Complexity

Worst case time complexity: **O(N (log N)^2)** comparisons
Average case time complexity:)O**(N (log N)^2)** comparisons
Best case time complexity: **O(N log N)**
Space complexity: **O(1)**.

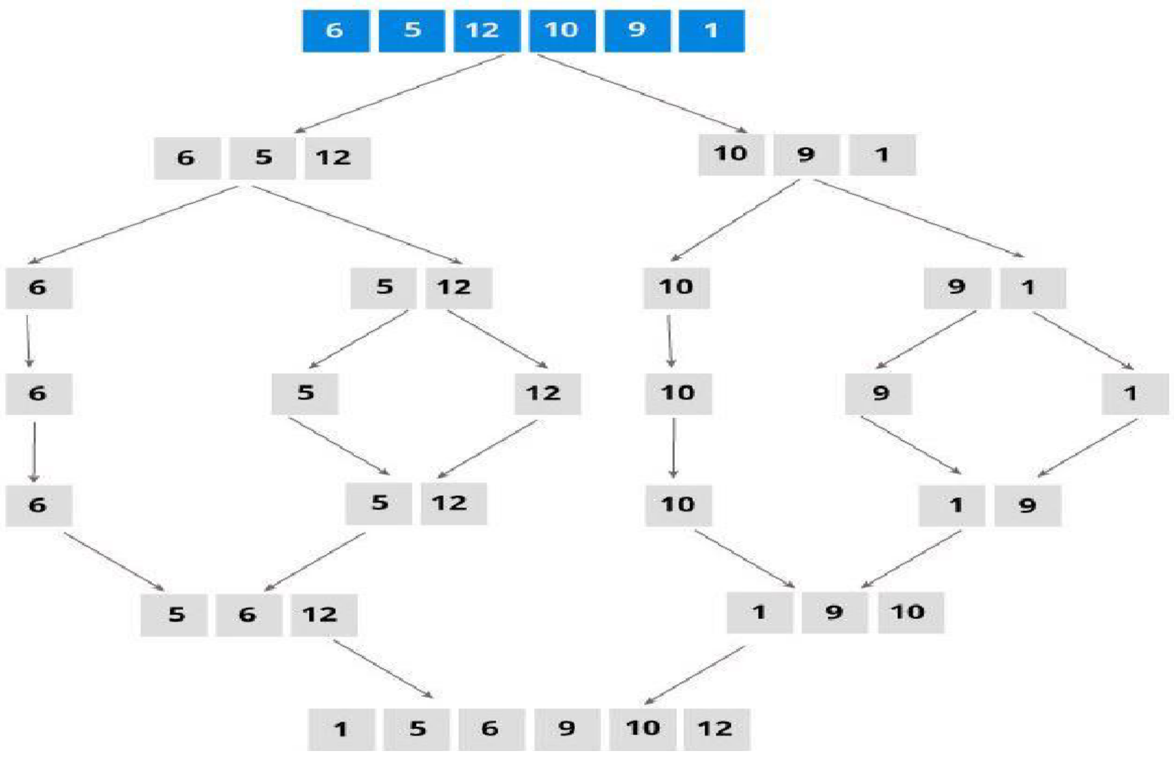
Concept of Divide and Conquer algorithm:

- ✔ **Divide and conquer** is an algorithm design paradigm based on multi-branched recursion. A **divide-and-conquer** algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.
- ✔ Both merge **sort** and quick sort employ a common **algorithmic** paradigm based on recursion. This paradigm, **divide-and-conquer**, breaks a problem into sub problems that are similar to the original problem, recursively solves the sub problems, and finally combines the solutions to the sub problems to solve the original problem.

The concept of Divide and Conquer involves three steps:

1. **Divide** the problem into multiple small problems.
2. **Conquer** the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
3. **Combine** the solutions of the subproblems to find the solution of the actual problem.

Merge sort:
Merge Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements, recursively, hence consuming less time.
Algorithm
The Merge Sort function repeatedly divides the array into two halves until we reach a stage where we try to perform Merge Sort on a subarray of size 1 i.e. p == r.
After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.



C module

```
void mergeSort(int a[], int beg, int end)
{
   int mid;
   if(beg<end)
   {
      mid = (beg+end)/2;
      mergeSort(a,beg,mid);
      mergeSort(a,mid+1,end);
      merge(a,beg,mid,end);
   }
}
void merge(int a[], int beg, int mid, int end)
{
   int i=beg,j=mid+1,k,index = beg;
   int temp[10];
   while(i<=mid && j<=end)
   {
      if(a[i]<a[j])
      {
         temp[index] = a[i];
         i = i+1;
      }
      else
      {
            temp[index] = a[j];
            j = j+1;
        }
        index++;
    }
    if(i>mid)
    {
        while(j<=end)
        {
            temp[index] = a[j];
            index++;
            j++;
        }
    }
    else
    {
        while(i<=mid)
        {
            temp[index] = a[i];
            index++;
            i++;
        }
    }
    k = beg;
    while(k<index)
    {
        a[k]=temp[k];
        k++;
    }
}
```

Performance Analysis:

Worst case complexity:	O(nlogn)
Best case complexity:	O(nlogn)
Average case complexity:	O(nlogn)
Space complexity	O(1)

Quick Sort:

Quick Sort is one of the most efficient sorting algorithms and is based on the splitting of an array into smaller ones

Quick Sort is also based on the concept of **Divide and Conquer**, just like merge sort. But in quick sort all the heavy lifting(major work) is done while **dividing** the array into subarrays, while in case of merge sort, all the real work happens during **merging** the subarrays.

