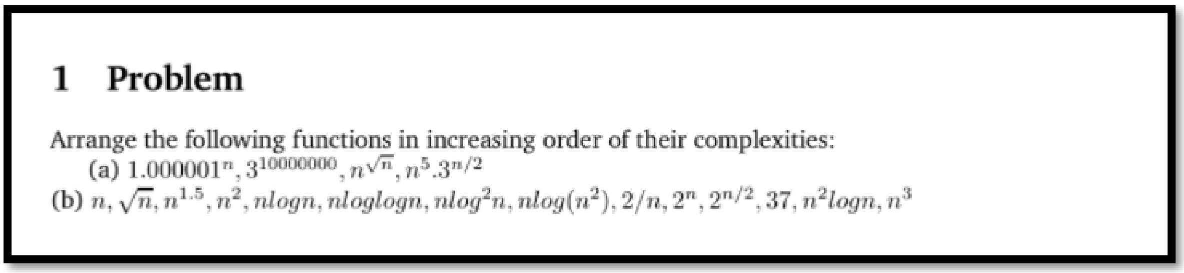
CS203: Collection of Problems

Name: Hritik Kumar

Roll no.: 202051088



1 Problem

Arrange the following functions in increasing order of their complexities:

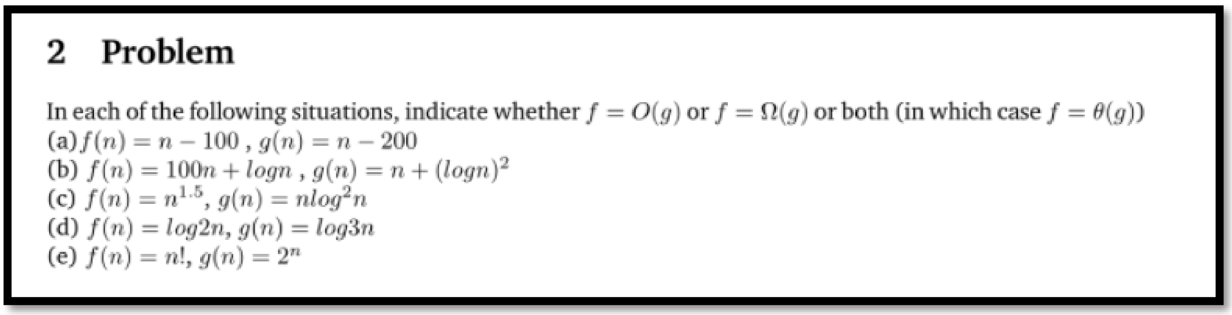
(a) $1.000001^n, 3^{10000000}, n^{\sqrt{n}}, n^5.3^{n/2}$

(b) $n, \sqrt{n}, n^{1.5}, n^2, nlogn, nloglogn, nlog^2n, nlog(n^2), 2/n, 2^n, 2^{n/2}, 37, n^2logn, n^3$

Solution 1

a) $3^{10000000} < n^5 < 1.000001^n < 3^{n/2} < n^{sqrt(n)}$

b) $2/n < 37 < sqrt(n) < n < nloglogn < nlogn < nlog(n^2)$
$< nlog^2n << n^{1.5} < n^2 < n^2logn < n^3 < 2^{n/2} < 2^n$



2 Problem

In each of the following situations, indicate whether $f = O(g)$ or $f = \Omega(g)$ or both (in which case $f = \theta(g)$)

