Assignment- Longest Monotonically Increasing Subsequence

Let $A := a_1, a_2, a_3, a_4 \ldots a_n$, $B := b_1, b_2, b_3, b_4, \ldots, b_k$ be sequence of +ve integers.

We say that B is a subsequence of A if $k < n$ and $\exists$ an increasing function $i : \{1, 2, 3, \ldots, k\} \to \{1, 2, 3, \ldots, n\}$ such that $b_j = a_{i(j)}$ $\forall$ $1 \leq j \leq k$.

If B is monotonically increasing, i.e, $b_j \leq b_{j'}$ whenever $j < j'$, then B is said to be a monotonically increasing subsequence of A.

A longest monotonically increasing subsequence (LMIS) of A is an increasing subsequence of A of maximum length.

Example: Let $A = \{20, 50, 30, 10, 40\}$. The sequence $\{10, 40\}$ is monotonically increasing of A even though it is not the longest. Finally, the sequence $\{20, 30, 40\}$ is the LMIS of A.

ALGORITHM:

A: $\{a_1, a_2, a_3, a_4, \ldots, a_n\}$, find LMIS of A.

Solution by Dynamic Programming.

For each $1 \leq i \leq n$, define $m(i)$ to be the length of any longest monotonically increasing subsequence of $a_1, a_2, a_3, \ldots a_i$ that has a_i as the last element of subsequence.

We seek $\max\{m(i) : 1 \leq i \leq n\}$.

OPTIMAL SUBSTRUCTURE PROPERTY

If $i=1$, then a_i is obviously the LMIS of itself. Thus $m(1) = 1$.

Now suppose $1 < i \leq n$.

Fix such i. Let $B := b_1, b_2, b_3, \ldots, b_K$ be an LMIS of $a_1, a_2, a_3, \ldots a_i$ subject to $b_K = a_i$. Then $b_1, b_2, b_3, \ldots, b_{K-1}$ must be an LMIS of $a_1, a_2, a_3 \ldots a_{i-1}$.

This means that there exists some j, where $1 \leq j \leq i-1$, such that $b_1, b_2, \ldots, b_{K-1}$ is an LMIS of $a_1, a_2, \ldots, a_j$ subject to $b_{K-1} = a_j$.

RECURRENCE

The above reasoning shows $m(i)$ satisfies the recurrence.

$$m(i) = \begin{cases} 1 & : \text{if } i = 1 \\ \max\{1, m(j)+1 : 1 \leq j < i \text{ and } a_j \leq a_i\} & \text{if } 1 < i \leq n. \end{cases}$$

RUNNING TIME $\div O(N^2)$, where N = size of the array.

Space complexity : $O(N)$

Elements of the Dynamic Programming Approach.

* i) We have to define problem variables.

↳ The state of the problem depends only on one parameter i.e., N here, the size of the array.

* ii) We have to define the size and table structure:

↳

Dry Run of this approach

input : arr[] = {3, 10, 2, 11}

LMIS [] = {1, 1, 1, 1} = initialize

Iteration-wise Simulation :-

1.) arr[2] > arr[1] → {LMIS[2] = max(LMIS[2], 1+LMIS[1]) = 2}

2.) arr[3] < arr[1] → {No change}

3.) arr[3] < arr[2] → {No change}

4.) arr[4] > arr[1] → {LMIS[4] = max(LMIS[4], 1+LMIS[1]) = 2}

5.) arr[4] > arr[2] → {LMIS[4] = max(LMIS[4], 1+LMIS[2]) = 3}

6.) arr[4] > arr[3] → {LMIS[4] = max(LMIS[4], 1+LMIS[3]) = 3}

Hence, the length of the LMIS[] of the given arr[] is 3.

Code : Pseudo code

```
1. function LMIS(a1, a2, a3, ... an)
2.     for K ← 1 to n do
3.         length[K] ← 1
4.         for j ← K+1 to n do
5.             if aj > aK then
6.                 length[j] ← max{length[j], 1+length[K]}
7.     return max 1≤i≤n {length[i]}
```

Assignment - 2 → Find 1st & 2nd Maximum/Minimum.

Pseudo Code :- Algorithm.

Step- i) Initialize 1st-largest variable = 0 &
2nd largest variable = -1

" ii) Traversing the array [i.e., i ← 0 to n-1]

a) If curr_element, i.e, a[i] > a[First]
then, second = First
First = a[i].

b) If the current element lies between first and second, then update second to store the value of current variable as -
second = a[i]

" iii) Return the value stored in second.

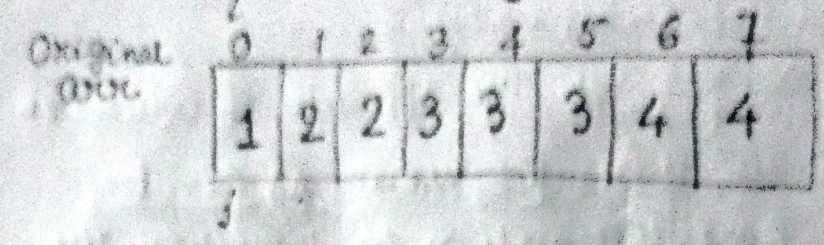
Complexity Analysis :- Here in this algorithm only one traversal of the array is required.

