1) Write a program to implement Quick Sort.

```
#include<stdio.h>
int partition(int arr[], int lb, int ub)
   {
      int pivot = arr[ub];
      int i = (lb - 1);
      for (int j = lb; j < ub; j++) {
         if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
         }
      }
         int temp = arr[i+1];
         arr[i+1] = arr[ub];
         arr[ub] = temp;
         return(i+1);
}
void QuickSort(int arr[], int lb, int ub) {
   if (lb < ub) {
      int pi = partition(arr, lb, ub);
      QuickSort(arr, lb, pi - 1);
      QuickSort(arr, pi + 1, ub);
      }
}
int main() {
   int n=5,lb,ub;
   int arr[] = {5,4,3,2,1};
```

```
    QuickSort(arr, 0, n-1);
    for(int i=0; i<5; i++) {
        printf("%d ", arr[i]);
    }
}
```

OUTPUT

1 2 3 4 5

2) Write a program to implement Merge Sort.

```
#include <stdio.h>
void conquer(int arr[], int si, int mid, int ei)
{
   int merge[ei-si+1];
   int idx1 = si;
   int idx2 = mid+1;
   int x=0;
   while(idx1 <= mid && idx2 <= ei) {
      if(arr[idx1] < arr[idx2]) {
         merge[x++] = arr[idx1++];
      } else {
          merge[x++] = arr[idx2++];
      }
   }
   while(idx1 <= mid) {
      merge[x++] = arr[idx1++];
   }
   while(idx2 <= ei) {
      merge[x++] = arr[idx2++];
   }
   for(idx1=si,x=0; x<ei-si+1; x++,idx1++ ) {
      arr[idx1] = merge[x];
   }
}
void devide(int arr[], int si, int ei)
{
   if (si < ei) {
      int mid = si + (ei-si)/2;
```

```
        devide(arr, si, mid);
        devide(arr, mid + 1, ei);
        conquer(arr, si, mid, ei);
    }
}
int main()
{
    int n=5;
    int arr[] = {5,4,3,2,1};
    devide(arr, 0, n-1);
    for(int i=0; i<5; i++) {
        printf("%d ", arr[i]);
    }
}
```

OUTPUT

1 2 3 4 5

3) Write a program to implement Bubble Sort.

```
#include<stdio.h>
void bubblesort(int arr[]) {
   int temp, n=5;
   for(int i=0; i<n-1; i++) {
      for(int 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;
         }
      }
   }
   printf("-----------------\nSorted array is:\n-----------------\n");
   for(int i=0; i<5; i++) {
   printf(" %d ", arr[i]);
   }
}
int main() {
   int arr[] = {6, 5, 4, 3, 2};
   bubblesort(arr);
}
```

OUTPUT

Sorted array is:

----

2 3 4 5 6

4) Write a program to implement selection sort.

```
#include<stdio.h>
void selection_sort(int arr[])
 {
   int n=5,temp, min;
   for(int i=0; i<n-1; i++)
      {
      min=i;
      for(int j=i+1; j<n; j++) {
         if(arr[min]>arr[j]) {
            min = j;
         }
         if(min != i)
        {
            temp = arr[i];
            arr[i] = arr[min];
            arr[min] = temp;
         }
      }
   }
   printf("-----------------\nSorted array is:\n-----------------\n");
   for(int i=0; i<n; i++) {
      printf("%d ", arr[i]);
   }
}
int main()
{
   int arr[] = {6, 5, 4, 3, 2};
   selection_sort(arr);
```

}

OUTPUT

Sorted array is:

----

2 3 4 5 6

5) Write a program to implement Binary Search.

```
#include<stdio.h>
int BinarySearch(int a[], int i, int j, int key) {
   while(i<=j) {
      int mid = (i+j)/2;
      if(a[mid]==key) {
         return mid;
      }
      if(key > a[mid]) {
        return BinarySearch(a, mid+1, j, key);
      }
      if(key < a[mid]) {
         return BinarySearch(a, i, mid-1, key);
      }
   }
   return -1;
}
void main() {
   int a[] = {2, 3, 4, 5, 6};
   int n = sizeof(a) / sizeof(a[0]);
   int key = 4;
   int result = BinarySearch(a, 0, n-1, key);
   printf("The Element is found at position: %d", result);
}
```

OUTPUT

The Element is found at position: 2

Question 1: Explain worst case, average case and best-case time complexity with proper example.

