46. Permutations

<u>Array</u>

Backtracking

Medium

Problem Description

Given an array nums of distinct integers, the goal is to find all the possible permutations of these integers. A permutation is a rearrangement of the elements in an array in a different order. Since the array contains distinct integers, each permutation will also contain all the elements of the original array but in a different sequence. The result can be returned in any order, meaning that there is no need to sort or arrange the permutations in a particular sequence. The goal is to list all distinct ways the elements can be ordered.

The intuition behind the solution is to use **Depth-First Search (DFS)**. DFS is a common algorithm for traversing or searching tree

or graph data structures. The idea is to follow one branch of the tree down as many levels as possible until the end is reached,

Intuition

When dealing with permutations, we can imagine each permutation as a path in a decision tree, where each level represents an element in the permutation and each branch represents a choice of which element to place next.

Here's how we can visualize this approach:

1. Start with an empty permutation.

2. For every index in the permutation, try placing each unused element (one that has not been used in this particular path/branch).

and then proceed to the next branch.

3. After an element is placed at the current index, mark it as used and move to the next index. 4. Once an index is placed with all possible elements, backtrack (step back) and try the next possible element for the previous index.

backtrack at the appropriate times, hence constructing all unique permutations.

to the index in the permutation where we are placing the next element.

- 5. Repeat until all elements are placed in the permutation, meaning we've reached the end of the branch. This permutation is then added to the list of results.
- 6. When all branches are explored, we will have found all the permutations. The dfs() function is a recursive method that implements this approach. It uses the index i to keep track of the depth of
- recursion (which corresponds to the current position in the permutation). The vis array helps track which elements have been used, and the t array represents the current permutation being constructed. Once index i reaches n, the length of nums, it means a full permutation has been created and it's added to the answer ans.

Solution Approach The solution uses a classic Depth-First Search (DFS) algorithm implemented through recursion to explore all potential

permutations in a systematic way. The key to the DFS algorithm in this context is to treat each level of recursion as a position in

To sum up, the problem is solved by systematically exploring all possible placements of elements using DFS, while making sure to

the permutation, and to attempt to fill that position with each possible unused element from the nums array. Here's a step-by-step breakdown of the algorithm:

DFS. At first, all values are False indicating no elements have been used yet. Create a temporary array t of length n. This array will hold the current permutation as it is being built.

Initialize an n-length boolean array vis to keep track of which elements from nums have been used in the current branch of the

An empty list ans is initialized to store all the completed permutations. The dfs function is defined to perform the DFS. It takes an argument i which is the current depth of the DFS, corresponding

- In the dfs function, the base condition checks if i is equal to n. If true, it means we've reached the end of a branch in the DFS and a full permutation has been constructed, so we append a copy of this permutation to ans.

If the base condition is not met, the function proceeds to iterate over all elements of nums.

• For each element, if it has not already been used (i.e., vis[j] is False), we mark it as used by setting vis[j] to True. • We then place this element in the i-th position of the current permutation being built t[i].

This approach uses a combination of recursive DFS for the traversal mechanism, backtracking for generating permutations

without duplicates, and dynamic data structures to keep track of the used elements and current permutations. The elegance of

The initial call to dfs is made with an argument of 0 to start the DFS from the first position in the permutation.

- Call dfs(i + 1) to proceed to the next level of depth, attempting to find an element for the next position. After returning from the deeper recursive call, we reset vis[j] to False to "unchoose" the element, thus enabling it to be used in a different branch or permutation. This is the backtracking step.
- this solution lies in how it natively handles the permutations' creation without redundancy, which is essential in problems where the order of elements matters and you want to consider all possible orderings.
- **Example Walkthrough** Let's illustrate the solution approach using a simple example where our array nums is [1,2,3]. We want to find all permutations of

2, set vis[1] to True, and place 2 in t[1]. a. We call dfs(i=2).

