Operating Systems and Algorithms - Assignment 1

Analysis Report

Sreeja Guduri (2021102007)

1. N-th term of a GP

ALGORITHM:

The logic used in the code is to calculate the power of k using recursion. We know that if n is even, and if we know the answer to k^(n/2) then k^n will just be k^(n/2) * k^(n/2) .Similarly, if n was odd then the answer would be k^(n/2) * k^(n/2) * k.

Thus, we recursively call the function for n/2 until we reach the base case which is (n<=1) at which point, we return 'ans' as 1.

TIME COMPLEXITY:

The problem is a typical divide-and-conquer question. Since, we divide the problem into two halves every time, the time complexity of this step is O(logn).

Now, when we put it back together, we just multiply the answer with itself, and multiplication is an O(1) operation. Thus, the time complexity of this algorithm is O(logn) with a base 2.

Since, we call the function for 'T' test cases the total time complexity would come up to O(T*logn).

SPACE COMPLEXITY:

Every time we call the function, we make a new frame in the stack and we half it 'logn' times. Thus, the space used is proportional to the depth of the stack which is logn.

Hence, space complexity is O(logn) with a base 2.

2. N-number bucket problem

ALGORITHM:

The idea is to use a modified binary search to find the least maximum value we could be left with after performing the operation ‘k’ times.

We first find the maximum element in the array, which will be the answer if we don’t find a way to end up with a smaller max element by performing the operation. So now perform binary search by taking mid as the average of the low and high elements.

We see if we can reach ‘mid’ as the lowest max element in at most k steps. If we find that the number of steps goes above k, we know that ‘mid’ can’t be the max element so we update low to be (mid+1) and search the second half for a higher max.

However, if we find that we can get to mid in at most k steps, our task now is to find out if we can get to a lower max element. So we update high to mid and now search in the lower half.

In the end we return ‘high’ cause it will contain the max element that could be reached in at most k steps.

TIME COMPLEXITY:

It is clear from the algorithm that we keep dividing the array into two halves based on the value of the max element of the array. Thus, the time complexity of halving the array is log(M) where M is the max element.

We also go through the N length array to calculate the number of steps it would take to reach ‘mid’ every time, which makes the total time complexity O(N*logM).

SPACE COMPLEXITY:

Since we just use a vector to store the elements the space complexity should be O(n).

3. COUNT THE NUMBER OF INTERSECTIONS

ALGORITHM:

This approach uses a modified merge sort algorithm to get to the answer. We first realise that if P[i] < P[j] (where P is the vector of x coordinates on the line y = 10) then the two line segments intersect only if Q[j] < Q[i] (where Q is the vector of x coordinates on the line y = 0)

Thus, we first sort the P vector and then arrange the elements of Q such that they have the same index as their corresponding x-coordinate in P. Then, the problem just boils down to calculating the number of inversions in the array Q. This means that we need to calculate the number of times i < j (which implies P[i] <P[j]) but Q[j] < Q[i].

To calculate the number of inversions, we use the logic of merge sort. This is because when doing merge sort we can calculate the number of inversions in the right and left subarrays and then the number of inversions when merging.

To find the number of inversions during merging, if we know that an element a[i] in the left subarray is greater than a[j] in the right subarray, then every element after a[i] to a[mid] will also be greater than a[mid] which results in (mid - i) inversions.

The total number of intersections is thus the number of inversions of the second x-coordinate vector.

TIME-COMPLEXITY:

We initially sort the P vector using the sort function which is O(nlogn). Also, since we use a modified form of merge sort, it results in the same time complexity. The time to divide the problem into halves is logn and the time to merge the halves together is n. Thus, the total time complexity is O(nlogn).

SPACE-COMPLEXITY:

The space complexity of the problem will be O(n) because we are just dealing with n elements at a time and not storing anything else.