(a) $f(n) = n - 100$, $g(n) = n - 200$

(b) $f(n) = 100n + logn$, $g(n) = n + (logn)^2$

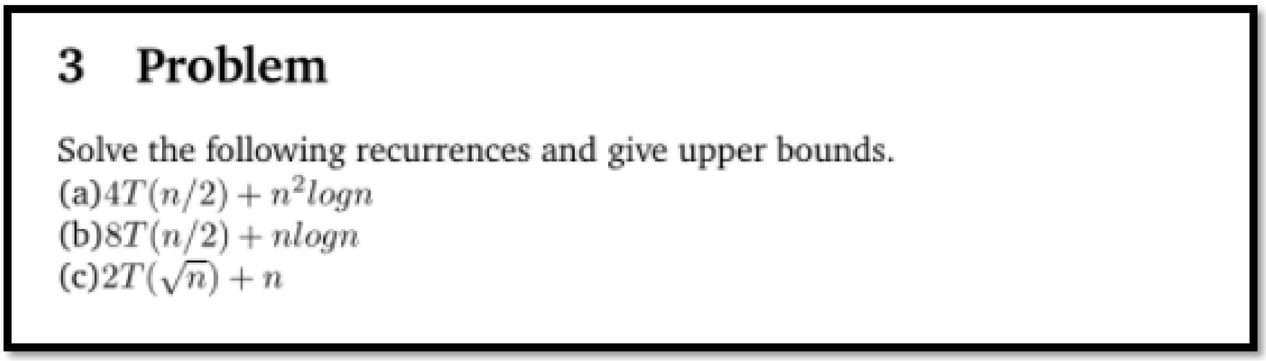
(c) $f(n) = n^{1.5}$, $g(n) = nlog^2n$

(d) $f(n) = log2n$, $g(n) = log3n$

(e) $f(n) = n!$, $g(n) = 2^n$

Solution 2

a. $f = \theta(g)$
b. $f = O(g)$
c. $f = \Omega(g)$
d. $f = \theta(g)$
e. $f = \Omega(g)$



3 Problem

Solve the following recurrences and give upper bounds.
(a)$4T(n/2) + n^2 logn$
(b)$8T(n/2) + nlogn$
(c)$2T(\sqrt{n}) + n$

Solution 3

We can solve the question using master theorem

A. 4T (n/2) + n^2logn

$T(n) = 4T(n/2) + n^2 logn$
By master theorem a = 4, b = 2, k =2
and p = 1. Now, logba = k = 2
And p> -1
Therefore TC = $O(n^2 log^2 n)$

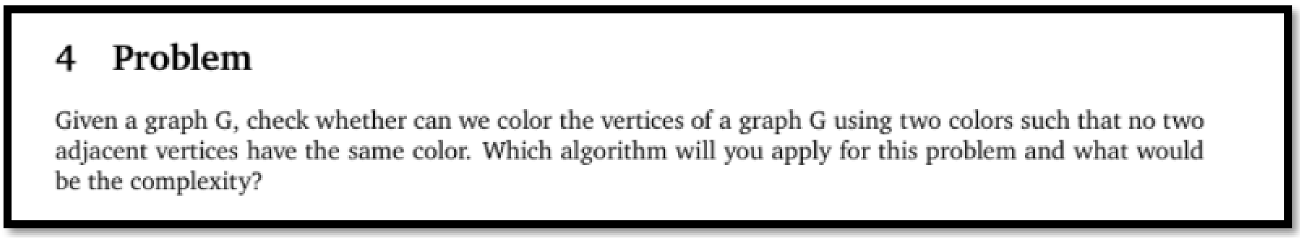
B. 8T (n/2) + nlogn

T(n) = 8T(n/2) + nlogn
By master theorem a = 8, b = 2, k = 1
and p =1 Now logba > k
Therefore $T_C = O(n^{logba}) = O(n^3)$

C. 2T (√n) + n

T(n) = 2T(√n) + n
Let $n = 2^m$ -> $n^{1/2} = 2^{m/2}$

- $T(2^m) = T(2^{m/2}) + 2^m$
- $S(m) = S(m/2) + m$
- $a = 1$, $b = 2$, $k = 1$ and $p = 0$
- By master theorem
- $T_C = O(n^k \log^0 n)$
- $T_C = O(n)$



4 Problem

Given a graph G, check whether can we color the vertices of a graph G using two colors such that no two adjacent vertices have the same color. Which algorithm will you apply for this problem and what would be the complexity?

Solution 4

To check if the graph can be colored using only two colors such that no two adjacent nodes have the same color, we can use the concept of Bipartite Graph.
Bipartite graph are the graphs which when colored we have to use only two colors and no two adjacent nodes have the same color or we can say that it is a graph which can be divided into two sets such that there are edges only from either u to v to v to u given that v and u are the two sets where we are dividing the nodes.

