47. Permutations II

Medium Array Backtracking

Problem Description

The problem requires us to generate all possible unique permutations of a given collection of numbers `nums`. The `nums` array may contain duplicates, which adds a layer of complexity because we have to ensure that our permutations are unique and do not include the same sets of numbers more than once. The challenge is to come up with a way that efficiently explores all potential combinations without revisiting the same permutations.

Intuition

The intuition behind the solution is to use Depth-First Search (DFS) in combination with backtracking to explore all possible orderings of numbers. However, since we have potential duplicates in the array, we have to be very cautious about how we recurse.

Firstly, sorting the `nums` array helps bring duplicate elements next to each other, which is crucial for our later steps to detect and skip duplicates efficiently.

Then, we use a `vis` array of boolean values to keep track of which elements have been used in the current permutation. This is to avoid reusing elements unintentionally, as each number must appear exactly once in any permutation.

While we are constructing permutations, we need to handle the possibility of duplicates. We incorporate a condition to skip over a number if it is the same as the previous number and the previous number has not been included in the current permutation. This check ensures that we are not generating any duplicate permutations because it prevents the algorithm from picking the same element twice when the elements are identical and the previous one is unused.

By using recursive DFS, we explore all paths in the search space that lead to valid unique permutations. As we hit the base case where the depth (`i`) equals the length of the numbers array (`n`), we've successfully built a valid permutation and we append a copy of the current permutation (represented by array `t`) to our results list (`ans`).

This approach ensures that each permutation in the results list `ans` is unique and contains all elements from the `nums` array exactly once.

Solution Approach

The solution approach is based on Depth-First Search (DFS) and backtracking. The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead if possible, else by backtracking.

Here is how we implement the solution:

1. **Sort the Array**: First, we sort the `nums` array. Sorting is crucial because it makes the detection of duplicates easy by placing them next to each other.
2. **Prepare Helper Structures**: We create a helper array `t` to store the current permutation and a `vis` array to keep track of whether an element at a given index has been used in the current permutation or not.
3. **Define a DFS Helper Function**: We define a recursive function `dfs(i)` where `i` is the current index in the `t` array that we are trying to fill. This function will try to fill `t[i]` with every possible number from the `nums` array that hasn't been used yet (as indicated by the `vis` array).
4. **Recursion and Backtracking**: The `dfs` function iterates through `nums`. For each number at index `j`:
   - If `vis[j]` is `True` (the number has been used already), skip it.
   - If `nums[j]` is equal to `nums[j - 1]` and `vis[j - 1]` is `False` (the number is a duplicate and the previous occurrence hasn't been used yet), skip it to prevent a duplicate permutation.
   - Otherwise, choose `nums[j]` by setting `t[i]` to `nums[j]`, marking `vis[j]` as `True`, and recursively calling `dfs(i + 1)`.
5. **Save and Reset on Backtracking**: Whenever we reach the base case of `dfs`, which is when `i == n`, it means we have filled up the `t` array with a valid permutation. We add a copy of `t` to the answer list `ans`. Then, we backtrack by resetting the `vis[j]` to `False`, essentially marking the number at index `j` as unused and available for future permutations.
6. **Invoke and Return**: We kickstart the DFS by calling `dfs(0)`. After the recursive calls are done, all unique permutations are stored in `ans`, which we then return.

By following this approach, we effectively avoid constructing duplicate permutations and generate all unique permutations in an efficient manner.

Example Walkthrough

Let's walk through an example where `nums = [1, 1, 2]` to illustrate this approach.

