**Algorithms and Problem-Solving Lab (15B17CI471)**
**Week -3**
**Topic: Divide and Conquer**

Q.1.Cubic integer root x of n is largest number x such that $x_3$<=n. Find the value of x given n using divide and conquer approach. Also analyse the complexity.

```
#include<iostream>
using namespace std;
void cubeRoot(int n, int left, int right, int &ans)
{
    if(left<=right)
    {
        int mid= (left+right)/2;
        int cube= mid*mid*mid;
        if(cube<=n)
        {
            ans= mid;
            cubeRoot(n, mid+1, right, ans);
        }
        else if(cube>n)
        {
            cubeRoot(n, left, mid-1, ans);
        }
    }
}
int main()
{
    for(int i=2; i<= 1000; i++)
    {
        int ans= -1;
        cout<<" i :"<<i<<" ";
        cubeRoot(i, 0, i-1, ans);
        cout<<ans<<endl;
    }
}
```

```
i :2 1
i :3 1
i :4 1
i :5 1
i :6 1
i :7 1
i :8 2
i :9 2
i :10 2
i :11 2
i :12 2
i :13 2
i :14 2
i :15 2
i :16 2
i :17 2
i :18 2
i :19 2
i :20 2
i :21 2
i :22 2
i :23 2
i :24 2
i :25 2
i :26 2
i :27 3
i :28 3
i :29 3
i :30 3
i :31 3
i :32 3
i :33 3
i :34 3
i :35 3
i :36 3
i :37 3
i :38 3
i :39 3
i :40 3
i :41 3
i :42 3
i :43 3
i :44 3
i :45 3
i :46 3
i :47 3
i :48 3
i :49 3
i :50 3
i :51 3
i :52 3
i :53 3
i :54 3
i :55 3
i :56 3
i :57 3
```

```
 i :975 9
 i :976 9
 i :977 9
 i :978 9
 i :979 9
 i :980 9
 i :981 9
 i :982 9
 i :983 9
 i :984 9
 i :985 9
 i :986 9
 i :987 9
 i :988 9
 i :989 9
 i :990 9
 i :991 9
 i :992 9
 i :993 9
 i :994 9
 i :995 9
 i :996 9
 i :997 9
 i :998 9
 i :999 9
 i :1000 10

Process returned 0 (0x0)   execution time : 0.257 s
Press any key to continue.
```

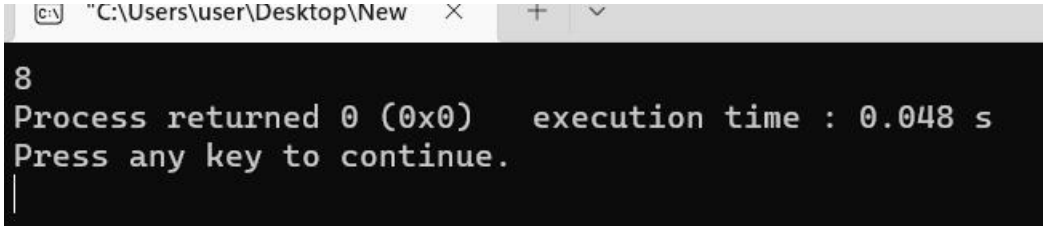
Q2. Given a sorted array in which all elements appear twice (one after one) and one element appears only once. Find that element in O(log n) complexity.
Example:
Input: arr[] = {1, 1, 3, 3, 4, 5, 5, 7, 7, 8, 8}
Output: 4
Input: arr[] = {1, 1, 3, 3, 4, 4, 5, 5, 7, 7, 8}
Output: 8

```
#include <bits/stdc++.h>
using namespace std;

int singleSearch(int *arr, int start, int end)
{
    if (start > end)
    {
        return -1;
    }
    int mid = (start + end) / 2;
    if (mid % 2 == 0)
    {
        if (arr[mid] == arr[mid + 1])
        {
            return singleSearch(arr, mid + 2, end);
        }
        else if (arr[mid] == arr[mid - 1])
        {
            return singleSearch(arr, start, mid - 2);
        }
        else
        {
            return arr[mid];
        }
    }
    else
    {
        if (arr[mid] == arr[mid - 1])
        {
            return singleSearch(arr, mid + 1, end);
        }
        else if (arr[mid] == arr[mid + 1])
        {
            return singleSearch(arr, start, mid - 1);
        }
```

```
        else
        {
            return arr[mid];
        }
    }
    return -1;
}

int main()
{
    int arr[] = {1, 1, 3, 3, 4, 4, 5, 5, 7, 7, 8};
    cout << singleSearch(arr, 0, sizeof(arr) / sizeof(int) - 1);
    return 0;
}
```



