

# Simplify Java Subset Generation



Sure, I can help simplify the syntax and refactor the variable names to shorter ones. Here's a revised version of your code with shorter, meaningful names and simplified syntax:

java

```
import java.util.*;

class TUF {
    // Print the unique subsets
    static void printAns(List<String> ans) {
        System.out.println("The unique subsets are:");
        System.out.println(ans.toString().replace(",", " "));
    }

    // Generate all subsets and add them to the result set
    public static void generateSubsets(int[] nums, int idx, List<Integer> subset,
Set<String> resultSet) {
        if (idx == nums.length) {
            Collections.sort(subset);
            resultSet.add(subset.toString());
            return;
        }

        // Include current element
        subset.add(nums[idx]);
        generateSubsets(nums, idx + 1, subset, resultSet);

        // Exclude current element
        subset.remove(subset.size() - 1);
        generateSubsets(nums, idx + 1, subset, resultSet);
    }

    // Get all unique subsets
    public static List<String> subsetsWithDup(int[] nums) {
        List<String> uniqueSubsets = new ArrayList<>();
        Set<String> resultSet = new HashSet<>();
        List<Integer> subset = new ArrayList<>();
        generateSubsets(nums, 0, subset, resultSet);

        // Add all unique subsets to the list
        uniqueSubsets.addAll(resultSet);
        return uniqueSubsets;
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 2};
        List<String> result = subsetsWithDup(nums);
        printAns(result);
    }
}
```

## Explanation:

## 1. printAns Method:

- **Purpose:** Print the unique subsets.
- **Simplification:** Removed extra spaces in the print statement for a cleaner output.

## 2. generateSubsets Method:

- **Purpose:** Recursively generate all subsets and add them to the result set.
- **Parameter** ``nums``: Input array.
- **Parameter** ``idx``: Current index in the input array.
- **Parameter** ``subset``: Current subset being generated.
- **Parameter** ``resultSet``: Set to store unique subsets.

## 3. subsetsWithDup Method:

- **Purpose:** Initialize the necessary data structures and start the subset generation process.
- **Return:** List of unique subsets.

## 4. main Method:

- **Purpose:** Entry point to run the example.
- **Simplification:** Added an example input to demonstrate the functionality.

## Common Mistakes:

1. **Not Sorting Subsets:** Forgetting to sort subsets before adding them to the set can result in duplicate subsets in different orders.
2. **Incorrect Base Case Handling:** Not properly handling the base case in recursion can lead to incorrect results or infinite recursion.

## Why for What:

- **Why sort the subset before adding to the set?**
  - Sorting ensures that subsets with the same elements in different orders are considered the same, preventing duplicates.
- **Why use a HashSet for the result?**
  - A HashSet automatically handles duplicates, ensuring only unique subsets are stored.

## Next Iteration:

- **Exploration:** On each recursion step, the method explores both including and excluding the current element (``nums[idx]``).
- **Base Case:** When the index ``idx`` equals the length of the array, it means we have considered all elements and the current subset is added to the set after sorting.

Feel free to ask if you need further clarification or more examples!



Let's perform a dry run of the provided code with the example input `nums = {1, 2, 2}` to understand its execution step-by-step.

## Dry Run:

### Initial Setup:

- `nums = {1, 2, 2}`
- `uniqueSubsets = []`
- `resultSet = {}` (initially empty)
- `subset = []` (initially empty)

### Step-by-Step Execution:

#### 1. First Call to `generateSubsets(nums, 0, [], resultSet)`:

- **Current Index (idx):** 0
- **Current Subset:** []
- Options: Include `nums[0]` (1) or exclude it.

#### 2. Include `nums[0]` (1):

- Call `generateSubsets(nums, 1, [1], resultSet)`:
  - **Current Index (idx):** 1
  - **Current Subset:** [1]
  - Options: Include `nums[1]` (2) or exclude it.

#### 3. Include `nums[1]` (2):

- Call `generateSubsets(nums, 2, [1, 2], resultSet)`:
  - **Current Index (idx):** 2
  - **Current Subset:** [1, 2]
  - Options: Include `nums[2]` (2) or exclude it.