False, True]. We call dfs(i=2) to decide the third element.

def permute(self, nums: List[int]) -> List[List[int]]:

we have a complete permutation

Helper function to perform depth-first search for permutation generation

Check if the number at index j is already used in the current permutation

visited = [False] * len_nums # Create a visited list to track numbers that are used

 $current_permutation = [0] * len_nums # Temp list to store the current permutation$

Backtrack: unmark the number at index j as visited for the next iteration

If the current index has reached the length of nums list,

permutations.append(current_permutation[:])

Iterate over the nums list to create permutations

len_nums = len(nums) # Store the length of the input list

backtrack(0) # Start generating permutations from index 0

permutations = [] # Result list to store all the permutations

Finally, the ans list is returned which now contains all possible permutations.

We initialize a boolean array vis of length 3 (n=3 in this case), all set to False, which will help us keep track of the elements that have been used in the current permutation. We also create a temporary array t to build the current permutation and an empty list ans to store all permutations. vis = [False, False, False] t = [0, 0, 0] ans = []

Our for loop in the dfs function will consider each element in nums: a. On the first iteration, we choose 1. We set vis [0] to

The next level of recursion is to place the third element. 1 and 2 are marked as used, so we pick 3, set vis [2] to True, and

Now, i equals to n. We've reached the end of the branch and have a complete permutation [1,2,3], which we add to ans.

Now we are in a new level of recursion, trying to fill t[1]. Again, we traverse nums. We skip 1 because vis[0] is True. We pick

ans = [

[1, 2, 3],

[1, 3, 2],

[2, 1, 3],

class Solution:

def backtrack(index):

return

return permutations

if index == len_nums:

for j in range(len_nums):

if not visited[j]:

backtrack(index + 1)

visited[j] = False

this array.

place 3 in t[2].

Start the DFS with dfs(i=0). At this level of recursion, we are looking to fill in the first position of the t array.

True and place 1 in t[0]. b. We call dfs(i=1) to decide on the second element of the permutation.

We backtrack by returning to where we picked 3. We unmark vis[2] (vis[2]=False), trying to explore other possibilities for this position, but there are no more elements left to use. So, we backtrack further.

Back at the second element decision step (dfs(i=1)), we backtrack off of element 2 and pick 3 for t[1]. vis is now [True,

In this call, 2 is the only unused element, so we put it in t[2], making the permutation [1,3,2]. We add this to ans.

This recursive process continues, systematically exploring each possible permutation and backtracking after exploring each

- branch to the fullest extent. This ensures that we explore all permutations without duplication. In the end, ans would be:
- [2, 3, 1], [3, 1, 2], [3, 2, 1]

Solution Implementation **Python**

This walk through represents how the algorithm builds up permutations and how it builds the result step by step.

```
# If not visited, mark it as visited and add to current permutation
visited[j] = True
current_permutation[index] = nums[j]
# Recurse with next index
```

Java

class Solution {

#include <vector>

class Solution {

};

TypeScript

};

public:

#include <functional> // For the std::function

// Function to generate all permutations of the input vector of integers.

std::vector<std::vector<int>>> permutations; // To store all permutations.

std::vector<int> current_permutation(n); // Current permutation vector.

std::vector<bool> visited(n, false); // Visited flags for nums elements.

// Recursive Depth-First Search (DFS) function to generate permutations.

for (int i = 0; i < n; ++i) { // Iterate through nums elements.

dfs(depth + 1); // Recurse with the next depth.

visited[i] = true; // Mark as visited.

return permutations; // Return all the generated permutations.

if (depth == n) { // Base case: if the current permutation is complete.

if (!visited[i]) { // If the ith element has not been visited.

visited[i] = false; // Unmark as visited for backtracking.

permutations.emplace_back(current_permutation); // Add to permutations.

current_permutation[depth] = nums[i]; // Set in current permutation.

std::vector<std::vector<int>> permute(std::vector<int>& nums) {

std::function<void(int)> dfs = [&](int depth) {

return; // End of branch.

dfs(0); // Start the DFS with depth 0.

