Competitive Programming For Beginners

Agenda

- Coding Problems
  - Approaches
  - Time Complexity
  - Space Complexity

QUESTION

Find the element that is present once in an array where every other element is present twice?

Input:[1,3,5,6,6,3,1]
Output: 5

Input:[10,30,50,60,60,30,10]
Output: 50

First Approach

Using NESTED LOOPS: Time Complexity: O(n*n)

```
for(int i=0;i<arr.length;i++) {
                for(int j=0;j<arr.length;j++) {
                        if(i != j && arr[i]==arr[j])
                            flag=1;
                  }
                   if(flag==0)
                     {
                        System.out.println(arr[i]);
                        break;
                      }
                  flag=0;
```

Second Approach

Using SORTING: Time Complexity: O(n*log n)

```
Arrays.sort(arr);
        for(int i=0;i<arr.length;i=i+2)
            {
                if(arr[i]!=arr[i+1])
                  {
                    System.out.println(arr[i]);
                    break;
                  }
            }
```

Third Approach

Using HASHMAP: Time Complexity: O(n) Space Complexity: O(n)

```
HashMap<Integer,Integer> map = new HashMap<Integer,Integer>();
        for(int i=0;i<arr.length;i++) {
                if(map.containsKey(arr[i]))
                    map.put(arr[i],map.get(arr[i])+1);
                 else
                   map.put(arr[i],1);
            }
            System.out.println(map);  //FOR DEBUGGING
              for(int x :map.keySet()) {
               if(map.get(x)==1){
                       System.out.println(x);
                          break;
                    }
```

Fourth Approach

Using for loop: Time Complexity: O(n) Space Complexity: O(1)

```
int res=0;
        for(int i=0;i<arr.length;i++) {
            {
                res= res^ arr[i];

            }
         System.out.println(res);
```

Bitwise XOR

INPUT X	INPUT Y	OUTPUT
0	0	0
0	1	1
1	0	1
1	1	0

So, 8 ^8 = 0 And 8^0 = 8
That is what we used: 1^1 ^ 3^3 ^ 5 ^ 6^6 = 5^0 = 5

What if the elements in array are sorted ?

Input:[1,1,3,3,5,6,6]
Output: 5

Using LOOPS: Time Complexity: O(n)

```
for(int i=0;i<arr.length;i=i+2)
    {
            if(arr[i]!=arr[i+1])
              {
                 System.out.println(arr[i]);
                 break;
              }
    }
```

Second Approach

Using Two Pointer: Time Complexity: O(n) Space Complexity: O(1)

```
int left_pointer = 0;
        int right_pointer = arr.length - 1;
        while (left_pointer < right_pointer) {
              if (left_pointer < right_pointer && arr[left_pointer] != arr[left_pointer + 1])
                        return arr[left_pointer];
              if (left_pointer < right_pointer && arr[right_pointer] != arr[right_pointer - 1])
                        return arr[right_pointer];

                    left_pointer += 2;                    //1,1,3,3,5,6,6
                    right_pointer -= 2;
               }
               return arr[left_pointer];
```

Third Approach

Using BinarySearch: Time Complexity: O(log n) Space Complexity: O(1)

Input array: 1,1,3,3,5,6,6]
0 1 2 3 4 5 6

First occurrence: Even indexes (0,2,4,..)
Second occurrence: Odd indexes (1,3,5,..)

But that element (5)

First occurrence: Odd indexes (1,3,5,7,...)
Second occurrence: Even indexes (0,2,4,6...)

```
int low = 0;
int  high = arr.length -1;
while(low < high){
    int mid = (low +high)/2;
     if(low==high)
     {
          System.out.println(arr[low]);
          break;
     }
    if( ( (mid & 1) == 0) && arr[mid] == arr[mid+1])         //1,1,2,2,3,3,5,6,6
        low = mid+2;
    else if( (mid & 1) == 0)
        high = mid;
    else if( ( (mid & 1) == 1) && arr[mid] == arr[mid-1])    //1,2,2,3,3,4,4,5,5
        low = mid+1;
    else if( (mid &1) == 1)
        high = mid-1;
}
```

QUESTION

Find the maximum sum subarray of size k?

Input: [2, 1, 5, 1, 3, 2], k=3

Output: 9 //Subarray with maximum sum is [5, 1, 3].

