Next Permutation Problem

This lecture segment covers the **next permutation problem**, which involves finding the next lexicographically greater permutation of a given array of integers.

Understanding Permutations

- A **permutation** is a rearrangement of elements. For an array of `n` distinct elements, there are $n!$ possible permutations.
- The goal is to find the next permutation in **sorted (dictionary) order**.

For example, given the array `[3, 1, 2]`, the possible permutations in sorted order are:

1. `[1, 2, 3]`
2. `[1, 3, 2]`
3. `[2, 1, 3]`
4. `[2, 3, 1]`
5. `[3, 1, 2]`
6. `[3, 2, 1]`

The next permutation after `[3, 1, 2]` is `[3, 2, 1]`.

Edge Cases

If the given array is already the last permutation in sorted order (e.g., `[3, 2, 1]`), the next permutation is the first one (`[1, 2, 3]`).

Brute Force Approach

The brute force solution involves these steps:

1. **Generate** all possible permutations of the array.
2. **Sort** the permutations in dictionary order.
3. **Search** for the input array within the sorted permutations.
4. Return the next permutation in the list. If the input array is the last one, return the first permutation.

Generating Permutations

- Generating all permutations can be done using recursion.
- Refer to lectures 12 and 13 of the recursion playlist for detailed explanations.
- In an interview, describing the steps is usually sufficient without implementing the code.

Time Complexity

- Generating all permutations of an array of length `n` takes at least $n!$ time.
- Each permutation is of length `n`, so the overall time complexity is at least $O(n! * n)$.
- For an array of length 15, $n!$ is approximately 10^{12}, making the brute force approach highly inefficient.

Interview Strategy

- In an interview, describe the brute force approach at a high level.
- Mention the high time complexity and the need for optimization.

STL Solution (C++)

C++ provides an STL function called `next_permutation` that directly computes the next permutation of a given array.

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

int main() {
    std::vector<int> nums = {3, 1, 2};
    std::next_permutation(nums.begin(), nums.end());
    for (int num : nums) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

The interviewer might ask you to implement this function manually, leading to the most optimal solution.

Optimal Solution: Observations

To understand the optimal solution, we'll draw inspiration from how words are ordered in a dictionary.

- In a dictionary, words are sorted lexicographically.
- Words that are "next" to each other tend to have a longer prefix match.

For example, in the English dictionary, "raj" appears before "rax," and "rax" appears before "rbx" because the prefixes "ra" are the same. The longer the matching prefix, the closer the words are in lexicographical order.

Finding the Next Biggest Number

When trying to find the next biggest number, we need to consider prefix matches and how rearranging numbers affects the overall value.

Prefix Matching

If we have numbers like `0 0 3 4 5`, rearranging `0 0` won't change the prefix match. We aim to find the longest prefix match possible.

Impossibilities in Rearranging

- Matching `4`: If we match `4` with `2 1 5 4`, we have `3 0 0` left. No matter how we rearrange `3 0 0`, we can't get a number greater than the original.
- Matching `3`: Similarly, matching `3` gives us `4 3 0 0`, and no rearrangement makes it larger.
- Matching `2`: Matching `2` leaves `4 5 3 0 0`. Again, rearranging doesn't yield a greater number because we need a number greater than `5` to keep the first two prefixes the same.

One Prefix Match

With `1 5 4 3 0 0`, we can create `5 4 3 1 0 0`, which is greater. However, it's not immediately after `1 5 4 3 0 0`, as `2 4 5 3 1 0 0` exists in between.

Target

Our target is to find a number greater than `1` to replace it. Among available numbers `5, 4, 3, 0, 0`, we prefer the smallest number greater than `1`, which is `3`.

Why Rearranging Fails

Rearranging fails when we need a number greater than a certain value, but no such number exists in the remaining elements. For example, with `2 1 5 4 3 0 0`, we needed a number greater than `5`, which wasn't available.

Finding the Breakpoint

We're looking for a breakpoint in the graph of numbers.

> A breakpoint occurs when there's a dip in the increasing order of numbers.

This can be found by iterating through the array and checking when `array[i]` is smaller than `array[i+1]`.

$$breakpoint = when\ array[i] < array[i+1]$$

Observations and Algorithm Intuition

1. **Find the breakpoint:** Identify where the increasing sequence breaks.
2. **Find the smallest number greater than the breakpoint:** Locate the smallest number to the right of the breakpoint that is still greater than the number at the breakpoint.
3. **Place remaining numbers in sorted order:** After placing the number greater than the breakpoint, sort the remaining numbers in ascending order to keep the overall number as small as possible while still being greater than the original.