Let us understand the algorithm for it:

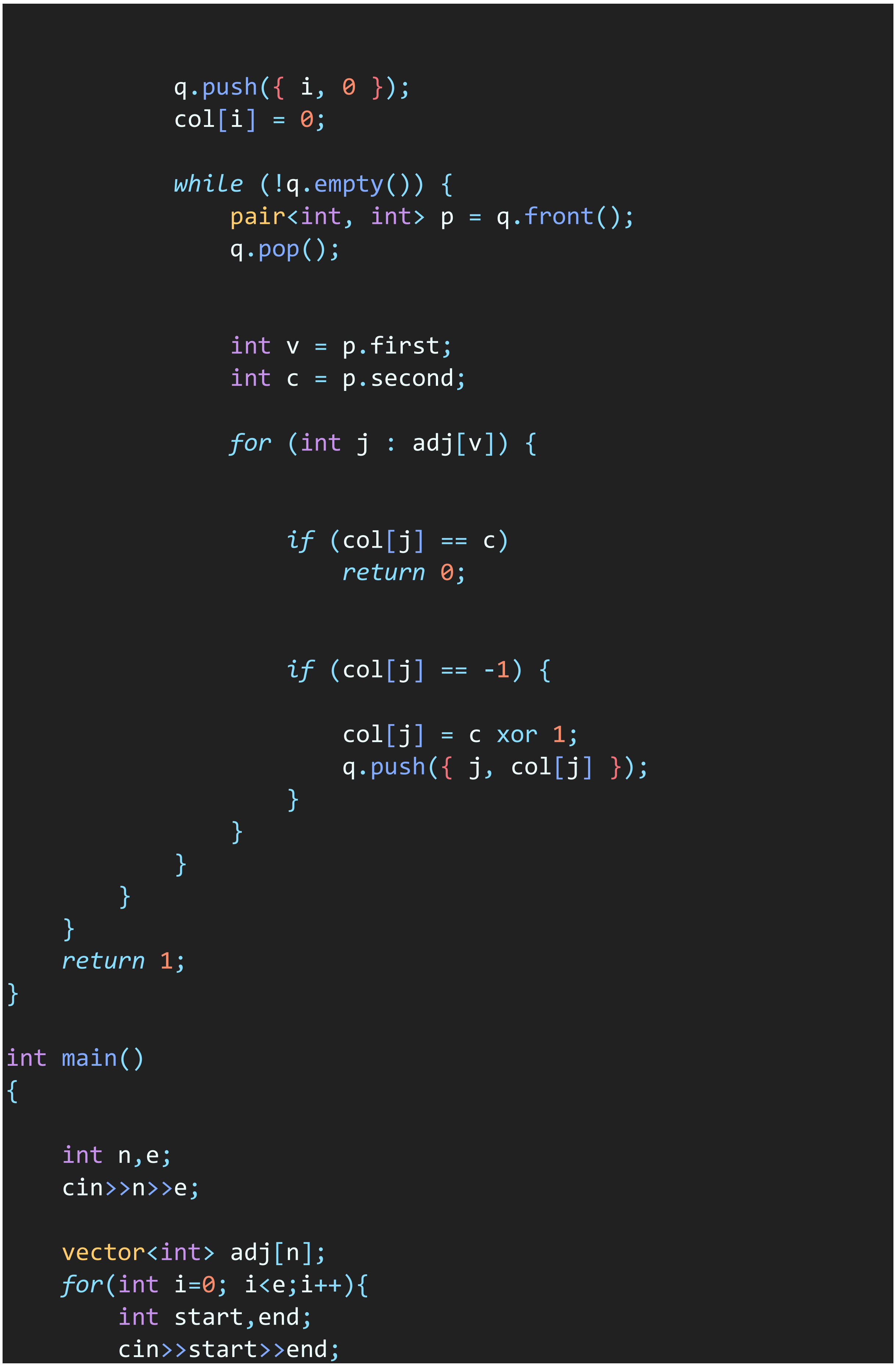
Algorithm:

