16/06/2023

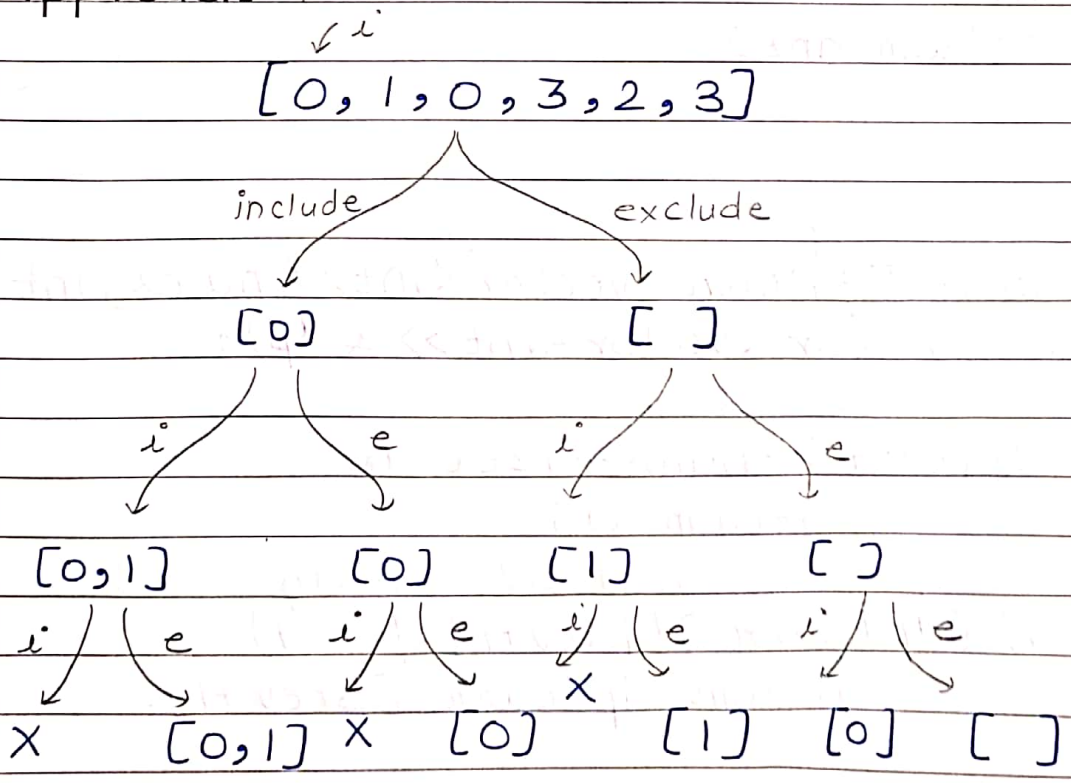
Q1 Longest increasing subsequence

i/p → {5, 8, 3, 2, 1, 9, 7}

Total subsequences possible = 2^n. Filter out the increasing subsequence and we have to return the longest length subsequence.

Increasing subsequence → [5, 8, 9] → example of increasing subsequence

Approach



Note → We need to include a particular element only when it is bigger than previous element.

Code

```
int solveRec (vector<int> &nums, int curr, int prev)
    // Base case
    if (curr >= nums.size())
        return 0;
    // include call
    int include = 0;
    // 1st element or previous element smaller
    if (prev == -1 || nums[curr] > nums[prev]){
        include = 1 + solveRec (nums, curr+1, curr);
    }
    //exclude call
    int exclude = 0 + solveRec (nums, curr+1, prev);
    int ans = max(include, exclude);
    return ans;
}
```