#### 4. Include `nums[2]` (2):

- Call `generateSubsets(nums, 3, [1, 2, 2], resultSet)`:
  - **Current Index (idx):** 3 (base case)
  - **Current Subset:** [1, 2, 2]
  - **Action:** Sort and add `[1, 2, 2]` to `resultSet`
  - `resultSet = {"[1, 2, 2]"}`

#### 5. Backtrack (Exclude `nums[2]` (2)):

- Call `generateSubsets(nums, 3, [1, 2], resultSet)`:

- **Current Index (idx):** 3 (base case)
- **Current Subset:** [1, 2]
- **Action:** Sort and add `[1, 2]` to `resultSet`
- `resultSet = {"[1, 2]", "[1, 2, 2]"}`

#### 6. Backtrack (Exclude `nums[1]` (2)):

- Call `generateSubsets(nums, 2, [1], resultSet)`:
  - **Current Index (idx):** 2
  - **Current Subset:** [1]
  - Options: Include `nums[2]` (2) or exclude it.

#### 7. Include `nums[2]` (2):

- Call `generateSubsets(nums, 3, [1, 2], resultSet)`:
  - **Current Index (idx):** 3 (base case)
  - **Current Subset:** [1, 2]
  - **Action:** `[1, 2]` is already in `resultSet`
  - `resultSet` remains `{"[1, 2]", "[1, 2, 2]"}`

#### 8. Backtrack (Exclude `nums[2]` (2)):

- Call `generateSubsets(nums, 3, [1], resultSet)`:
  - **Current Index (idx):** 3 (base case)
  - **Current Subset:** [1]
  - **Action:** Sort and add `[1]` to `resultSet`
  - `resultSet = {"[1]", "[1, 2]", "[1, 2, 2]"}`

#### 9. Backtrack (Exclude `nums[0]` (1)):

- Call `generateSubsets(nums, 1, [], resultSet)`:
  - **Current Index (idx):** 1
  - **Current Subset:** []
  - Options: Include `nums[1]` (2) or exclude it.

#### 10. Include `nums[1]` (2):

- Call `generateSubsets(nums, 2, [2], resultSet)`:
  - **Current Index (idx):** 2
  - **Current Subset:** [2]
  - Options: Include `nums[2]` (2) or exclude it.

#### 11. Include `nums[2]` (2):

- Call `generateSubsets(nums, 3, [2, 2], resultSet)`:
  - **Current Index (idx):** 3 (base case)
  - **Current Subset:** [2, 2]
  - **Action:** Sort and add `[2, 2]` to `resultSet`
  - `resultSet = {"[1]", "[1, 2]", "[1, 2, 2]", "[2, 2]"}`

**12. Backtrack (Exclude `nums[2]` (2)):**

- Call `generateSubsets(nums, 3, [2], resultSet)``:
  - **Current Index (idx):** 3 (base case)
  - **Current Subset:** [2]
  - **Action:** Sort and add `[2]` to `resultSet``
  - `resultSet = {"[1]", "[1, 2]", "[1, 2, 2]", "[2]", "[2, 2]"}``

**13. Backtrack (Exclude `nums[1]` (2)):**

- Call `generateSubsets(nums, 2, [], resultSet)``:
  - **Current Index (idx):** 2
  - **Current Subset:** []
  - Options: Include `nums[2]`` (2) or exclude it.

**14. Include `nums[2]` (2):**

- Call `generateSubsets(nums, 3, [2], resultSet)``:
  - **Current Index (idx):** 3 (base case)
  - **Current Subset:** [2]
  - **Action:** `[2]` is already in `resultSet``
  - `resultSet`` remains `{"[1]", "[1, 2]", "[1, 2, 2]", "[2]", "[2, 2]"}``

**15. Backtrack (Exclude `nums[2]` (2)):**

- Call `generateSubsets(nums, 3, [], resultSet)``:
  - **Current Index (idx):** 3 (base case)
  - **Current Subset:** []
  - **Action:** Sort and add `[]`` to `resultSet``
  - `resultSet = {"[]", "[1]", "[1, 2]", "[1, 2, 2]", "[2]", "[2, 2]"}``

**Final Steps:**

- Convert `resultSet`` to `uniqueSubsets`` list.
- `uniqueSubsets = ["[]", "[1]", "[1, 2]", "[1, 2, 2]", "[2]", "[2, 2]"}``

**Summary of Result:**

The unique subsets generated are:

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

The code successfully generates all unique subsets, including duplicates in the original array `nums``. This approach ensures all possible combinations are considered, sorted, and stored in a set to handle

duplicates.