Detailed Explanation of Observations

- **Observation 1:** Find the longest prefix match by identifying the breakpoint.
- **Observation 2:** After finding the breakpoint, find the smallest number to the right of the breakpoint that is still greater than the number at the breakpoint, in order to create the next greater number. For example, switch `1` with `3` because `3` is the smallest number to the right that is greater than one
- **Observation 3:** Fill the rest of the empty slots with the remaining numbers in sorted order. This is because `2 3` is greater than `2 1`, so you want to keep everything as small as possible, so that the new number is right next to the old number.

Algorithm Design

Based on the observations, we can design the algorithm:

1. Find the **longest prefix match** and identify the **breaking point** (dip). A dip happens when the current number is greater than the next number.
2. After locating the dip, pick a number that is slightly greater than the number that caused the dip.

Example

Consider an array `1 2 3 4 5`. This is the smallest permutation. The last permutation is `5 4 3 2 1`.

Array	Dip?	Prefix Match Possible?
`1 2 3 4 5`	No	No
`5 4 3 2 1`	Yes	Yes

In `5 4 3 2 1`, there's a dip because the numbers are decreasing. The last index where a dip can occur is `n - 2`.

$lastIndexWithDip = n - 2$

Code

The first step is to find this index.

```
for (I = n-2; I >= 0; I--){
    if (array[I] is a dip value){
        index = I
        break
    }
}
```

If the initial index is -1, then there is no dip and the index does not exist.

Next Permutation Algorithm

Edge Case

If there is **no dip** (the array is in descending order, meaning the index is -1), it indicates that the current permutation is the last lexicographical one. In this case, you simply **reverse the array** to get the smallest (first) permutation.

Algorithm Steps

1. Find the Breakpoint:
    - Start from the back of the array.
    - Find the first element `arr[i]` that is smaller than the element to its right `arr[i+1]`. This point is called the "dip" or "breakpoint".

```
if arr[i] < arr[i+1]:
    # this is the break
```

    - If no such index is found, the given array is the last permutation, so reverse the entire array.
2. Find the Smallest Greater Element:
    - Again, start from the back of the array.
    - Find the element that is greater than `arr[index]` (the breakpoint) but also the smallest among all such elements.
    - Swap this element with `arr[index]`.

```
if arr[i] > arr[index]:
    # swap arr[i] and arr[index]
```

3. Reverse the Suffix:
    - Take the portion of the array to the right of the breakpoint (i.e., from `index + 1` to the end of the array).
    - Reverse this portion to make it in ascending order.

```
reverse(arr.begin() + index + 1, arr.end());
```

Why These Steps?

- **Swapping**: Swapping places `2 3` in the correct order relative to the elements before the breakpoint.
- **Reversing**: After swapping, the portion to the right of the breakpoint needs to be in ascending order to ensure the next smallest permutation. If it's already in decreasing order, reversing it achieves the smallest possible arrangement.

Example

Given the sequence `2 1 5 4 3 0 0`:

1. Breakpoint: `1` (since `1 < 5`).
2. Find smallest greater than `1` in `5 4 3 0 0`: It's `3`. Swap `1` and `3` to get `2 3 5 4 1 0 0`.
3. Reverse `5 4 1 0 0` to get `0 0 1 4 5`.

Result: `2 3 0 0 1 4 5`

Code Implementation

In C++, the algorithm can be implemented using the following steps:

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

using namespace std;

vector<int> nextPermutation(vector<int>& arr) {
    int n = arr.size();
    int index = -1;

    // Find the breakpoint
    for (int i = n - 2; i >= 0; i--) {
        if (arr[i] < arr[i + 1]) {
            index = i;
            break;
        }
    }

    // If there is no breakpoint, reverse the array
    if (index == -1) {
        reverse(arr.begin(), arr.end());
        return arr;
    }

    // Find the smallest greater element and swap
    for (int i = n - 1; i > index; i--) {
        if (arr[i] > arr[index]) {
            swap(arr[i], arr[index]);
            break;
        }
    }

    // Reverse the suffix
    reverse(arr.begin() + index + 1, arr.end());
    return arr;
}

int main() {
    vector<int> nums = {2, 1, 5, 4, 3, 0, 0};
    vector<int> result = nextPermutation(nums);

    cout << "Next permutation: ";
    for (int num : result) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Time and Space Complexity

- **Time Complexity**: $O(3n)$, which simplifies to $O(n)$ because the algorithm involves at most three passes through the array.
- **Space Complexity**: $O(1)$ because the algorithm modifies the array in-place without using extra space (excluding the space used to store the input array).