Sliding Window

2 1 5 1 3 2

2 1 5 1 3 2

2 1 5 1 3 2

2 1 5 1 3 2

2 1 5 1 3 2

2 1 5 1 3 2

Sliding Window

Time Complexity: O(n) Space Complexity: O(1)

```
public static int function(int k, int[] arr) {
    int sum = 0, maxSum = 0;
    int windowStart = 0;
    for (int windowEnd = 0; windowEnd < arr.length; windowEnd++) {
      sum += arr[windowEnd];
      if (windowEnd >= k - 1) {
        maxSum = Math.max(maxSum, sum);
        sum -= arr[windowStart];
        windowStart++;
      }
    }
    return maxSum;
  }
```

QUESTION

Find the maximum sum subarray ?

Input: [-2, 1, 5, 1, 3,-2]

Output: 10 //Subarray with maximum sum is [1, 5, 1, 3].

Kadane's Algorithm

	Local	Global
-2 1 5 1 -3 4	-2	Int.MinVal
	-2>global	-2
	0	
-2 1 5 1 -3 4	1	
	1>global	1
-2 1 5 1 -3 4	6	
	6>1	6
-2 1 5 1 -3 4	7>6	
	7	7
-2 1 5 1 -3 4	4>7	
-2 1 5 1 -3 4	8>7	
	8	8

Kadane's Algorithm

Time Complexity: O(n) Space Complexity: O(1)

```
static int function(int arr[])
    {
        int global = Integer.MIN_VALUE, local = 0;
        for (int i = 0; i < arr.length; i++)
        {
             local = local + arr[i];
            if (global < local)
                 global = local;
            if (local < 0)
                local = 0;
        }
        return global;
    }
```

Fibonacci Series

Series: 0 1 1 2 3 5 8....

What is its recurrence relation?

Fibonacci Series

```
static int fibS(int num)
 {
     if (num <= 1)
        return num;

    return fibS(num-1) + fibS(num-2);
 }
```

No of binary string without consecutive 1's

Input: N = 2
Output: 3   //3 strings are 00, 01, 10

Input: N = 3
Output: 5 //5 strings are 000, 001, 010, 100, 101

No of binary string without consecutive 1's

0	1	1	2	3	5	8
			n=1	n=2	n=3	n=4

```
static int function(int n)
   {
        if(n==0||n==1)
             return n;
        return function(n-1)+function(n-2);
   }
   public static void main (String args[])
   {
        int n=3;
        System.out.println(function(n+2));
   }
```

No of binary string without consecutive 1's

```
static int function(int n)
{
        if(n==0)
                return 0;
         if(n==1)
                return 2;
          if(n==2)
                 return 3;
        return function(n-1)+function(n-2);
}
public static void main (String args[])
{
        int n=4;
        System.out.println(function(n));
}
```

Count no of ways to reach the top. You can take either 1 or 2 step at a time

Input: n = 1
Output: 1 //only 1 stair to climb

Input: n = 2
Output: 2 // (1, 1) and (2)

Input: n = 4
Output: 5 //(1, 1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 2)

Count no of ways to reach the top. You can take either 1 or 2 step at a time

```
static int fib(int n){
        if(n==0||n==1)
                return n;
        return fib(n-1)+fib(n-2);
 }
 public static void main (String args[]){
        int n=4;
        System.out.println(fib(n+1));
 }
```

Catalan Series

Series: 1 1 2 5 14 42 132……..

1. c(0) = c(1)= 1
2. c(2) = 1 1 / 1 1 = 1+1 = 2
3. c(3) = 1 1 / 1 1 / 2 2 = 2+1+2= 5

What is its recurrence relation?

