takeUforward

~ Strive for Excellence

October 24, 2021 ▪ Arrays / Data Structure

next_permutation : find next lexicographically greater permutation

Problem Statement: Given an array Arr[] of integers, rearrange the numbers of the given array into the lexicographically next greater permutation of numbers.

If such an arrangement is not possible, it must rearrange it as the lowest possible order (i.e., sorted in ascending order).

Example 1 :

```
Input format: Arr[] = {1,3,2}
```

```
Explanation: All permutations of
{1,2,3} are {{1,2,3} , {1,3,2},
{2,13} , {2,3,1} , {3,1,2} ,
{3,2,1}}. So, the next permutation
just after {1,3,2} is {2,1,3}.
```

Example 2:

```
Input format: Arr[] = {3,2,1}

Output: Arr[] = {1,2,3}

Explanation: As we see all
permutations of {1,2,3}, we find
{3,2,1} at the last position. So,
we have to return the topmost
permutation.
```

Solution

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Solution 1 Brute Force: Finding all possible permutations.

Approach :

Step 1: Find all possible permutations of elements present and store them.

Step 2: Search input from all possible permutations.

Longest String Chain | (DP- 45)
Longest Divisible Subset | (DP-44)
Longest Increasing Subsequence | Binary Search | (DP-43)
Printing Longest Increasing Subsequence | (DP-42)
Longest Increasing Subsequence | (DP-41)

Accolite Digital Amazon Arcesium Bank of America Barclays BFS Binary Search Binary Search Tree Commvault CPP DE Shaw DFS DSA Self Paced google HackerEarth infosys inorder Java Juspay Kreeti

Step 3: Print the next permutation present right after it.

For reference of how to find all possible permutations, follow up https://www.youtube.com/watch?v=f2ic2Rsc9pU&t=32s. This video shows for distinct elements but code works for duplicates too.

Time Complexity :

For finding, all possible permutations, it is taking N!xN. N represents the number of elements present in the input array. Also for searching input arrays from all possible permutations will take N!. Therefore, it has a Time complexity of O(N!xN).

Space Complexity :

Since we are not using any extra spaces except stack spaces for recursion calls. So, it has a space complexity of O(1).

Solution 2 : Using C++ in-built function

C++ provides an in-built function called next_permutation() which directly returns the lexicographically next greater permutation of the input.

C++ Code

```
#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

int main() {
    int arr[] = {1,3,2};

    next_permutation(arr,arr+3);//using in

    cout<<arr[0]<<" "<<arr[1]<<" "<<arr[2]

    return 0;
}
```

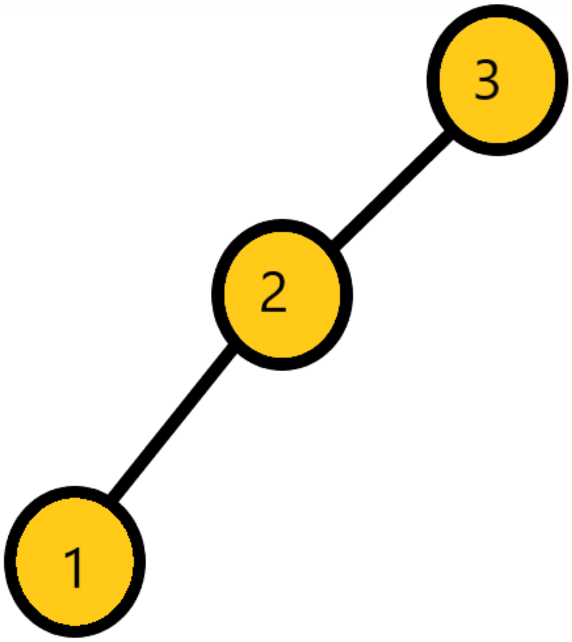
Solution 3 :

Intuition :

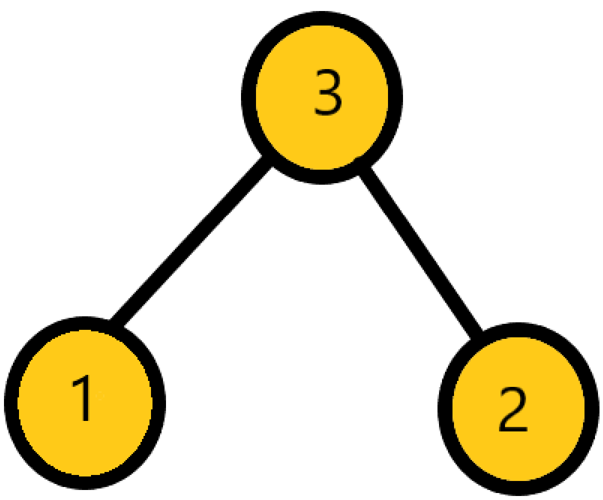
Intuition lies behind the lexicographical ordering of all possible permutations of a given array. There will always be an increasing sequence of all possible permutations when observed.

Let's check all sequences of permutations of {1,2,3}.