1. **Sort the Array**: First of all, we sort the array `nums`. The array is already sorted `[1, 1, 2]` so no changes are made. Sorting is important to identify duplicates.
2. **Prepare Helper Structures**:
   - We create an array `t` to store the current permutation sequence, which is initially empty.
   - An array `vis` is created to keep track if an element has been added to the current permutation. Initially, `vis = [False, False, False]` because no numbers have been used yet.
3. **Define a DFS Helper Function**: We define a recursive function `dfs(i)`, where `i` denotes the number of elements in the current permutation. This function attempts to generate all unique permutations by trying to fill `t` with elements from `nums`.
4. **Recursion and Backtracking**:
   - Start the first recusive call `dfs(0)`. Here, we attempt to pick the first element (`i=0`) for our permutation `t`.
   - The `dfs` function iterates through the indices of `nums` (`[0, 1, 2]` in our example).
   - For each `j` in `[0, 1, 2]`:
     - On the first iteration `j=0`, since `vis[0]` is `False`, we can pick `nums[0]` to be part of the permutation. So `t[0]` becomes `1`, and `vis[0]` becomes `True`. We call `dfs(1)` to pick the next element for `t`.
     - Inside `dfs(1)`, we cannot pick `nums[1]` because it would be a duplicate (`nums[1]` is equal to `nums[0]` and `vis[0]` is now `True`). We skip and proceed to `j=2`.
     - We pick `nums[2]`, so `t[1]` becomes `2`, and `vis[2]` becomes `True`. We call `dfs(2)` to pick the last element for `t`.
     - Inside `dfs(2)`, we only have one choice left, which is `nums[1]` since `vis[1]` is `False`. We set `t[2]` to `1`, and now `t = [1, 2, 1]`. We reached the base case because `i` equals `n (3)` – the length of `nums`. We add `[1, 2, 1]` to `ans`.
     - Now we backtrack. We reset `vis[1]` to `False` and go up to `dfs(1)`.
     - In `dfs(1)`, we backtrack again by resetting `vis[2]` to `False` and return to `dfs(0)`.
     - Now `j=1` is the current index in `dfs(0)`, and since `nums[1]` is a duplicate with an unused previous element (`nums[0]`), it is skipped.
     - We proceed with `j=2`, by setting `t[0]` to `2`, `vis[2]` to `True`, and call `dfs(1)`.
     - In `dfs(1)`, now we can use `nums[0]` and `nums[1]` because `nums[0]` is not a duplicate in the current context of `t`. We create permutations `[2, 1, 1]` in a similar way and add them to `ans`.
5. **Save and Reset on Backtracking**: Each time we hit the base case (`i == n`), we have a complete permutation to add to `ans`. We then backtrack, undoing the last step in our permutation to free up elements for new permutations.
6. **Invoke and Return**: We start by invoking `dfs(0)`, and after all the recursive calls and backtracks, we get `ans = [[1, 1, 2], [1, 2, 1], [2, 1, 1]]`, which are all the unique permutations of `nums`.

By following this ordered approach, we have now generated all valid unique permutations of `nums` by ensuring we don't produce any repetition arising from duplicates.

Python Solution

```python
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        # Helper function to perform depth-first search (DFS) to find unique permutations
        def backtrack(index: int):
            # If the current index is equal to the size of nums, we have a complete permutation to add to the answer
            if index == size:
                permutations.append(current_permutation[:])
                return

            # Iterate through each number trying to construct the next permutation
            for j in range(size):
                # Skip this number if it has been used already or if it's a duplicate and its previous instance was not used
                if visited[j] or (j > 0 and nums[j] == nums[j - 1] and not visited[j - 1]):
                    continue

                # Choose the number nums[j] and mark it as visited
                current_permutation[index] = nums[j]
                visited[j] = True

                # Recur to construct the next index's permutation
                backtrack(index + 1)

                # Unchoose the number nums[j] and mark it as unvisited for further iterations
                visited[j] = False

        size = len(nums)
        nums.sort()  # Sort nums to handle duplicates
        permutations = []  # This will hold all unique permutations
        current_permutation = [0] * size  # Temporary list to hold a single permutation
        visited = [False] * size  # List to keep track of visited indices in nums

        # Start the DFS from index 0
        backtrack(0)

        return permutations  # Return all the collected permutations