// Function to generate all permutations of an array of numbers

int n = nums.size(); // Get the size of the nums vector.

```
// List to hold all the permutations
    private List<List<Integer>> permutations = new ArrayList<>();
   // Temporary list to hold the current permutation
   private List<Integer> currentPermutation = new ArrayList<>();
   // Visited array to keep track of the elements already included in the permutation
   private boolean[] visited;
   // Array of numbers to create permutations from
   private int[] elements;
   // Method to initiate the process of finding all permutations
    public List<List<Integer>> permute(int[] nums) {
       elements = nums;
       visited = new boolean[nums.length];
       backtrack(0);
       return permutations;
   // Helper method to perform backtracking
   private void backtrack(int index) {
       // Base case: if the permutation size is equal to the number of elements, add it to the answer
       if (index == elements.length) {
           permutations.add(new ArrayList<>(currentPermutation));
           return;
       // Iterate through the elements array
       for (int j = 0; j < elements.length; ++j) {</pre>
           // If the element at index j has not been visited, include it in the permutation
           if (!visited[j]) {
               // Mark the element at index j as visited
               visited[j] = true;
                // Add the element to the current permutation
                currentPermutation.add(elements[j]);
                // Continue to the next level of depth (next index)
               backtrack(index + 1);
               // Backtrack: remove the last element added and mark it as not visited
                currentPermutation.remove(currentPermutation.size() - 1);
               visited[j] = false;
C++
```

```
const n = nums.length; // Length of the array to permute
const results: number[][] = []; // Results array that will hold all permutations
```

function permute(nums: number[]): number[][] {

```
// Helper function 'depthFirstSearch' to explore the permutations using DFS strategy
      const depthFirstSearch = (currentIndex: number) => {
          // If the current index reaches the end of array, record the permutation
          if (currentIndex === n) {
              results.push([...nums]); // Add a copy of the current permutation
              return;
          // Iterate over the array to swap each element with the element at 'currentIndex'
          for (let swapIndex = currentIndex; swapIndex < n; swapIndex++) {</pre>
              // Swap the elements
              [nums[currentIndex], nums[swapIndex]] = [nums[swapIndex], nums[currentIndex]];
              // Recursively call 'depthFirstSearch' with the next index
              depthFirstSearch(currentIndex + 1);
              // Swap back the elements to revert to the original array before the next iteration
              [nums[currentIndex], nums[swapIndex]] = [nums[swapIndex], nums[currentIndex]];
      };
      // Initiate depth-first search starting from index 0
      depthFirstSearch(0);
      // Return all generated permutations
      return results;
class Solution:
   def permute(self, nums: List[int]) -> List[List[int]]:
       # Helper function to perform depth-first search for permutation generation
        def backtrack(index):
           # If the current index has reached the length of nums list,
           # we have a complete permutation
            if index == len nums:
                permutations.append(current_permutation[:])
                return
           # Iterate over the nums list to create permutations
            for j in range(len_nums):
               # Check if the number at index j is already used in the current permutation
               if not visited[j]:
                    # If not visited, mark it as visited and add to current permutation
                    visited[j] = True
                    current_permutation[index] = nums[j]
                   # Recurse with next index
                    backtrack(index + 1)
                   # Backtrack: unmark the number at index j as visited for the next iteration
                    visited[j] = False
        len_nums = len(nums) # Store the length of the input list
        visited = [False] * len_nums # Create a visited list to track numbers that are used
        current_permutation = [0] * len_nums # Temp list to store the current permutation
```

The time complexity of the algorithm is determined by the number of recursive calls made, and the work done in each call. The function dfs is called recursively until it reaches the base case (i == n).

Time Complexity

return permutations

Time and Space Complexity

permutations = [] # Result list to store all the permutations

backtrack(0) # Start generating permutations from index 0

For n distinct elements, there are n! (factorial of n) permutations. At each level of the recursion, we make n choices, then n - 1 for the next level, and so on, which means we are doing n! work as there are that many permutations to generate and for each of them we do O(1) operation. Hence, the time complexity is O(n!).

The provided Python code generates all permutations of a list of integers, using a backtracking algorithm.

Space Complexity The space complexity consists of the space used by the recursive call stack and the space used to maintain the state (visited

1. Recursive Call Stack: Since the depth of the recursion is n, at most 0(n) functions will be placed on the call stack simultaneously.

0(n).

array vis and temporary list t).

2. State Maintenance: The list vis and t require O(n) space each. The total space complexity, therefore, is O(n) + O(n) * O(2) = O(n). However, since O(2) is a constant factor, it simplifies to

Taking all this into account, the space complexity is O(n) for maintaining the auxiliary data structure and the recursive call stack depth.