1. Start

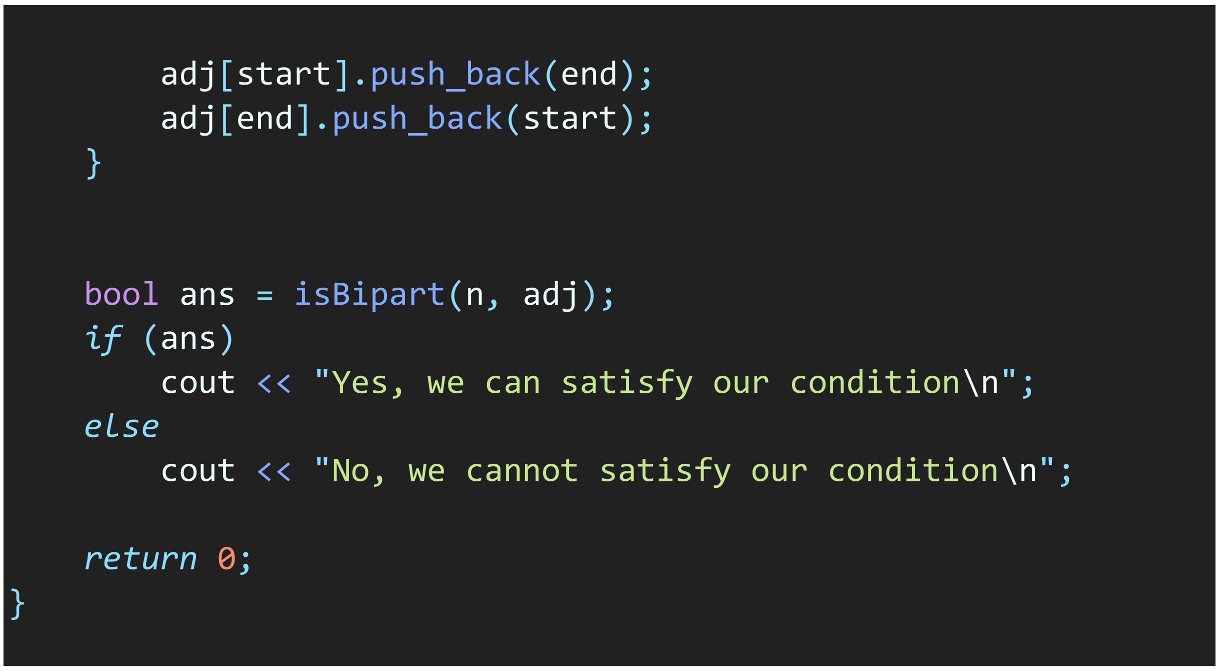
2. Pick up a node and color it with any one color, say red
3. Traverse to its adjacent vertices and check
    a. If they are not colored color them with a color other than red, i.e., blue
    b. If it is colored then check if the color is same as the color of parent, then we have to break and say no it is not possible, else we continue to the neighbors
4. Now if we go to next neighbors then we do the same above step.
5. If we traverse all the nodes then we finally return true and say that the graph is bipartite i.e., we can color the nodes with only two colors such that no two adjacent nodes have the same color.
6. End

Code:

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

bool isBipart(int V, vector<int> adj[])
{

    vector<int> col(V, -1);

    queue<pair<int, int> > q;


    for (int i = 0; i < V; i++) {


        if (col[i] == -1) {
```

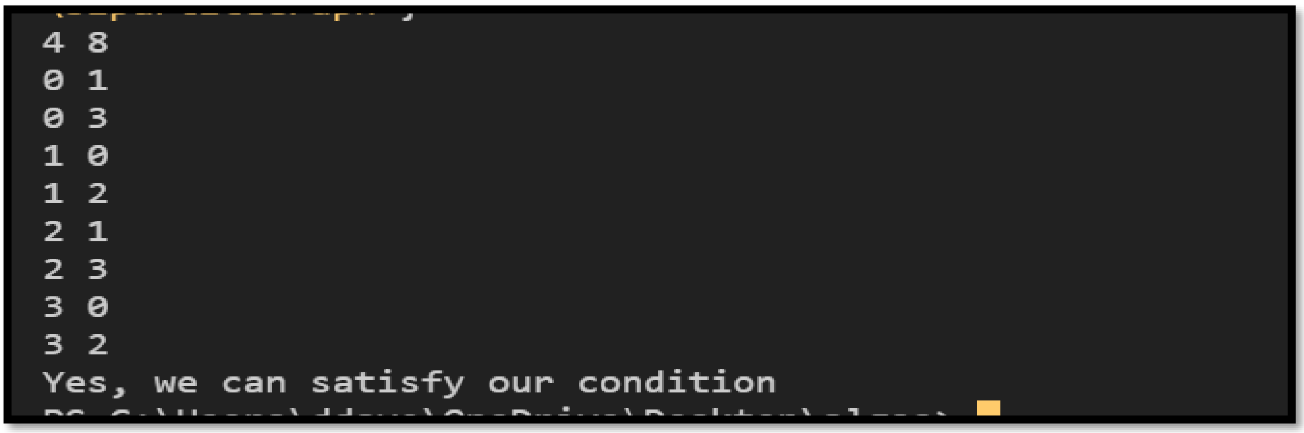


```
            q.push({ i, 0 });
            col[i] = 0;

            while (!q.empty()) {
                pair<int, int> p = q.front();
                q.pop();


                int v = p.first;
                int c = p.second;

                for (int j : adj[v]) {


                    if (col[j] == c)
                        return 0;


                    if (col[j] == -1) {

                        col[j] = c xor 1;
                        q.push({ j, col[j] });
                    }
                }
            }
        }
    }
    return 1;
}

int main()
{

    int n,e;
    cin>>n>>e;

    vector<int> adj[n];
    for(int i=0; i<e;i++){
        int start,end;
        cin>>start>>end;
```



