78. Subsets

Medium Bit Manipulation Array Backtracking

Leetcode Link

Problem Description

The LeetCode problem asks us to generate all the possible subsets from a given list of unique integers. The term 'subset' refers to any combination of numbers that can be formed from the original list, including the empty set and the set itself. For instance, if the input is `[1,2,3]`, then the subsets would include `[]`, `[1]`, `[2]`, `[3]`, `[1,2]`, `[1,3]`, `[2,3]`, and `[1,2,3]`. The objective is to list down all these possibilities. We should also ensure that there are no duplicate subsets in the solution and the subsets can be returned in any order.

Intuition

To solve the problem, we are using a concept known as Depth-First Search (DFS). The process involves exploring each element and deciding whether to include it in the current subset (`t`). At each step, we have the choice to either include or not include the current element in our subset before moving to the next element.

For example, if our set is `[1, 2, 3]`, the DFS approach starts at the first element (`1` then `2` then `3`). For each element, we take two paths: in one path we include the element into the subset, and in the other, we do not.

Here's what the decision tree would look like for `[1, 2, 3]`:

1. Start with an empty subset `[]`.
2. Choose whether to include `1` or not, resulting in subsets `[]` and `[1]`.
3. For each of these subsets, choose whether to include `2`, leading to `[]`, `[1]`, `[2]`, and `[1, 2]`.
4. Finally, for each of these, choose whether to include `3`, ending with `[]`, `[1]`, `[2]`, `[1, 2]`, `[3]`, `[1, 3]`, `[2, 3]`, and `[1, 2, 3]`.

At the end of this process, we have explored all possible combinations, and our `ans` list contains all of them. Each recursive call represents a decision, and by exploring each decision, we exhaust all possibilities and build the power set.

Solution Approach

The reference solution provided is a recursive approach using Depth-First Search (DFS) to build up all possible subsets. Here's a step-by-step explanation of how the code implements this algorithm:

- We define a helper function called `dfs` with the parameter `i`, which represents the current index in the `nums` array we are considering.
- The base case for the recursion is when `i` equals the length of `nums`. At this point, we know we've considered every element. We make a copy of the current subset `t` and append it to our answer list `ans`.
- If we haven't reached the end of the array, we have two choices: we can either include the current element or not. First, we choose not to include it by simply calling `dfs(i + 1)` without modifying `t`.
- After returning from the above call, we then decide to include the current element by appending `nums[i]` to `t`. Then we call `dfs(i + 1)` again to continue exploring further with the element included.
- After the second recursive call returns, we need to backtrack, which means removing the last element added to `t`, essentially undoing the decision to include the current element. This is done by calling `t.pop()`.
- The initial call to `dfs` starts with index `0`, indicating that we start exploring from the first element of the array.
- The algorithm uses a backtracking pattern, which is evident in the decision to include or not include an element and the subsequent reversal of that decision.
- The use of recursion and backtracking allows us to explore all possible combinations of elements, resulting in the power set of `nums`. This is because at each step we go as deep as we can into one combination (depth-first), and then we backtrack to explore a new one.
- The code avoids the formation of duplicate subsets by relying on the fact that `nums` contains unique elements and by systematically exploring all possible combinations without repeating them.

The data structure used to keep track of the current subset under construction is a simple list `t`. The list `ans` collects the subsets as they are constructed. The reason we take a slice `t[:]` while appending to `ans` is to pass a copy of the current subset instead of a reference to `t`, which will continue to be modified as the algorithm proceeds.

To summarize, the solution uses a DFS approach to recursively build subsets and backtrack when necessary, adding each complete subset to the solution once all elements have been considered.

Example Walkthrough

Let's take a smaller example set `[1, 2]` to illustrate the solution approach.

1. Initially, our subset `t` is empty, and we start at index `0`. The first decision is whether to include `1` in our subset or not.
2. We first choose not to include `1`, so our subset `t` remains `[]`. We call `dfs(1)` to handle the next element (index `1` now refers to `2` in our set).
3. At index `1`, again we decide whether to include `2` or not. First, we choose not to include it, so subset `t` remains as `[]`. We've now considered every element, so we append this subset to our answer list `ans`, which now has `[[]]`.
4. Backtracking to the previous choice for `2`, this time we include it in the subset `t`, which now becomes `[2]`. Again, each element has been considered, so we add `[2]` to `ans`, which becomes `[[], [2]]`.
5. Now we have finished exploring the possibilities for index `1` (with element `2`), so we backtrack to the situation before including `2`. This means we remove `2` from our subset `t` by calling `t.pop()`.
6. We then backtrack to the first decision at index `0` and choose the path where we include `1` in our subset. Now, `t` contains `[1]`. We move to index `1` by calling `dfs(1)` to make decisions for `2`.
7. At index `1`, we start with decision not to include `2` first, which means our subset `t` does not change, and is `[1]`. Since we are at the end, we add this to `ans`, resulting in `[[], [2], [1]]`.
8. Finally, we include `2` in our subset to have `[1, 2]` and since all elements are now considered, we include this subset in `ans`, resulting in `[[], [2], [1], [1, 2]]`.

Throughout this process, we've been adding our subset `t` only when we have considered all elements, which means when `i` is equal to the length of `nums`. This ensures we include every possible combination in our `ans`, the list of all subsets (`[]`, `[1]`, `[2]`, `[1, 2]` for our example).

The solution methodically explores all combinations through recursive DFS calls and includes backtracking to make sure that we cover different subsets as we make different decisions (to include/not include an element). The incremental nature of adding to the subset `t`, along with the corresponding backtracking step, ensures the completeness and correctness of the algorithm.