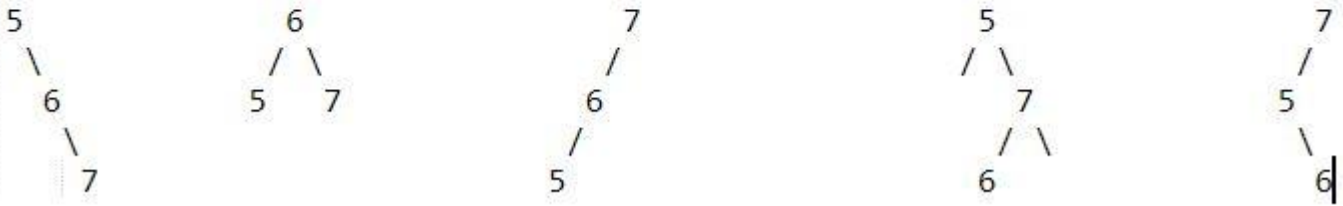
Catalan Series

```
static int catS(int num) {
        if (num <= 1)
                return 1;
        int res = 0;
        for (int i=0; i<n; i++)
                res += catS(i)*catS(n-i-1);
        return res;
}

 public static void main(String[] args) {
       System.out.print(catS(5));
}
```

Number of BST's possible from n nodes ?

2 Nodes (5,7)

3 Nodes(5,6,7)



QUESTION

Find the Next Largest element / Next Greater element to right

Input: [2, 1, 5, 1, 3, 2]

Output: [5,5,-1,3,-1,-1]

First Approach

Time Complexity: O(n*n) Space Complexity: O(1)

```
public static int function(int[] arr) {
        int out=0;
    for (int itr = 0; itr < arr.length; itr ++) {
                out = -1;
          for( int jtr=itr+1; jtr<arr.length; jtr++) {
                    if( arr[itr]<arr[jtr]
                    {
                          out = arr[jtr]; break;
                     }
           }
               System.out.println(out);
       }
 }
```

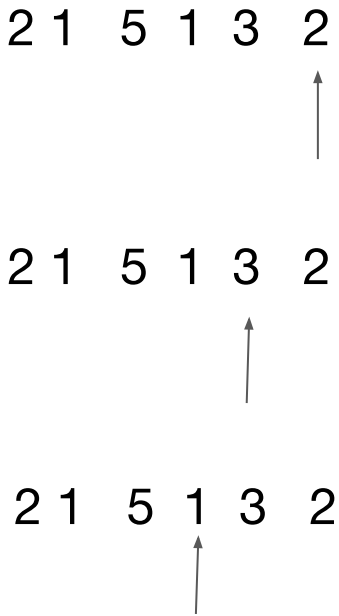
LOGIC

2 1 5 1 3 2

STACK

5
1
3
2

Stack is not empty : s.top() [Greater the array element]
Stack is empty: -1
s.pop() [Less than the element] && stack is not empty



Stack	output
2	-1
pop()	-1
3	
1	3
3	

Second Approach

Time Complexity: O(n) Space Complexity: O(n)

```
public static int function(int[] arr) {
        Stack<Integer> s = new Stack<>(); int new_arr[] = new int[arr.length], i=0;
    for (int itr = arr.length-1; itr>=0; itr --) {
            if(s.size()==0)      new_arr[i++]=-1;

            else if( s.size()>0  && s.peek() >arr[itr])
                    new_arr[i++]= s.peek();

            else if(s.size()>0 && s.peek() <= arr[itr]){
                  while(s.size()>0   &&  s.peek()<=arr[itr] )   s.pop();
                   if(s.size()==0)       new_arr[i++] = -1;
                   else                   new_arr[i++]=s.peek();
                }
            s.push(arr[itr]);
           // return the reversed new_arr
    }
}
```

QUESTION

Find the Next Greater element to left

Input: [2, 1, 5, 1, 3, 2]

Output: [-1,2,-1,5, 5,3]

First Approach

Time Complexity: O(n*n) Space Complexity: O(1)

```
        for (int itr = 0; itr < arr.length; itr ++) {

          for( int jtr=itr-1; jtr>=0; jtr--) {


       }
 }
```

Second Approach

Time Complexity: O(n) Space Complexity: O(n)

```
public static int function(int[] arr) {
        Stack<Integer> s = new Stack<>(); int new_arr[] = new int[arr.length], i=0;
    for (int itr = 0; itr<arr.length; itr ++) {
            if(s.size()==0)      new_arr[i++]=-1;

            else if( s.size()>0  && s.peek() >arr[itr])
                    new_arr[i++]= s.peek();

            else if(s.size()>0 && s.peek() <= arr[itr]){
                  while(s.size()>0   &&  s.peek()<=arr[itr] )   s.pop();
                   if(s.size()==0)       new_arr[i++] = -1;
                   else                   new_arr[i++]=s.peek();
                }
            s.push(arr[itr]);
    }
}
```