Yes, we can use memoization to optimize the generation of unique subsets by storing already computed results for specific states to avoid redundant calculations. Memoization can be implemented using a map (or hash table) to store subsets that have been processed for a given index.

Here's how you can modify the code to include memoization:

1. **Introduce a memoization map** to store already processed subsets for a given index.
2. **Check the memoization map** before processing a new subset.
3. **Store the results** in the memoization map after processing.

Here's the refactored code with memoization:

java

```
import java.util.*;

class TUF {
    // Print the unique subsets
    static void printAns(List<String> ans) {
        System.out.println("The unique subsets are:");
        System.out.println(ans.toString().replace(",", " "));
    }

    // Generate all subsets and add them to the result set with memoization
    public static void generateSubsets(int[] nums, int idx, List<Integer> subset,
        Set<String> resultSet, Map<String, Boolean> memo) {
        String key = idx + " " + subset.toString();

        // Check if this state is already processed
        if (memo.containsKey(key)) {
            return;
        }

        // Mark this state as processed
        memo.put(key, true);

        if (idx == nums.length) {
            Collections.sort(subset);
            resultSet.add(subset.toString());
            return;
        }

        // Include current element
        subset.add(nums[idx]);
        generateSubsets(nums, idx + 1, subset, resultSet, memo);

        // Exclude current element
        subset.remove(subset.size() - 1);
        generateSubsets(nums, idx + 1, subset, resultSet, memo);
    }

    // Get all unique subsets
    public static List<String> subsetsWithDup(int[] nums) {
        List<String> uniqueSubsets = new ArrayList<>();
        Set<String> resultSet = new HashSet<>();
        List<Integer> subset = new ArrayList<>();
    }
}
```

```

Map<String, Boolean> memo = new HashMap<>();
generateSubsets(nums, 0, subset, resultSet, memo);

// Add all unique subsets to the list
uniqueSubsets.addAll(resultSet);
return uniqueSubsets;
}

public static void main(String[] args) {
    int[] nums = {1, 2, 2};
    List<String> result = subsetsWithDup(nums);
    printAns(result);
}
}

```

## Explanation:

### 1. Memoization Map (`memo`):

- A map is introduced to store processed states. The key is a combination of the current index (`idx`) and the current subset (`subset`).

### 2. Key Generation:

- The key is created by concatenating the index and the subset converted to a string.

### 3. State Check:

- Before processing a new state, the code checks if the current state (`key`) has already been processed by looking it up in the memoization map.

### 4. Mark State as Processed:

- After processing a state, it is marked as processed by adding the `key` to the memoization map.

## Why for What:

### • Why use memoization?

- Memoization helps to avoid redundant calculations by storing and reusing the results of already processed states, which can significantly improve the efficiency, especially for large input arrays.

## Dry Run with Memoization:

Using the same input `nums = {1, 2, 2}`:

- The `memo` map will track the states and avoid redundant subset generations.

## Example of State Keys:

- `0 []` for index 0 and subset `[]`

- `1 [1]` for index 1 and subset `[1]`
- `2 [1, 2]` for index 2 and subset `[1, 2]`

By using memoization, the code will skip processing states that have already been encountered, thus reducing the number of recursive calls and improving performance. This approach ensures that each unique subset is only generated once per state, thereby optimizing the overall execution.

Feel free to ask if you need further clarification or more examples!



Sure! Here is the complete code wrapped in the `Solution` class with the `subsetsWithDup` method. I will remove the `main` function and adjust the return type of the method to `List<List<Integer>>` as required:

java

```
import java.util.*;

class Solution {
    // Generate all subsets and add them to the result set with memoization
    private void generateSubsets(int[] nums, int idx, List<Integer> subset,
        Set<List<Integer>> resultSet, Map<String, Boolean> memo) {
        String key = idx + " " + subset.toString();

        // Check if this state is already processed
        if (memo.containsKey(key)) {
            return;
        }

        // Mark this state as processed
        memo.put(key, true);

        if (idx == nums.length) {
            List<Integer> sortedSubset = new ArrayList<>(subset);
            Collections.sort(sortedSubset);
            resultSet.add(sortedSubset);
            return;
        }

        // Include current element
        subset.add(nums[idx]);
        generateSubsets(nums, idx + 1, subset, resultSet, memo);

        // Exclude current element
        subset.remove(subset.size() - 1);
        generateSubsets(nums, idx + 1, subset, resultSet, memo);
    }

    // Get all unique subsets
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        Set<List<Integer>> resultSet = new HashSet<>();
        List<Integer> subset = new ArrayList<>();
        Map<String, Boolean> memo = new HashMap<>();
        generateSubsets(nums, 0, subset, resultSet, memo);

        // Convert the set to a list
        return new ArrayList<>(resultSet);
    }
}
```



```
    }
}
```

## Explanation:

### 1. Method `generateSubsets``:

- This private method recursively generates all subsets and uses memoization to avoid redundant calculations.
- It takes `nums`` (input array), `idx`` (current index), `subset`` (current subset being built), `resultSet`` (set to store unique subsets), and `memo`` (map for memoization).

### 2. Memoization Check:

- Before processing a new state, the code checks if the state has already been processed by looking it up in the memoization map using a `key`` which is a combination of the current index and subset.

### 3. Method `subsetsWithDup``:

- This public method initializes the required data structures and starts the subset generation process.
- It returns a list of unique subsets converted from the `resultSet``.

## How to Use:

- You can create an instance of the `Solution`` class and call the `subsetsWithDup`` method with the desired input array to get the unique subsets.
- Example:

```
java
```

```
Solution solution = new Solution();
int[] nums = {1, 2, 2};
List<List<Integer>> result = solution.subsetsWithDup(nums);
for (List<Integer> subset : result) {
    System.out.println(subset);
}
```

This code ensures that the subsets are generated efficiently using memoization and all unique subsets are returned as a list of lists of integers.



Sure! Using dynamic programming (DP) is another effective way to generate subsets, especially when dealing with duplicates. The idea is to iteratively build subsets by considering each element of the input array and handling duplicates accordingly.

Here's how you can use DP to generate all unique subsets:

## Dynamic Programming Approach:

### 1. Initialize the DP Table:

- Start with an initial empty subset.

### 2. Iterate Through Each Element:

- For each element, create new subsets by adding the element to all existing subsets.
- Use a set to handle duplicates.

## Code Implementation:

java

```
import java.util.*;

class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        // Sort the array to handle duplicates
        Arrays.sort(nums);

        // List to store the final list of unique subsets
        List<List<Integer>> uniqueSubsets = new ArrayList<>();

        // Start with an empty subset
        uniqueSubsets.add(new ArrayList<>());

        int startIndex = 0;
        int endIndex = 0;

        // Iterate through each number in the sorted array
        for (int i = 0; i < nums.length; i++) {
            startIndex = 0;

            // If the current element is a duplicate, adjust the starting index
            if (i > 0 && nums[i] == nums[i - 1]) {
                startIndex = endIndex + 1;
            }

            endIndex = uniqueSubsets.size() - 1;

            // Iterate through subsets created so far
            int n = uniqueSubsets.size();
            for (int j = startIndex; j < n; j++) {
                // Create a new subset by adding the current element to the existing subset
                List<Integer> subset = new ArrayList<>(uniqueSubsets.get(j));
                subset.add(nums[i]);

                // Add the new subset to the list of unique subsets
                uniqueSubsets.add(subset);
            }
        }

        return uniqueSubsets;
    }
}
```

## Explanation:

### 1. Sorting the Array:

- Sorting the input array helps in easily handling duplicates.

## 2. Initial Subset:

- Start with an empty subset `[]`.

## 3. Iterating Through Each Element:

- For each element in the sorted array:
  - Adjust the `startIndex` to skip subsets created by previous duplicates.
  - Create new subsets by adding the current element to all existing subsets from `startIndex` to `endIndex`.

## 4. Handling Duplicates:

- If the current element is the same as the previous one, update `startIndex` to the index just after the subsets created in the previous iteration to avoid duplicate subsets.