Python Solution

```python
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        # A helper function using depth-first search to find all subsets
        def depth_first_search(index: int):
            # Once we've considered all elements, take a snapshot of the current subset
            if index == len(nums):
                all_subsets.append(current_subset[:])
                return

            # Exclude the current element and move to the next
            depth_first_search(index + 1)

            # Include the current element and move to the next
            current_subset.append(nums[index])
            depth_first_search(index + 1)

            # Backtrack: remove the current element before going up the recursion tree
            current_subset.pop()

        # This list will hold all the subsets
        all_subsets = []
        # This is a temporary list to hold the current subset
        current_subset = []
        # Start the depth-first search from index 0
        depth_first_search(0)
        # Return the final list of all subsets
        return all_subsets
```

Java Solution

```java
class Solution {
    // A list to store all subsets
    private List<List<Integer>> subsetsList = new ArrayList<>();

    // A temporary list to store one subset
    private List<Integer> tempSubset = new ArrayList<>();

    // An array to store the given numbers
    private int[] numbers;

    /**
     * This is the main method that returns all possible subsets of the given array.
     * @param nums Array of integers for which subsets are to be found.
     * @return A list of all possible subsets of the given array.
     */
    public List<List<Integer>> subsets(int[] nums) {
        this.numbers = nums;
        // Start the Depth-First Search (DFS) with the first index
        depthFirstSearch(0);
        return subsetsList;
    }

    /**
     * This method uses Depth-First Search (DFS) to explore all potential subsets.
     * @param index The current index in the numbers array being explored.
     */
    private void depthFirstSearch(int index) {
        // If the current index has reached the length of the array,
        // it means we've formed a subset which can now be added to the list of subsets.
        if (index == numbers.length) {
            subsetsList.add(new ArrayList<>(tempSubset));
            return;
        }

        // Case 1: The current number is excluded from the subset,
        // so we simply call dfs on the next index.
        depthFirstSearch(index + 1);

        // Case 2: The current number is included in the subset.
        // Add the current number to the tempSubset.
        tempSubset.add(numbers[index]);
        // Move on to the next index to explore further with the current number included.
        depthFirstSearch(index + 1);

        // Backtrack: remove the last number added to the tempSubset,
        // this effectively removes the current number from the subset.
        tempSubset.remove(tempSubset.size() - 1);
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        // Initialize the answer vector to hold all subsets
        vector<vector<int>> answer;
        // Temporary vector to hold the current subset
        vector<int> currentSubset;

        // Define a recursive function to generate all possible subsets
        function<void(int)> generateSubsets = [&](int index) -> void {
            // Base case: If we have considered all numbers
            if (index == nums.size()) {
                // Add the current subset to the answer
                answer.push_back(currentSubset);
                return;
            }
            // Recursive case 1: Exclude the current number and move to the next
            generateSubsets(index + 1);

            // Include the current number in the subset
            currentSubset.push_back(nums[index]);
            // Recursive case 2: Include the current number and move to the next
            generateSubsets(index + 1);

            // Backtrack: Remove the current number before going back up the recursion tree
            currentSubset.pop_back();
        };

        // Start the recursion with the first index
        generateSubsets(0);
        return answer;
    }
};
```

Typescript Solution

```typescript
// This function generates all possible subsets of the given array.
function subsets(nums: number[]): number[][] {
    // 'allSubsets' will store all the subsets.
    const allSubsets: number[][] = [];

    // 'currentSubset' is a temporary storage to build each subset.
    let currentSubset: number[] = [];

    // Helper function to perform depth-first search to build subsets.
    const buildSubsets = (index: number): void => {
        // If 'index' is equal to the length of 'nums',
        // a subset is complete and we can add a copy to 'allSubsets'.
        if (index === nums.length) {
            allSubsets.push(currentSubset.slice());
            return;
        }
        // Recursive case 1: Exclude the current element and move to the next.
        buildSubsets(index + 1);

        // Include the current element in the 'currentSubset'.
        currentSubset.push(nums[index]);

        // Recursive case 2: Include the current element and move to the next.
        buildSubsets(index + 1);

        // Backtrack: Remove the last element before going up the recursive tree.
        currentSubset.pop();
    };

    // Start building subsets from the first index.
    buildSubsets(0);
    return allSubsets;
}
```

Time and Space Complexity

The provided code is a solution for finding all subsets of a given set of numbers. This is implemented using a depth-first search (DFS) approach with backtracking.

Time Complexity:

Each number has two possibilities: either it is part of a subset or it is not. Thus, for each element in the input list `nums`, we are making two recursive calls. This results in a binary decision tree with a total of 2^n leaf nodes (where n is the number of elements in `nums`).

This leads to a total of 2^n function calls. In each call, we deal with $O(1)$ complexity operations (excluding the recursive calls), such as appending an element to the temporary list `t` or appending the list to `ans`.

Therefore, the time complexity of the code is $O(2^n)$.

Space Complexity:

For space complexity, we consider two factors: the space used by the recursive call stack and the space used to store the output.

1. **Recursive Call Stack:** In the worst case, the maximum depth of the recursive call stack is n (the number of elements in `nums`), as we make a decision for each element. Hence, the space used by the call stack is $O(n)$.
2. **Output Space:** The space used to store all subsets is the dominating factor. Since there are 2^n subsets and each subset can be at most n elements, the output space complexity is $O(n * 2^n)$.

However, it is important to note that in the context of subsets or combinations problems, the space used to store the output is often considered as auxiliary space and not part of the space complexity used for algorithmic analysis. If we only consider the auxiliary space (ignoring the space for the output), the space complexity would be $O(n)$.

Taking both aspects into account, the total space complexity of the code is $O(n + 2^n)$ considering the space for the output, or $O(n)$ if we are only considering auxiliary space.