```
        adj[start].push_back(end);
        adj[end].push_back(start);
    }


    bool ans = isBipart(n, adj);
    if (ans)
        cout << "Yes, we can satisfy our condition\n";
    else
        cout << "No, we cannot satisfy our condition\n";

    return 0;
}
```

Output:

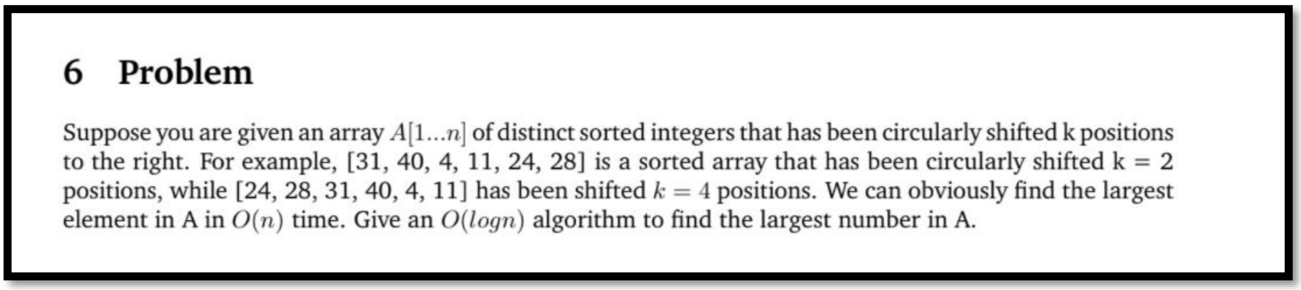


```
4 8
0 1
0 3
1 0
1 2
2 1
2 3
3 0
3 2
Yes, we can satisfy our condition
```

Time Complexity:

In this algorithm to traverse to the nodes we are simply following BFS (Breadth First Search)
So, the time complexity goes to O(V+E)

Hence the complexity for this algo where we use adjacency list and BFS, the time complexity goes to O(V+E).



6 Problem

Suppose you are given an array $A[1...n]$ of distinct sorted integers that has been circularly shifted k positions to the right. For example, [31, 40, 4, 11, 24, 28] is a sorted array that has been circularly shifted k = 2 positions, while [24, 28, 31, 40, 4, 11] has been shifted $k = 4$ positions. We can obviously find the largest element in A in $O(n)$ time. Give an $O(logn)$ algorithm to find the largest number in A.

Solution 5

Here we are given a rotated sorted array and we are given the key that how many times we have rotated the sorted array.
We can find the maximum in O(N) simply by iterating once in the array but we want an algorithm that can do this in O(logN).

So, we can use the binary search only that works on sorted array so we can use a technique to find the pivot in the array and then apply binary search in it.
In a rotated sorted array pivot element will be the element whose next element is smaller than the previous.
And this pivot element will be the largest element in the array.

Algorithm:

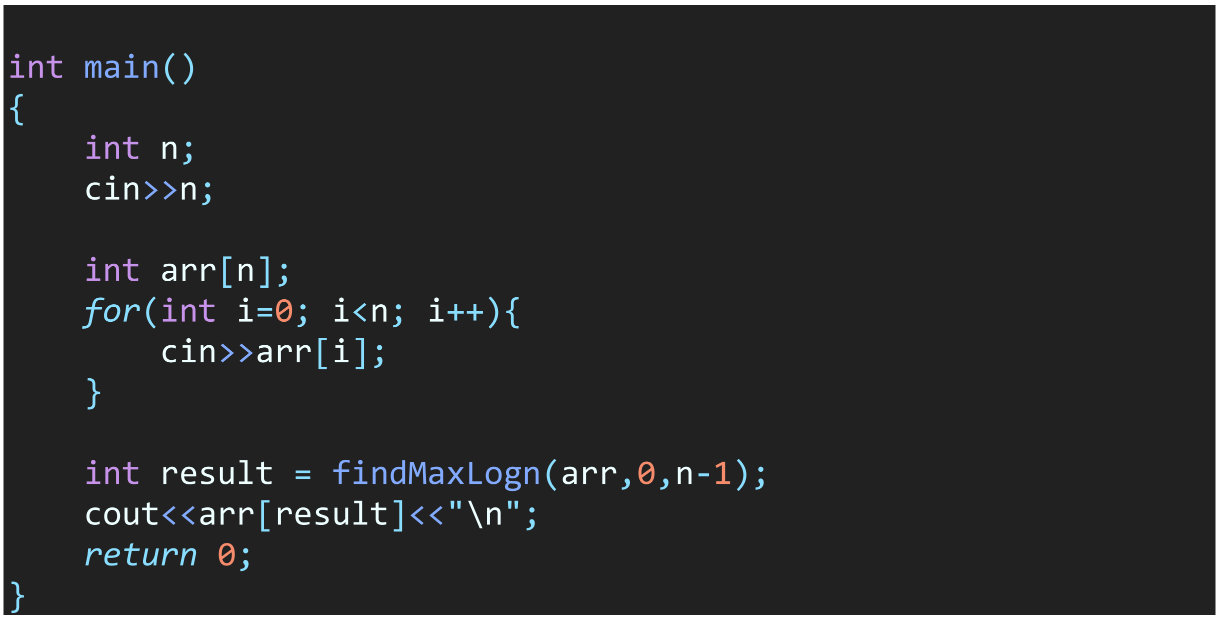
1. Start
