90. Subsets II

Medium Bit Manipulation Array Backtracking

Problem Description

The task is to return all possible subsets (also known as the power set) of a given integer array `nums`, which may contain duplicates. The important constraint is that the solution set must not include duplicate subsets. These subsets can be returned in any order. A subset is a set containing elements that are all found in another set, which in this case is the array `nums`. The power set is the set of all subsets including the empty set and the set itself.

For example, if `nums` is `[1,2,2]`, the possible unique subsets without considering order are:

```
- []
- [1]
- [2]
- [1,2]
- [2,2]
- [1,2,2]
```

The challenge here is to ensure that while generating the subsets, duplicates are not created.

Intuition

The core idea to avoid duplicates in the subsets is to sort the array `nums`. Sorting brings identical elements next to each other, making it easier to avoid duplicates when constructing the subsets.

The solution uses Depth-First Search (DFS) to explore all the possible subsets. The process starts with an empty subset, and at each level, it includes the next number from the array into the current subset, then recursively continues to add the remaining numbers to form new subsets. This is done until all numbers have been considered. After each number is processed for a given subset, it is removed again (this operation is also known as backtracking), and the next number is tried.

To ensure no duplicated subsets are generated, there are two conditions:

1. **Sort the Numbers**: As mentioned, we start by sorting the array to bring duplicates together.
2. **Skip over duplicates**: While generating subsets, if the current element is the same as the previous element and the previous element was not included in the current subset being generated (checked by `i != u`), it is skipped to avoid creating a subset that has already been created.

This way, the `dfs` function systematically constructs all subsets, but avoids any repeats, ensuring that the power set it returns is free from duplicate subsets.

Solution Approach

The implementation of the solution uses recursion to generate the subsets and backtracking to make sure all possible subsets are considered. Here's how the implementation works, with reference to the given Python code above:

1. **Sort the Array**: The first thing the solution does is to sort `nums`. This is crucial because it ensures that duplicates are adjacent and can be easily skipped when generating subsets.

   ```
   nums.sort()
   ```

2. **Depth-First Search (DFS)**: The `dfs` function is then defined. DFS is a common algorithm for exploring all the possible configurations of a given problem. In this case, it's used to generate all possible subsets by making a series of choices (to include or not include an element in the current subset).

3. **Recursive Exploration**: Inside `dfs`, the current subset `t` is first included in the answer set `ans`. This represents the choice of not including any more elements in the current subset.

   ```
   ans.append(t[:])
   ```

4. **Iterative Inclusion**: The function then iterates through the remaining numbers starting from index `u`. If the number at the current index is the same as the one before it and it's not the first iteration of the loop (checked by `i != u`), it skips adding this number to avoid duplicates.

   ```python
   for i in range(u, len(nums)):
       if i != u and nums[i] == nums[i - 1]:
           continue
       t.append(nums[i])
       dfs(i + 1, t)
       t.pop()
   ```

   - `t.append(nums[i])` adds the current number to the subset.
   - A recursive call `dfs(i + 1, t)` is made to consider the next number in the array. The index `i + 1` ensures that the function does not consider the current number again, thus moving forward in the array.
   - `t.pop()` is the backtracking step, which removes the last added number from the subset, allowing the function to consider other possibilities after returning from the recursive call.

5. **Global Answer Set**: An empty list `ans` is initialized outside the `dfs` function. As the subsets are created, they are added to `ans`. Since `ans` is defined outside the scope of the `dfs` function, it retains all the solutions across recursive calls.

   ```
   ans = []
   ```

6. **Kick Start the DFS**: The `dfs` function is initially called with the starting index `0` and an empty subset `[]` to start the exploration.

   ```
   dfs(0, [])
   ```

At the end of the execution, the global `ans` contains all the subsets without duplicates, which is then returned as the final result.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using the array `[2,1,2]`.