Q3. List of points have been given on 2D Plane. Calculate K closest points to the origin (0,0) (Consider euclidean distance to find the distance between two points). Write a code to return the answer in any order. The solution is guaranteed to be unique.

```
#include <bits/stdc++.h>
using namespace std;

struct point
{
    int x;
    int y;
    double dist;
};

double distance(int x, int y)
{
    return sqrt((float)(x * x + y * y));
}

int partition(point arr[], int low, int high)
{
    point pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j].dist < pivot.dist)
        {
            i++;
            point temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    point temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}

void quickSort(point arr[], int low, int high)
{
    if (low < high)
```

```
    {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main()
{
    int n, k;
    cin >> n >> k;
    point *arr = new point[n];
    for (int i = 0; i < n; i++)
    {
        int a, b;
        cin >> a >> b;
        arr[i].x = a;
        arr[i].y = b;
        arr[i].dist = distance(a, b);
    }
    quickSort(arr, 0, n - 1);
    for (int i = 0; i < k; i++)
    {
        cout << "[" << arr[i].x << ", " << arr[i].y << "]";
        if(i!=k-1){
            cout << ", ";
        }
    }
    return 0;
}
```

```
3 2
3 3 5 -1 -2 4
[3, 3], [-2, 4]
```

Q4. Let there be an array of N random elements. We need to sort this array in ascending order. If n is very large (i.e. N= 1,00,000) then Quicksort may be considered as the fastest algorithm to sort this array. However, we can further optimize its performance by hybridizing it with insertion sort. Therefore, if n is small (i.e. N<= 10) then we apply insertion sort to the array otherwise Quick Sort is applied. Implement the above discussed hybridized Quick Sort and compare the running time of normal Quick sort and hybridized quick sort. Run each type of sorting 10 times on a random set of inputs and compare the average time returned by these algorithms.

```
#include <bits/stdc++.h>
using namespace std;

void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
void hybridizedSort(int arr[], int n)
{
```

```
    if (n <= 10)
    {
        for (int i = 1; i < n; i++)
        {
            int current = arr[i];
            int j = i - 1;
            while (arr[j] > current && j >= 0)
            {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = current;
        }
    }
    else
    {
        quickSort(arr, 0, n - 1);
    }
}
int main()
{
    int n;
    cin >> n;
    int *arr = new int[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    hybridizedSort(arr, n);
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    delete[] arr;
    return 0;
}
```

```
10
8 1 2 6 71 12 18 19 20 13
1 2 6 8 12 13 18 19 20 71
```

Q5. Consider a sorted array A of n elements. The array A may have repetitive/duplicate elements. For a given target element T, design and implement an efficient algorithm to find T's first and last occurrence in the array A. Also print the message if an element was not present in the array.

```
#include <bits/stdc++.h>
using namespace std;

void binarySearch(int *arr, int n, int key){
    int start = 0;
    int end = n-1;
    bool flag = false;
    int mid;
    while(start<=end && !flag){
        mid = start + (end-start)/2;
        if(arr[mid]==key){
            flag = true;
        } else if(arr[mid]>key){
            end = mid -1;
        } else if(arr[mid]<key){
            start = mid+1;
        }
    }
    if(!flag){
        cout << "Element not found in the array\n";
        return;
    }
    int x = mid;
    while(arr[x] == key){
        x--;
    }
    cout << "The first occurrence of element " << key << " is located at index " <<
(x+1) << "\n";
    x = mid;
    while(arr[x] == key){
        x++;
    }
    cout << "The last occurrence of element " << key << " is located at index " <<
(x-1) << "\n";
}

int main()
{
```

```
    int arr[] = {2, 5, 5, 5, 6, 6, 8, 9, 9, 9};
    for (int i = 1; i < 11; i++)
    {
        binarySearch(arr, sizeof(arr)/sizeof(int), i);
    }
    return 0;
}
```

```
The last occurrence of element 2 is located at index 0
Element not found in the array
The last occurrence of element 2 is located at index 0
Element not found in the array
Element not found in the array
The first occurrence of element 5 is located at index 1
The last occurrence of element 5 is located at index 3
The first occurrence of element 6 is located at index 4
The last occurrence of element 6 is located at index 5
Element not found in the array
The first occurrence of element 8 is located at index 6
The last occurrence of element 8 is located at index 6
The first occurrence of element 9 is located at index 7
The last occurrence of element 9 is located at index 9
Element not found in the array
```