2. Base case will be that if the (high < low) i.e., no

element or if high == low i.e., only 1 element then we simply just return -1 in the prior case and low in the other case

3. Now we find the mid of the array using (low+high)/2
    a. If mid < high and we get arr[mid] > arr[mid+1] then we return mid, as that's the position
    b. If mid > low and we get arr[mid] < arr[mid-1] then we return mid-1.
4. If arr[low] >= arr[mid] then we apply the same on the subarray low to mid
5. Else apply the same on the subarray mid+1 to high.
6. End

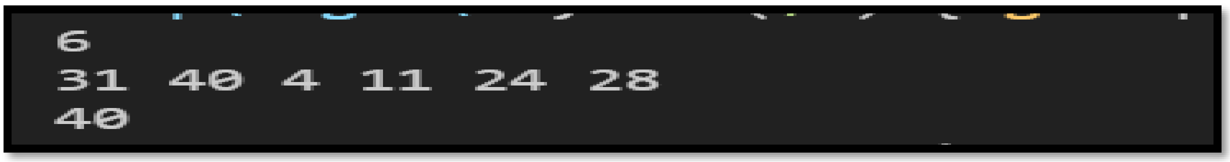
Code:

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

int findMaxLogn(int arr[], int low, int high)
{

    if (high < low)
        return -1;
    if (high == low)
        return low;

    int mid = (low + high) / 2;
    if (mid < high && arr[mid] > arr[mid + 1])
        return mid;

    if (mid > low && arr[mid] < arr[mid - 1])
        return (mid - 1);

    if (arr[low] >= arr[mid])
        return findMaxLogn(arr, low, mid - 1);

    return findMaxLogn(arr, mid + 1, high);
}
```

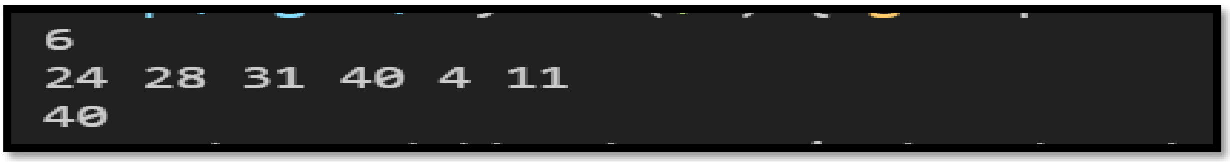


```
int main()
{
    int n;
    cin>>n;

    int arr[n];
    for(int i=0; i<n; i++){
        cin>>arr[i];
    }

    int result = findMaxLogn(arr,0,n-1);
    cout<<arr[result]<<"\n";
    return 0;
}
```

Output:



```
6
31 40 4 11 24 28
40
```



```
6
24 28 31 40 4 11
40
```

Time Complexity:

Here we are searching the pivot element i.e., the one which is going to be the maximum after rotating the sorted array using “binary search” technique in which we move in the array in gaps of half so the total traversals are of logN

Hence the following algorithm will get us the maximum element in time complexity of O(logN).