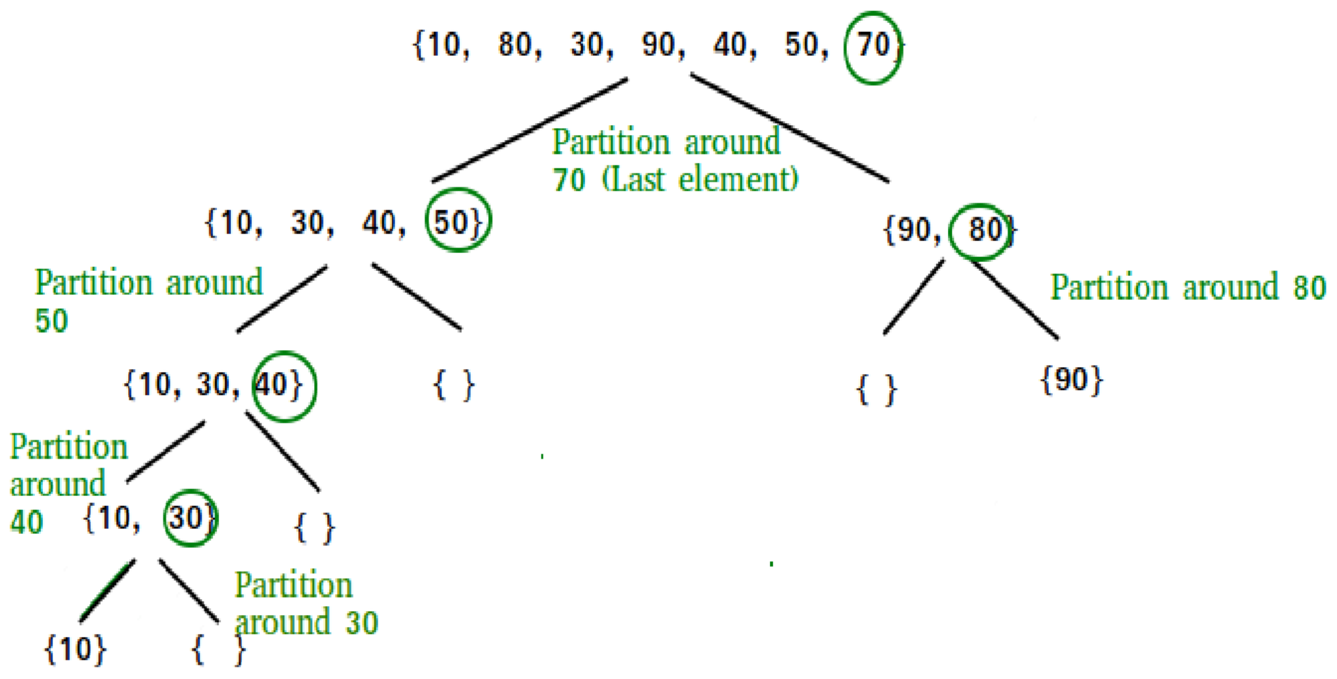
It is also called **partition-exchange sort**.

The recursive algorithm consists of four steps:

1) If there are one or no elements in the array to be sorted, return.

2) Pick an element in the array to serve as the "pivot" point. (Usually the left-most element in the array is used.)
3) Split the array into two parts – one with elements larger than the pivot and the other with elements smaller than the pivot.
4) Recursively repeat the algorithm for both halves of the original array.

Pivot element can be any element from the array, it can be the first element, the last element or any random element. Here we will take the rightmost element or the last element as **pivot**.



C module:

```
void Quicksort(int A[],int lb,int ub)            //A array name, lb:lower bound, ub:upper bound
int pivot;
if(lb>ub)
        {
pivot=partition(A,lb,ub);
Quicksort(A,lb,pivot-1);
Quicksort(A,pivot+1,ub);
}
}

int Partition(int A,int lb,in tub)
{
int start,end pivot_item=A[lb];
start=lb;
end=ub;
while(start<right)
        {
        while(A[start]<=pivot_item)
                {
                Start++;
                }
```

```
            while(A[end]>pivot_item)
                    {
                    end--;
                    }
            If(start<right)
                    {
                    swap(A[start],A[end])
                    }
    }
    swap(A[end],pivot_item);
    return end;
    }
```

Performance Analysis:

Worst case complexity:	O(n^2)
Best case complexity:	O(nlogn)
Average case complexity:	O(nlogn)
Space complexity	O(1)