1. **Sort the Array**: First, sort the array to become `[1,2,2]`. Sorting ensures any duplicates are next to each other which helps later in the process to avoid duplicate subsets.
2. **Depth-First Search (DFS)**: We define a `dfs` function that will be used to explore possible subsets.
3. **Recursive Exploration**: The exploration starts with an empty subset `[]`. This subset is immediately added to the answer set `ans` because it is a valid subset (the empty set).
4. **Iterative Inclusion**: For the first call to `dfs`, we will look at each element starting with the first element of the sorted array.
   - Include the first element `[1]`. At this stage, `dfs` is recursively called to consider the next number.
   - At the next level, we have `[1,2]`. We include it and then the `dfs` explores further.
   - Now we have `[1,2,2]`, which is also included.
   - After considering each of these, we backtrack to `[1,2]` and remove the last element to reconsider our options.
   - Since the next element is the same as the previous (as we just removed it), the loop skips adding `[1,2]` again because it would be a duplicate.
5. **Backtracking**: After the above steps, we backtrack to an empty subset `[]` and proceed to the next element:
   - The next element is `[2]`. We include it, and then `dfs` is called recursively.
   - At the next level, the function sees another `2`. However, since the previous element is also `2`, and we are not at the start of a new subset creation (checked by `i != u`), the algorithm takes care not to include `[2,2]` as this is already considered. This is where the duplicate gets skipped.
6. **Global Answer Set**: All the while, subsets like `[]`, `[1]`, `[1,2]`, `[1,2,2]`, and `[2]` are being added to the global `ans` list.
7. **Complete Exploration**: After the function explores all possibilities, the `ans` now contains, without duplicates:

```
- []
- [1]
- [2]
- [1,2]
- [1,2,2]
```

8. **Return Results**: Finally, the `dfs` has finished exploring, and `ans` with all the unique subsets is returned as the result.

This example demonstrates how the algorithm ensures all unique subsets of `[2,1,2]` are found by methodically exploring all options while skipping over duplicates due to sorting and the extra conditional skip in the recursive loop.

Python Solution

```python
from typing import List

class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        # Helper function to perform depth-first search
        def depth_first_search(start_index, current_subset):
            # Append a copy of the current subset to the answer list
            result.append(current_subset[:])

            # Iterate over the numbers starting from the current index
            for index in range(start_index, len(nums)):
                # Skip duplicates except for the start of a new subsection
                if index != start_index and nums[index] == nums[index - 1]:
                    continue
                # Include the current number and recurse
                current_subset.append(nums[index])
                depth_first_search(index + 1, current_subset)
                # Backtrack by removing the last number added
                current_subset.pop()

        # This list will store all the subsets
        result = []
        # First sort the input array to handle duplicates
        nums.sort()
        # Start the recursive process with an empty subset
        depth_first_search(0, [])
        # Return the generated subsets
        return result

# Example usage:
# sol = Solution()
# print(sol.subsetsWithDup([1, 2, 2]))  # Output: [[], [1], [1, 2], [1, 2, 2], [2], [2, 2]]
```

Java Solution

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Solution {
    // List to store the final subsets
    private List<List<Integer>> subsets;

    // The provided array of numbers, from which we will form subsets
    private int[] numbers;

    // Public method to find all subsets with duplicates
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        subsets = new ArrayList<>(); // Initialize the subsets list
        Arrays.sort(nums); // Sort the array to handle duplicates
        this.numbers = nums; // Store the sorted array in the numbers variable
        backtrack(0, new ArrayList<>()); // Start the backtrack algorithm
        return subsets; // Return the list of subsets
    }

    // Helper method to perform the backtrack algorithm
    private void backtrack(int index, List<Integer> currentSubset) {
        // Add a new subset to the answer list, which is a copy of the current subset
        subsets.add(new ArrayList<>(currentSubset));

        // Iterate through the numbers starting from index
        for (int i = index; i < numbers.length; ++i) {
            // Skip duplicates: check if the current element is the same as the previous one
            if (i != index && numbers[i] == numbers[i - 1]) {
                continue; // If it's a duplicate, skip it
            }

            // Include the current element in the subset
            currentSubset.add(numbers[i]);

            // Recursively call backtrack for the next elements
            backtrack(i + 1, currentSubset);