QUESTION

Find the Next Smallest element to left

Input: [2, 1, 5, 1, 3, 2]

Output: [-1, -1 ,1, -1, 1, 1]

First Approach

Time Complexity: O(n*n) Space Complexity: O(1)

```
        for (int itr = 0; itr < arr.length; itr ++) {

          for( int jtr=itr-1; jtr>0; jtr--) {


       }
 }
```

Second Approach

Time Complexity: O(n) Space Complexity: O(n)

```
public static int function(int[] arr) {
        Stack<Integer> s = new Stack<>(); int new_arr[] = new int[arr.length], i=0;
    for (int itr = 0; itr<arr.length; itr ++) {
            if(s.size()==0)      new_arr[i++]=-1;

            else if( s.size()>0  && s.peek() < arr[itr])
                    new_arr[i++]= s.peek();

            else if(s.size()>0 && s.peek() >= arr[itr]){
                  while(s.size()>0   &&  s.peek()>=arr[itr] )   s.pop();
                   if(s.size()==0)       new_arr[i++] = -1;
                   else                   new_arr[i++]=s.peek();
                }
            s.push(arr[itr]);
    }
}
```

QUESTION

Find the Next Smallest element to right

Input: [2, 1, 5, 1, 3, 2]

Output: [1, -1 ,1, -1, 2, -1]

First Approach

Time Complexity: O(n*n) Space Complexity: O(1)

```
        for (int itr = 0; itr < arr.length; itr ++) {

          for( int jtr=itr+1; jtr<arr.length; jtr++) {


       }
 }
```

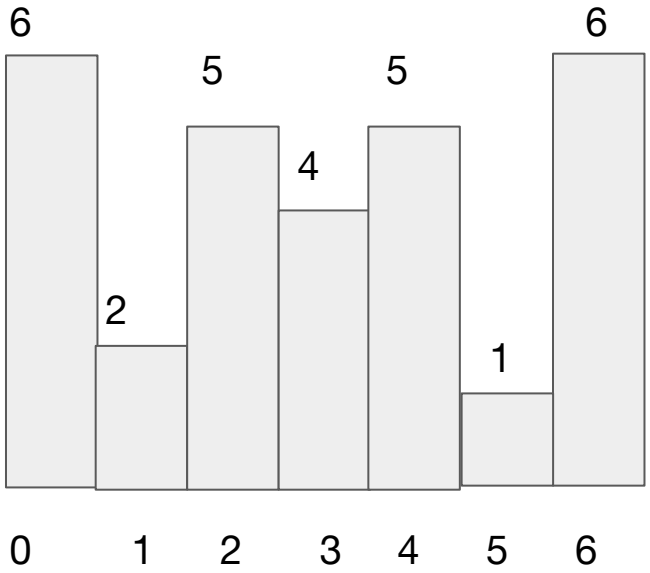
Second Approach

Time Complexity: O(n) Space Complexity: O(n)

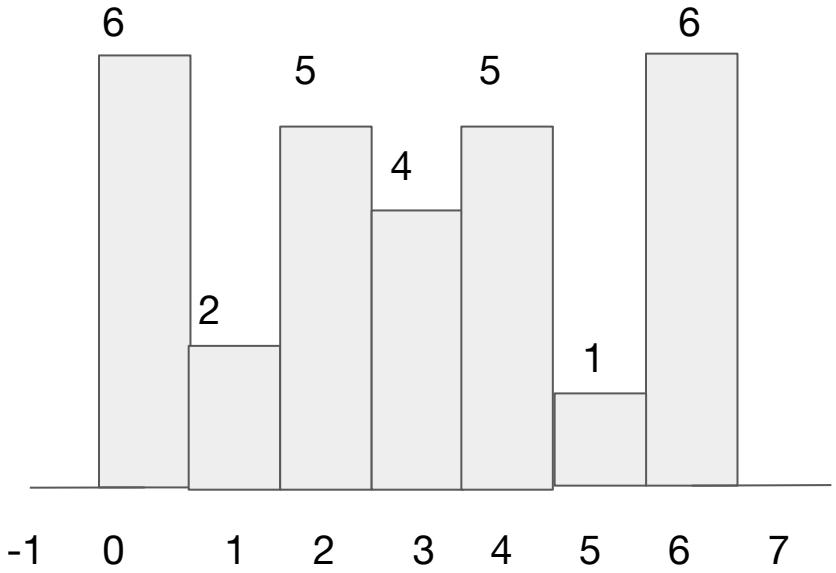
```
public static int function(int[] arr) {
        Stack<Integer> s = new Stack<>(); int new_arr[] = new int[arr.length], i=0;
    for (int itr =arr.length-1;itr>=0; itr --) {
            if(s.size()==0)      new_arr[i++]=-1;

            else if( s.size()>0  && s.peek() < arr[itr])
                    new_arr[i++]= s.peek();

            else if(s.size()>0 && s.peek() >= arr[itr]){
                  while(s.size()>0   &&  s.peek()>=arr[itr] )   s.pop();
                   if(s.size()==0)       new_arr[i++] = -1;
                   else                   new_arr[i++]=s.peek();
                 }
            s.push(arr[itr]);
           // return the reversed new_arr
    }
}
```