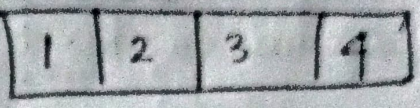
Hence, Time-Complexity : O(n)
As no extra space is required
Hence, Space Complexity : O(1).

Code :—

```
function_secondlargest(int arr[], int n)
{
    int first = 0, second = -1;
    for (int i=1; i<n; i++){
        if (a[i] > a[first]) {
            second = first;
            first = i;
        }
        else if (a[i] < a[first]) {
            if (second == -1 || a[second] < a[i])
                second = i;
        }
    }
    return second;
```

Assignment -3 → Write an efficient program to delete all duplicate elements from an sorted array. (using constant space)





Pseudo code :→

Step: i) initialize j=0

ii) for i=0 to n-2

iii) if arr[i] != arr[i+1]

iv) arr[j] = arr[i]

v) j++

vi) arr[j] = arr[n-1]

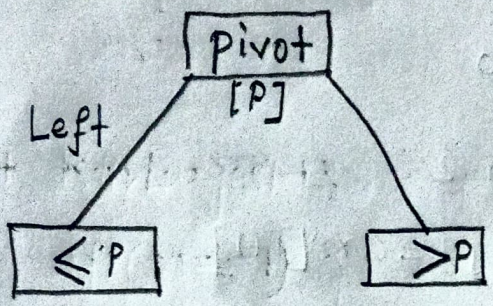
vii) for i=0 to j-1 // print first j elements of array arr.
print arr[i]

Complexity Analysis : The time complexity is O(N).

Space Complexity :→ As there is no extra space needed. Thus space complexity is O(1).

Assignment 4 :→ Finding K-th maximum element by partitioning the array.

In this approach we will use the approach of QuickSort Algorithm. In this approach, we need a pivot element and a partition function to partition the array around the pivot element.



Using This approach, we can find out the (n-K)th smallest element (because it is the same as finding out the Kth-largest element in the array) And Finally, we will return to this element.

Algorithm :—

Step-I :- Take size of the array = N.
Check if (N==1):
Return arr[0]

Step-II :- Create variables : i) lower, ii) high, iii) target.
Initialize it : lower = 0, high = N-1, target = N-K

Step-III : Run a while loop until lower ≤ high, and perform the following steps.

(a) Create a variable pivot and initialize it with the value returned by the partition function.
i.e, pivot = partition (arr, lower, high)

(b) if (pivot < target) then
lower = pivot+1

(c) Else if (pivot > target) then
high = pivot-1

(d) Else return arr[pivot] (this is =arr[n-k])

Step-IV : Partion_function (arr, l, h)

i) Create two variables and assign it high = arr[h] and x = l

ii) Run a loop from i=l to (h-1)

a) If (arr[i] < high) Then
swap(arr[x], arr[i])
x++

iii) After the loop has ended,
swap(arr[x], arr[h])

Code :-

```
Partition_function(int arr[], int l, int h){
    int high = arr[h];
    int x = l;
    for(int i=l; i<h; i++){
        if(arr[i] < high){
            swap(arr[x], arr[i]);
            x++;
        }
    }
    swap(arr[x], arr[h]);
    return x;
}

int KthLargestElement(int a[], int n, int k){
    if(n==1){
        return a[0];
    }
    int l = 0;
    int h = n-1;
    int target = n-k;
    while(l<=h){
        int pivot = partition(arr, l, h);
        if(pivot < target){
            l = pivot + 1;
        }
        else if(pivot > target){
            h = pivot - 1;
        }
        else{
            return a[pivot];
        }
    }
    return -1;
}
```

Complexity Analysis: In this approach, we are performing at most n-operations by running a while loop from l=0 to h=(n-1) where l = lower index and h = higher index, which can take O(N) time-complexity in the worst case (l=0 and h=n-1).

Inside the while loop, we are calling the partition function, which is taking O(N) time complexity in the worst case because we are running a loop from i=1 to (h-1).

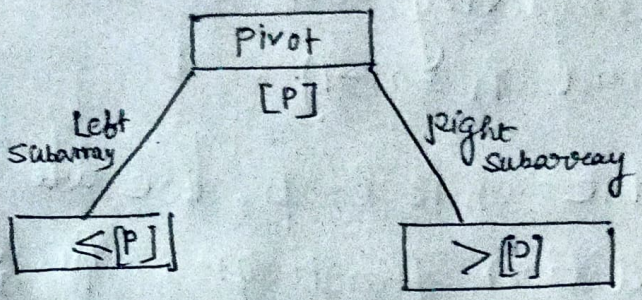
So, the total worst case time-complexity for this approach to find the K-th largest element in an array where n is the size of the array is $O(N) * O(N) = O(N^2)$

Space Complexity :— In this approach, we are not using any space, so the space complexity for this approach is O(1).