- {1,2,3}

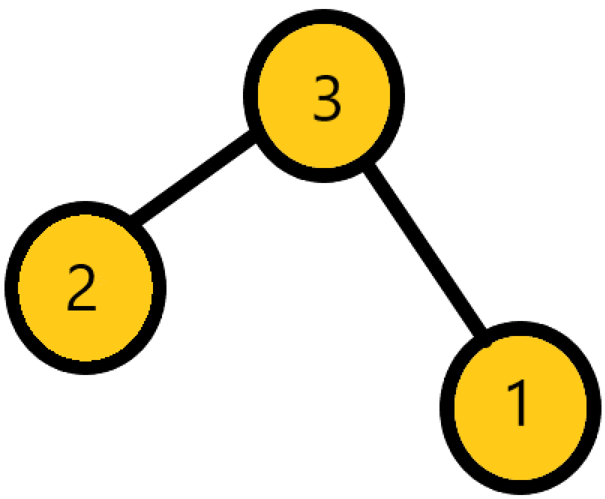


- {1,3,2}



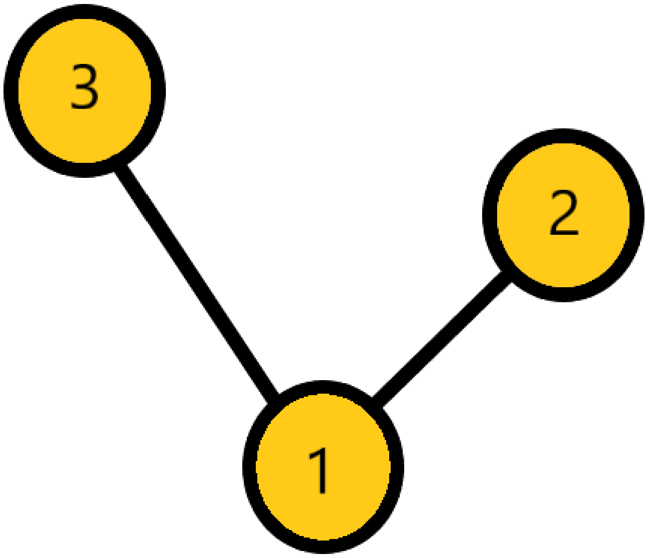
- {2,1,3}

- {2,3,1}

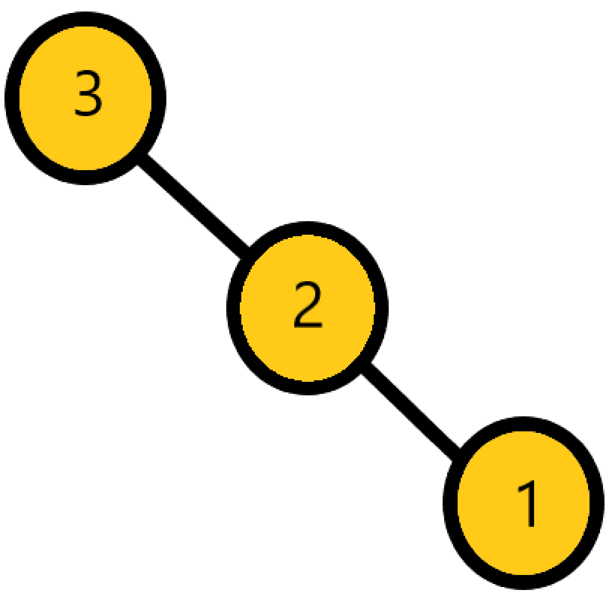


- {3,1,2}

×



- {3,2,1}



Thus, we can see every sequence has increasing order. Hence, our approach aims to get a peak from where the increasing sequence starts. This is what we achieve from our first step of the approach.

Then, we need to get just a larger value than the point where the peak occurs. To make rank as few as possible but greater than input array, just perverse array from breakpoint achieved from the first step of the approach. We achieve these from all remaining steps of our approach.

Approach :

Step 1: Linearly traverse array from backward such that ith index value of the array is less than (i+1)th index value. Store that index in a variable.

Step 2: If the index value received from step 1 is less than 0. This means the given input array is the largest lexicographical permutation. Hence, we will reverse the input array to get the minimum or starting permutation. Linearly traverse array from backward. Find an index that has a value greater than the previously found index. Store index is another variable.

Step 3: Swap values present in indices found in the above two steps.

Step 4: Reverse array from index+1 where the index is found at step 1 till the end of the array.