## How to Use:

- Create an instance of the `Solution` class and call the `subsetsWithDup` method with the desired input array to get the unique subsets.
- Example:

java

```
Solution solution = new Solution();
int[] nums = {1, 2, 2};
List<List<Integer>> result = solution.subsetsWithDup(nums);
for (List<Integer> subset : result) {
    System.out.println(subset);
}
```

This approach efficiently handles duplicates and ensures all unique subsets are generated without using recursion, making it an iterative solution with dynamic programming.



To solve the problem of finding the sums of all subsets in a list, we can use a recursive approach similar to the one provided, but I'll refine and simplify it. The goal is to implement the `subsetSums` function, which will return a list of all subset sums.

Here's the cleaned-up version of the provided solution, wrapped in a `Solution` class without the `main` method:

## Code Implementation:

java

```
import java.util.*;

class Solution {
    // Helper function to recursively calculate subset sums
    private void subsetSumsHelper(int ind, int sum, List<Integer> arr, int n, List<Integer>
sumSubset) {
        // Base case: if index reaches the end of the array, add the current sum to the
sumSubset list
        if (ind == n) {
            sumSubset.add(sum);
            return;
        }

        // Recursive call: include the current element in the subset sum
        subsetSumsHelper(ind + 1, sum + arr.get(ind), arr, n, sumSubset);

        // Recursive call: exclude the current element from the subset sum
        subsetSumsHelper(ind + 1, sum, arr, n, sumSubset);
    }

    // Function to return all subset sums
    public List<Integer> subsetSums(List<Integer> arr, int n) {
        List<Integer> sumSubset = new ArrayList<>();
        subsetSumsHelper(0, 0, arr, n, sumSubset);
        Collections.sort(sumSubset); // Optional: sort the result list if needed
        return sumSubset;
    }
}
```

## Explanation:

### 1. Helper Function `subsetSumsHelper`:

- **Parameters:** `ind` (current index), `sum` (current subset sum), `arr` (input list), `n` (size of input list), `sumSubset` (list to store subset sums).
- **Base Case:** When `ind` equals `n` (i.e., we've considered all elements), add the current `sum` to `sumSubset`.
- **Recursive Calls:**
  - **Include Element:** Add `arr.get(ind)` to `sum` and move to the next element.
  - **Exclude Element:** Keep `sum` unchanged and move to the next element.

### 2. Main Function `subsetSums`:

- Initializes an empty list `sumSubset` to store the results.
- Calls `subsetSumsHelper` to populate `sumSubset`.
- Optionally sorts `sumSubset` before returning it (depending on problem requirements).

## Usage:

To use the `Solution` class and the `subsetSums` method:

java

```

public class Main {
    public static void main(String[] args) {
        Solution solution = new Solution();
        List<Integer> arr = Arrays.asList(5, 2, 1);
        List<Integer> result = solution.subsetSums(arr, arr.size());
        System.out.println("The sum of each subset is: " + result);
    }
}

```

## Dry Run Example:

For input `arr = {5, 2, 1}`:

- Subset sums will be calculated as follows:
  - `0` (empty subset)
  - `5` (subset {5})
  - `2` (subset {2})
  - `7` (subset {5, 2})
  - `1` (subset {1})
  - `6` (subset {5, 1})
  - `3` (subset {2, 1})
  - `8` (subset {5, 2, 1})
- The final sorted result will be: `[0, 1, 2, 3, 5, 6, 7, 8]`

This approach ensures that all subsets are considered, and their sums are correctly calculated and stored.



Yes, we can use memoization to optimize the computation of subset sums. Memoization helps avoid redundant calculations by storing the results of already computed states. Here's how you can incorporate memoization into the subset sum calculation:

## Code Implementation with Memoization:

java

```

import java.util.*;

class Solution {
    // Helper function to recursively calculate subset sums with memoization
    private void subsetSumsHelper(int ind, int sum, List<Integer> arr, int n, List<Integer>
sumSubset, Map<String, Boolean> memo) {
        // Create a unique key for the current state
        String key = ind + "," + sum;

        // Check if this state has already been computed
        if (memo.containsKey(key)) {
            return;
        }
    }
}