remains same

```
// Top down approach
int solveTopDown (vector<int>& nums, int curr,
int prev, vector<vector<int>> &dp){
    // Base case
    if (curr >= nums.size())
        return 0;
    //Step 3: Check if ans already exists
    if (dp[curr][prev+1] != -1)
        return dp[curr][prev+1];
    //include call
    int include = 0;
    if (prev != -1 || nums[curr] > nums[prev])
        include = 1 + solveTopDown (nums,
        curr+1, curr, dp);
    }
    //exclude
```

```
    int exclude = 0 + solveTopDown(nums, curr+1,
    prev, dp);
    //Step 2: Store ans in dp array
    // prev+1 as prev = -1 can be there and prev+1 = 0.
    dp[curr][prev+1] = ans;                    valid index
    return dp[curr][prev+1];
}
//Bottom up approach
int solveTab(vector<int> &nums){
    int n = nums.size();
    //Step 1: Create dp array
    vector<vector<int>> dp(n+1, vector<int>(n+1,0));
    //Step 2: Observe base case of top-down → Handled
    //Step 3: Reverse flow of top-down
    for(int curr = n-1; curr >= 0; curr--){
        for(int prev = curr-1; prev >= -1; prev--){
            //include call
            int include = 0;
            if(prev == -1 || nums[curr] > nums[prev]){
                include = 1 + dp[curr+1][curr+1];
            }
            //exclude
            int exclude = 0 + dp[curr+1][prev+1];
            dp[curr][prev+1] = max(include, exclude);
        }
    }
    return dp[0][0];
}
```

* Space optimization possible or not ?
Yes it is possible.

1) Create 2,1D arrays.
vector <int> currRow (n+1,0);
vector <int> nextRow (n+1,0);

2) Replace dp[curr+1]-- with nextRow[--]
and replace dp[curr]-- with currRow[--]

3) Shift → nextRow = currRow → Going upwards

DP with binary search

arr → [10, 9, 2, 5, 3, 7, 101, 18]

i = 0 → [10]
i = 1 → [9] (Overwrite)
i = 2 → [2] (Overwrite)
i = 3 → [2, 5] (Include)
i = 4 → [2, 3] (Overwrite)
i = 5 → [2, 3, 7] (Include)
i = 6 → [2, 3, 7, 101] (Include)
i = 7 → [2, 3, 7, 18] → length = 4
(Overwrite)

Note → overwrite and include is to be done.

Code

```
int solveOptimal (vector<int> & arr) {

    if (arr.size() == 0)
        return 0;
    vector <int> ans;
    ans.push_back(arr[0]);
```

```
for (int i = 1 ; i < arr.size() ; i++) {
        if (arr[i] > ans.back())
                ans.push_back(arr[i]);
        else { //overwrite
        // find index of just bigger element
         int index = lower_bound(ans.begin(),
         ans.end(), arr[i]) - ans.begin();
        ans[index] = arr[i];
    }
}
    return ans.size();
}
```

TC = O(nlogn)

* Russian doll approach

arr → { [5,4], [6,4], [6,7], [2,3] }

(5 → width, 4 → height)

Arrange them in increasing order of width. If width is same, then take higher height. Simply apply LIS solution on the width now.

[2,3]
[5,4]
[6,7]
[6,4]

Now apply LIS on [2,5,6,6] and hence here 3 is the answer.

Q2 Max height by stacking cuboid.

i/p → [50,45,20]
[95,37,53]
[45,23,12]

1) First sort. (We need maximum height.

w l h
[20,45,50]
[37,53,95]
[12,23,45]

2) Now sort according to the width.

w l h
[12,23,45]
[20,45,50]
[37,53,95]

3) Now simply apply LIS.

Code

```
int maxHeight (vector <vector <int>> & cuboids){
    //Sort every array
    for (auto &a : cuboids)
        sort (a.begin(), a.end());
    //Sort 2D array
    sort (cuboids.begin(), cuboids.end());
    //Apply LIS logic
    int ans = SpaceOpt (cuboids);
    return ans;
}
```

```
bool check (vector <int>a , vector <int>b) {
    //Compare length , width & height
    if (b[0] <= a[0] && b[1] <= a[1] &&
        b[2] <= a[2])
            return true ;
    else
            return false ;
}
```

Changes in LIS code (Space optimization)
majorly changed.

```
if (prev == -1 && check (nums [curr] , nums [prev])
{   // height to be added instead of 1
    include = nums [curr][2] + nextRow [curr+1];
}
```