Code :

C++ Code

```
class Solution {
public:
    void nextPermutation(vector<int>& nums
        int n = nums.size(), k, l;
        for (k = n - 2; k >= 0; k--) {
            if (nums[k] < nums[k + 1]) {
                break;
            }
        }
        if (k < 0) {
            reverse(nums.begin(), nums.end
        } else {
            for (l = n - 1; l > k; l--) {
                if (nums[l] > nums[k]) {
                    break;
                }
            }
            swap(nums[k], nums[l]);
            reverse(nums.begin() + k + 1,
        }
    }
};
```

Java Code

```
class Solution {
    public void nextPermutation(int[] A) {
        if(A == null || A.length <= 1) ret
        int i = A.length - 2;
        while(i >= 0 && A[i] >= A[i + 1])
        if(i >= 0) {
            int j = A.length - 1;
            while(A[j] <= A[i]) j--;
            swap(A, i, j);
        }
        reverse(A, i + 1, A.length - 1);
}
```

```
    A[j] = tmp;
}

public void reverse(int[] A, int i, int j)
    while(i < j) swap(A, i++, j--);
}
}
```

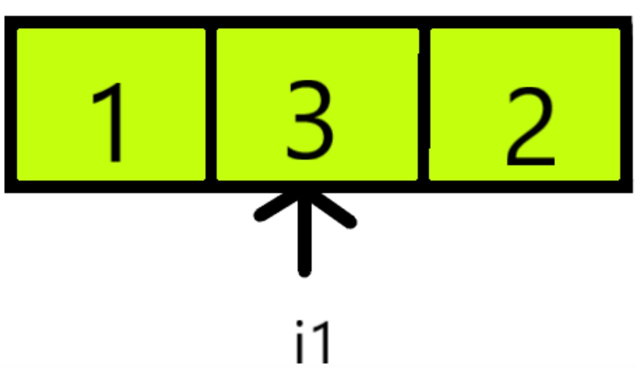
Time Complexity: For the first iteration backward, the second interaction backward and reversal at the end takes O(N) for each, where N is the number of elements in the input array. This sums up to 3*O(N) which is approximately O(N).

Space Complexity: Since no extra storage is required. Thus, its complexity is O(1).

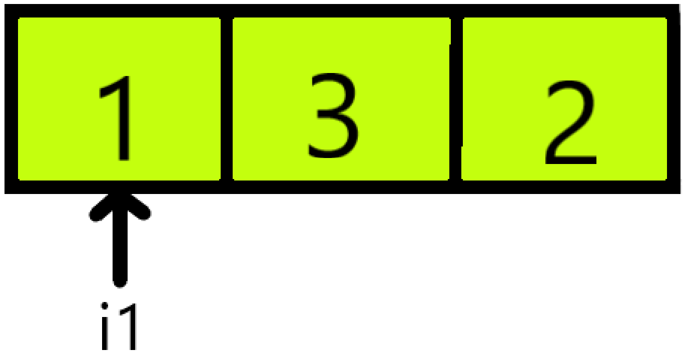
Dry Run :

We will take the input array {1,3,2}.

Step 1: First find an increasing sequence. We take i1 = 1. Starting traversing backward.

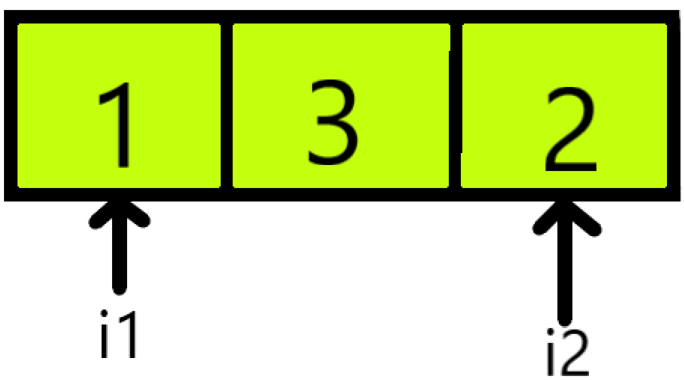


Step 2: Since 3 is not less than 2, we decrease i1 by 1.



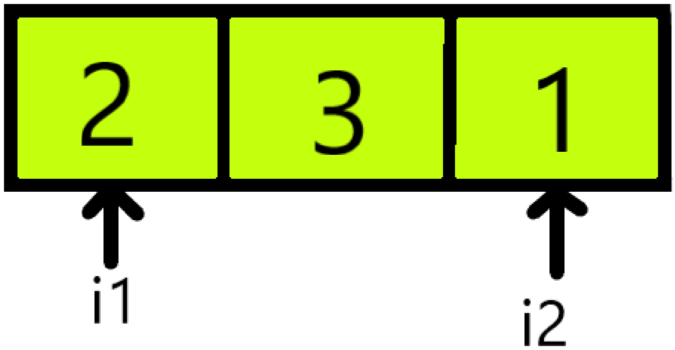
Step 3: Since 1 is less than 2, we achieved our start of the increasing sequence. Now, i1 = 0.

Step 4: i2 will be another index to find just greater than i1 indexed elements in the array. Point i2 to the last element.



Step 5: i2 indexed element is greater than i1 indexed element. So, i2 has a value of 2.

Step 6: Swapping values present in i1 and i2 indices.



Step 7: Reversing from i1+1 index to last of the array.

Thus, we achieved our final answer.

*Special thanks to **Dewanshi Paul** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article.*

Search in a sorted 2D matrix

Remove N-th node from the end of a Linked List

Load Comments

×