Worst Case Analysis:

- In the worst-case analysis, we calculate the upper limit of the execution time of an algorithm. It is necessary to know the case which causes the execution of the maximum number of operations.
- For linear search, the worst case occurs when the element to search for is not present in the array. When x is not present, the search () function compares it with all the elements of arr [] one by one. Therefore, the temporal complexity of the worst case of linear search would be Θ (n).

Average Case Analysis:

- In the average case analysis, we take all possible inputs and calculate the computation time for all inputs. Add up all the calculated values and divide the sum by the total number of entries.
- We need to predict the distribution of cases. For the linear search problem, assume that all cases are uniformly distributed. So, we add all the cases and divide the sum by (n + 1).

Best Case Analysis:

- In the best-case analysis, we calculate the lower bound of the execution time of an algorithm. It is necessary to know the case which causes the execution of the minimum number of operations. In the linear search problem, the best case occurs when x is present at the first location.
- The number of operations in the best case is constant. The best-case time complexity would therefore be Θ (1).
- Most of the time, we perform worst-case analysis to analyse algorithms. In the worst analysis, we guarantee an upper bound on the execution time of an algorithm which is good information.

Question 2: Why sorting algorithm are important?

Since sorting can often reduce the complexity of a problem, it is an important algorithm in Computer Science. These algorithms have direct applications in searching algorithms, database algorithms, divide and conquer methods, data structure algorithms, and many more.

Reason 1: we can learn various problem-solving approaches using sorting algorithms.

Reason 2: we can use sorting to solve several coding problems.

Reason 3: analysis of sorting algorithms is the best idea to learn analysis of algorithms.

Reason 4: sorting is a good problem to understand several variations of a coding problem, boundary conditions, and code optimization technique.

Question 3: Classify sorting algorithms based on 5 parameters.

Sorting algorithms can be categorized based on the following parameters:

1) **The number of swaps or inversions required:** This is the number of times the algorithm swaps elements to sort the input. Selection sort requires the minimum number of swaps.
2) **The number of comparisons:** This is the number of times the algorithm compares elements to sort the input.
3) **Whether or not they use recursion:** Some sorting algorithms, such as quick sort, use recursive techniques to sort the input. Other sorting algorithms, such as selection sort or insertion sort, use non-recursive techniques. Finally, some sorting algorithms, such as merge sort, make use of both recursive as well as non-recursive techniques to sort the input.
4) **Whether they are stable or unstable:** Stable sorting algorithms maintain the relative order of elements with equal values, or keys. Unstable sorting algorithms do not maintain the relative order of elements with equal values / keys.
5) **The amount of extra space required:** Some sorting algorithms can sort a list without creating an entirely new list. These are known as in-place sorting algorithms.

Question 4: Differentiate each and every sorting algorithm with real time applications.

- A real-world example of a bubble sort algorithm is how the contact list on your phone is sorted in alphabetical order. Or the sorting of files on your phone according to the time they were added.
- Quick Sort is used in operational research and event-driven simulation. Numerical computations and in scientific research, for accuracy in calculations most of the efficiently developed algorithm uses priority queue and quick sort is used for sorting.
- Selection sort : For example, picking apples from a tree. We can see all the apples on the tree, find and pluck the biggest one, and drop it into our basket (which, we can assume, always contains the apples in a sorted manner).
- Merge sort is clearly the ultimate easy example of this. In real life, we tend to break things up along useful lines. If we're sorting change, we first divide the coins up by denominations, then total up each denomination before adding them together.