ASSIGNMENT-5 : QUICKSORT AIGORITHM

Quicksort algorithm is based on divide and conquer approach where —

i) An array is divided into subarrays by selecting a pivot element from the array. in such a way that-

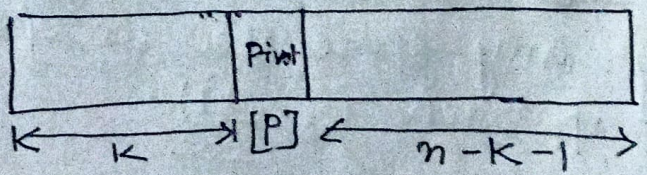


ii) The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.

iii) At this point, elements are already sorted. Finally elements are combined to form a sorted array.

Complexity Analysis.

Assuming after partition left side of the pivot has K - no. of elements. We take an array of n- integer.



$\therefore T(n) = n + T(K) + T(n-k+1)$

↓
For partition

Best Case: After partition, ~~list of~~ array is divided into equal halves.

$$\therefore T(n) = n + T(n/2) + T(n/2)$$

$$= 2T(n/2) + n$$

$$= 2\left(2 \cdot T(n/2^2) + n\right) + n$$

$$= 2^2 \cdot T(n/2^2) + 2n + n$$

$$= 2^2\left[T(n/2^3) + n\right] + 2n + n$$

$$= 2^3 \cdot T(n/2^3) + 2^2 n + 2^1 n + 2^0 n$$

$$\vdots$$

K^{th} term

$$= 2^K \cdot T(n/2^K) + n\left(2^0 + 2^1 + 2^2 + \cdots + 2^{K-1}\right)$$

If $\frac{n}{2^K} = 1$

$2^K = n$

$\boxed{K = \log_2 n}$

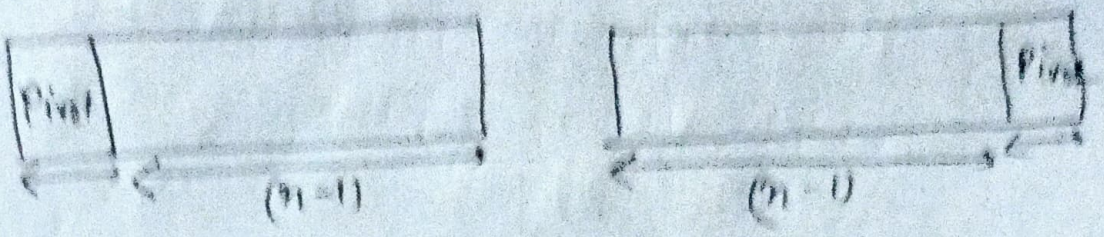
Hence, $T(n) = 2^K \cdot T(n/2^K) + n(2^K - 1)$

$= 2^K \cdot T(n/2^K) + n \cdot C_1 \quad (C_1 = 2^K - 1 = \text{Const})$

$\therefore T(n) = n \cdot T(1) + n \cdot \log_2 n$

$= O(n \log n)$.

Worst case :- After partition, array is divided into two parts, one consists of single element and other of $(n-1)$.



$\therefore T(n) = n + T(n-1)$

Now, $T(n) = n + T(n-2) + (n-1)$

$= n + (n-1) + T(n-2)$

$= n + (n-1) + T(n-3) + (n-2)$

$= n + (n-1) + (n-2) + T(n-3)$

$\vdots$

k-th term $= n + (n-1) + (n-2) + \cdots (n-k) + T(n-k)$

$= k \cdot n - \frac{k(n+1)}{2} + T(n-k)$

$= kn - \frac{k(k+1)}{2} + T(n-k)$

put $n = k$

$T(n) = T(0) + n^2 - \frac{n(n+1)}{2}$

$= T(0) + \frac{n(n-1)}{2}$

$= O(n^2)$

Hence, In worst case, time complexity of Quick sort is $O(n^2)$. If first or last element is choosen as pivot, then already sorted list gives the worst case complexity of $O(n^2)$.

Space Complexity -> O(1) -> Const. time

As there is no extra space needed in the sorting process.

ALGORITHM :->

```
partition (A, P, q)
    x ← A[P]
    i ← p-1
    j ← r+1
    while (True) {
```

```
int partition (int a[], int low, int high) {
    pivot = A[l];
    i = l, j = h
    while (i < j)
    {
```

```
partition (A, P, q)
{   x = A[P] ; // Pivot
    i = P ;
for (j = p+1 ; j <= q ; j++) {
        if (A[j] <= x)
            i = i+1;
        swap (A[i], A[j])
    }
}
    swap (A[i], A[P]);
}
```

```
Void QuickSort (int a[], int low, int high) {
    if (low < high) {
        int pivot = partition(a, low, high);
        Quicksort(a, low, pivot-1);
        Quicksort(a, pivot+1, high);
    }
}
```