            // Exclude the current element from the subset (backtrack step)
            currentSubset.remove(currentSubset.size() - 1);
        }
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function to generate all possible subsets with duplicates
    std::vector<std::vector<int>> subsetsWithDup(std::vector<int>& nums) {
        // Sort the numbers to handle duplicates easily
        std::sort(nums.begin(), nums.end());

        // This will be our result list of subsets
        std::vector<std::vector<int>> subsets;

        // Temporary list to build each subset
        std::vector<int> tempSubset;

        // Start the depth-first search from the first index
        depthFirstSearch(0, tempSubset, nums, subsets);

        return subsets;
    }

private:
    // Recursive DFS function to explore all subsets
    // 'index' is the current position in 'nums' array
    void depthFirstSearch(int index, std::vector<int>& tempSubset, std::vector<int>& nums, std::vector<std::vector<int>>& subsets) {
        // Add the current subset to the list of subsets
        subsets.push_back(tempSubset);

        // Iterate through the array starting from the 'index'
        for (int i = index; i < nums.size(); ++i) {
            // Skip the current number if it's the same as the previous one at the same tree depth
            if (i != index && nums[i] == nums[i - 1]) continue;

            // Include the current number in the subset
            tempSubset.push_back(nums[i]);

            // Continue searching further with the current number included
            depthFirstSearch(i + 1, tempSubset, nums, subsets);

            // Exclude the current number before going to the next iteration
            tempSubset.pop_back();
        }
    }
};
```

Typescript Solution

```typescript
function subsetsWithDup(nums: number[]): number[][] {
    // Sort the array to handle duplicates.
    nums.sort((a, b) => a - b);
    const length = nums.length;
    // Temporary array to hold intermediate solutions.
    const temp: number[] = [];
    // Resulting array of subsets.
    const result: number[][] = [];
    // Depth-first search function to explore subsets.
    const backtrack = (index: number) => {
        // If the end of the array is reached, add the current subset to the result.
        if (index === length) {
            result.push([...temp]);
            return;
        }
        // Include the current number in the subset.
        temp.push(nums[index]);
        backtrack(index + 1); // Move to the next number.

        // Remove the current number and skip all duplicates to prevent duplicate subsets.
        const lastNumber = temp.pop();
        while (index < length && nums[index] === lastNumber) {
            index++;
        }
        backtrack(index);
    };

    // Start the backtracking with the first index of the array.
    backtrack(0);
    // Return all the subsets when done.
    return result;
}
```

Time and Space Complexity

The given Python code defines a method `subsetsWithDup` that finds all possible subsets of a given set of numbers that might contain duplicates. To avoid duplicate subsets, the algorithm sorts the array and skips over duplicates during the depth-first search (DFS).

Time Complexity:

The time complexity of this algorithm can be analyzed based on the number of recursive calls and the operations performed on each call.

1. The function `dfs` is called recursively, at most, once for every element starting from the current index `u`. In the worst case, this means potentially calling `dfs` for each subset of `nums`.
2. Since `nums` has `n` elements, there are `2^n` possible subsets including the empty subset.
3. However, because the list is sorted and the algorithm skips duplicate elements within the same recursive level, the number of recursive calls may be less than `2^n`.
4. The `append` and `pop` methods of a list run in `O(1)` time.

Therefore, the time complexity of the algorithm is essentially bounded by the number of recursive calls and is `O(2^n)` in the worst case, when all elements are unique.

Space Complexity:

The space complexity is considered based on the space used by the recursion stack and the space needed to store the subsets (`ans`).

1. The maximum depth of the recursive call stack is `n`, which is the length of `nums`. Each recursive call uses space for local variables, which is `O(n)` space in the stack at most.
2. The list `ans` will eventually contain all subsets, and thus, it will have `2^n` elements, considering each subset as an element. However, the total space taken by all subsets combined is also considerable as each of the `2^n` subsets could have up to `n` elements. This effectively adds up to `O(n * 2^n)` space.

Considering these factors, the space complexity of the code can be defined as `O(n * 2^n)` because the space used by the algorithm is proportional to the number of subsets generated, and each subset at most can have `n` elements.