Maximum Rectangular Area in A Histogram

Given : [6 2 5 4 5 1 6]



Maximum Rectangular Area in A Histogram



Approach

2 1 4 1 1 -1 -1 Next Smallest Right
1 5 3 5 5 7 7 store indexes & reverse

-1 -1 2 2 4 -1 1 Next Smallest Left
-1 -1 1 1 3 -1 5 store indexes

max = Math.max(max, arr[i]*{ r[i]-l[i]-1})

Kth largest element

Input: [6 2 5 4 1 7]
K = 3

Output: 5

Input: [16 2 50 4 10 70]
K = 3

Output: 16

First Approach

Sort the Input array: [6 2 5 4 1 7]

[1 2 4 5 6 7]

Then, arr[k-1] is the element

Output: 4

First Approach

Sort the Input array: [6 2 5 4 1 7]

[1 2 4 5 6 7]

Then, arr[arr.length - k] is the element

Output: 5

Second Approach

```
PriorityQueue<Integer> heap = new PriorityQueue<Integer>(k);
  for(int i =0;i<arr.length;i++){
      heap.add(arr[i]);

      if(heap.size()>k){
          heap.poll();
      }
  }
  return heap.poll();
```

Sort nearly sorted array

Input: [2 5 4 1 7]
K = 3

Output: [1 2 4 5 7]

Input: [16 2 50 4 10 70]
K = 3

Output: [2 4 10 16 50 70]

First Approach

Sort the Input array: [2 5 4 1 7]

Output: [1 2 4 5 7]

But ‘ k’ was never used.

Second Approach

```
PriorityQueue<Integer> heap = new PriorityQueue<>();

        for(int i = 0; i < k + 1; i++)
        {
            heap.add(arr[i]);
        }

        int idx = 0;
        for(int i = k + 1; i < arr.length; i++)
        {
            arr[idx++] = heap.poll();
            heap.add(arr[i]);
        }

         while (!heap.isEmpty()) {
             arr[idx++] = heap.poll();
        }
```

Minimize the cost of connecting ropes

Input: [2 5 4 1 7]

Output: 40

Minimize the cost of connecting ropes

Input: [2 5 4 1 7]

Cost: 3+7+ 12 + 19 = 41

1+2 = 3

3 5 4 7

3+ 4 = 7

7 5 7

5+7 = 12

12 7

12+7

19

Minimize the cost of connecting ropes

Input: [2 5 4 1 7]

If we do sorting:
1 2 4 5 7

1+ 2 = 3 3 4 5 7

3+ 4 = 7 5 7 7

5+ 7 = 12 7 12

7 + 12= 19 cost : 41

Minimize the cost of connecting ropes

```
PriorityQueue<Integer> heap = new PriorityQueue<>();
    int min =0;
        for(int i = 0; i<arr.length; i++)
        {
            heap.add(arr[i]);
        }



         while (!heap.size()>1) {
             int temp = heap.remove()+ heap.remove();

             min += temp;
        }

        return min;
```

Summary

- We learnt how to solve different coding problems with different approaches.
- How to reduce time complexity?
- How to reduce space complexity?

Thank You