```

Java Solution

```java
class Solution {
    // List to store all unique permutations
    private List<List<Integer>> permutations = new ArrayList<>();
    // Temporary list to store one permutation
    private List<Integer> tempPermutation = new ArrayList<>();
    // Array of numbers to create permutations from
    private int[] numbers;
    // Visited flags to track whether a number has been used in the current permutation
    private boolean[] visited;

    public List<List<Integer>> permuteUnique(int[] nums) {
        // Sort the numbers to ensure duplicates are adjacent
        Arrays.sort(nums);
        // Initialize class variables
        this.numbers = nums;
        visited = new boolean[nums.length];
        // Start the depth-first search from the first index
        dfs(0);
        // Return the list of all unique permutations found
        return permutations;
    }

    private void dfs(int index) {
        // Base case: If the current permutation is complete, add it to the list of permutations
        if (index == numbers.length) {
            permutations.add(new ArrayList<>(tempPermutation));
            return;
        }
        // Iterate over the numbers to build all possible permutations
        for (int i = 0; i < numbers.length; ++i) {
            // Skip the current number if it's already been used or if it's a duplicate and the duplicate hasn't been used
            if (visited[i] || (i > 0 && numbers[i] == numbers[i - 1] && !visited[i - 1])) {
                continue;
            }
            // Add the current number to the current permutation and mark it as visited
            tempPermutation.add(numbers[i]);
            visited[i] = true;
            // Recursively continue building the permutation
            dfs(index + 1);
            // Backtrack by removing the current number and unmarking it as visited
            visited[i] = false;
            tempPermutation.remove(tempPermutation.size() - 1);
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to generate all unique permutations of vector 'nums'.
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        // First, sort the array to handle duplicates.
        sort(nums.begin(), nums.end());
        // Get the size of the nums vector.
        int size = nums.size();
        // This will hold all the unique permutations.
        vector<vector<int>> permutations;
        // Temporary vector to hold current permutation.
        vector<int> current(size);
        // Visited array to keep track of used elements.
        vector<bool> visited(size, false);

        // Recursive lambda function to perform Depth-First Search.
        function<void(int)> dfs = [&](int depth) {
            // If the current permutation is complete, add to permutations.
            if (depth == size) {
                permutations.emplace_back(current);
                return;
            }
            // Iterate over all elements in 'nums'.
            for (int i = 0; i < size; ++i) {
                // Skip already visited elements or duplicates not in sequence.
                if (visited[i] || (i > 0 && nums[i] == nums[i - 1] && !visited[i - 1])) {
                    continue;
                }
                // Place nums[i] in the current position.
                current[depth] = nums[i];
                // Mark this element as visited.
                visited[i] = true;
                // Recurse with next position.
                dfs(depth + 1);
                // Reset visited status for backtracking.
                visited[i] = false;
            }
        };
        // Start the recursive process with the first position.
        dfs(0);
        // Return the resulting permutations.
        return permutations;
    }
};

```

Typescript Solution

```typescript
// Given an array of numbers, this function returns all unique permutations
function permuteUnique(nums: number[]): number[][] {
    // Sort the input array
    nums.sort((a, b) => a - b);

    // Define the length of the nums array
    const length = nums.length;

    // This array will store all the unique permutations
    const results: number[][] = [];

    // Temporary array to store one permutation at a time
    const permutation: number[] = new Array(length);

    // Visited array to keep track of which elements are used
    const visited: boolean[] = new Array(length).fill(false);

    // Helper function to generate permutations using DFS (Depth-First Search)
    const dfs = (index: number) => {
        // If the temporary array is filled, add a copy to results
        if (index === length) {
            results.push([...permutation]);
            return;
        }

        for (let j = 0; j < length; ++j) {
            // Skip already visited elements or duplicates (to ensure uniqueness)
            if (visited[j] || (j > 0 && nums[j] === nums[j - 1] && !visited[j - 1])) {
                continue;
            }

            // Choose the element and mark as visited
            permutation[index] = nums[j];
            visited[j] = true;

            // Continue building the permutation
            dfs(index + 1);

            // Backtrack: unmark the element after recursive call returns
            visited[j] = false;
        }
    };

    // Start the DFS traversal from the first index
    dfs(0);

    // Return all the unique permutations
    return results;
}

```

Time and Space Complexity

The given Python code implements a backtracking algorithm to generate all unique permutations of a list of numbers.

Time Complexity

The time complexity of the algorithm is mainly influenced by the number of recursive calls (`dfs` function) made to construct the permutations. The sorting operation at the start has a time complexity of `O(N log N)`, where `N` is the number of elements in `nums`.

In the worst case (when all elements are unique), the number of unique permutations is `N!` (factorial of `N`). However, due to the branch pruning by checking `vis[j]` and the uniqueness condition `(j and nums[j] == nums[j - 1] and not vis[j - 1])`, the actual number of permutations explored is less than `N!`. This optimization is significant especially when `nums` contains many duplicates.

However, it's hard to define a precise time complexity in the presence of these optimizations without knowing the distribution of numbers in `nums`. In the worst case, we can consider the complexity to be `O(N!*N)`, as for each permutation, there is an `O(N)` check due to the uniqueness conditions and assigning values to `t`.

Space Complexity

The space complexity is determined by the amount of memory used to store the temporary arrays and the recursion stack.

- `ans` array which can potentially store `N!` permutations, each of size `N` in the case of all unique elements, so `O(N!*N)` space complexity for storing the output.
- Temporary array `t` of size `N`, and `vis` array also of size `N`, give `O(N)`.
- The maximum depth of the recursion stack is `N`, leading to `O(N)` space complexity.

Therefore, the total space complexity would be `O(N!*N)` due to the space required to store the output `ans`. If we don't count the space required for the output, the algorithm still uses `O(N)` space for the `t` and `vis` arrays, plus `O(N)` space for the recursion stack, leading to `O(N)` auxiliary space complexity.