```

```

    }

    // Mark this state as computed
    memo.put(key, true);

    // Base case: if index reaches the end of the array, add the current sum to the
sumSubset list
    if (ind == n) {
        sumSubset.add(sum);
        return;
    }

    // Recursive call: include the current element in the subset sum
    subsetSumsHelper(ind + 1, sum + arr.get(ind), arr, n, sumSubset, memo);

    // Recursive call: exclude the current element from the subset sum
    subsetSumsHelper(ind + 1, sum, arr, n, sumSubset, memo);
}

// Function to return all subset sums
public List<Integer> subsetSums(List<Integer> arr, int n) {
    List<Integer> sumSubset = new ArrayList<>();
    Map<String, Boolean> memo = new HashMap<>();
    subsetSumsHelper(0, 0, arr, n, sumSubset, memo);
    Collections.sort(sumSubset); // Optional: sort the result list if needed
    return sumSubset;
}
}

```

## Explanation:

### 1. Memoization Map (`memo`):

- A map is introduced to store already computed states. The key is a combination of the current index (`ind`) and the current subset sum (`sum`).

### 2. Key Generation:

- The key is created by concatenating the index and the subset sum with a comma.

### 3. State Check:

- Before processing a new state, the code checks if the current state (`key`) has already been computed by looking it up in the memoization map.

### 4. Mark State as Computed:

- After processing a state, it is marked as computed by adding the `key` to the memoization map.

## How to Use:

To use the `Solution` class and the `subsetSums` method:

java

```

public class Main {
    public static void main(String[] args) {
        Solution solution = new Solution();
        List<Integer> arr = Arrays.asList(5, 2, 1);
        List<Integer> result = solution.subsetSums(arr, arr.size());
        System.out.println("The sum of each subset is: " + result);
    }
}

```

## Dry Run Example with Memoization:

For input `arr = {5, 2, 1}`:

- The memoization map will help skip redundant calculations, ensuring each state is processed only once.

## Memoization Example States:

- `0,0` for index 0 and sum 0
- `1,0` for index 1 and sum 0
- `1,5` for index 1 and sum 5
- `2,0` for index 2 and sum 0
- `2,2` for index 2 and sum 2
- `2,5` for index 2 and sum 5
- `2,7` for index 2 and sum 7
- and so on...

By using memoization, the code avoids recalculating sums for the same subsets, thus optimizing the execution, especially for larger inputs. This approach ensures all unique subset sums are efficiently computed and stored.



Here's the corrected note for your reference:

## Note:

"If you use memoization, the answer will not contain the sums of duplicate subsets. To ensure that you get all subset sums, including duplicates, do not use the memoization approach for the subset sums problem as described above."

## Updated Code Without Memoization:

To ensure the subset sums include all duplicates, here is the clean version of the code without memoization:

java

```

import java.util.*;

class Solution {
    // Helper function to recursively calculate subset sums
    private void subsetSumsHelper(int ind, int sum, List<Integer> arr, int n, List<Integer>
sumSubset) {
        // Base case: if index reaches the end of the array, add the current sum to the
sumSubset list
        if (ind == n) {
            sumSubset.add(sum);
            return;
        }

        // Recursive call: include the current element in the subset sum
        subsetSumsHelper(ind + 1, sum + arr.get(ind), arr, n, sumSubset);

        // Recursive call: exclude the current element from the subset sum
        subsetSumsHelper(ind + 1, sum, arr, n, sumSubset);
    }

    // Function to return all subset sums
    public List<Integer> subsetSums(List<Integer> arr, int n) {
        List<Integer> sumSubset = new ArrayList<>();
        subsetSumsHelper(0, 0, arr, n, sumSubset);
        Collections.sort(sumSubset); // Optional: sort the result list if needed
        return sumSubset;
    }
}

```

## Explanation:

### 1. Helper Function `subsetSumsHelper`:

- **Parameters:** `ind` (current index), `sum` (current subset sum), `arr` (input list), `n` (size of input list), `sumSubset` (list to store subset sums).
- **Base Case:** When `ind` equals `n` (i.e., we've considered all elements), add the current `sum` to `sumSubset`.
- **Recursive Calls:**
  - **Include Element:** Add `arr.get(ind)` to `sum` and move to the next element.
  - **Exclude Element:** Keep `sum` unchanged and move to the next element.

### 2. Main Function `subsetSums`:

- Initializes an empty list `sumSubset` to store the results.
- Calls `subsetSumsHelper` to populate `sumSubset`.
- Optionally sorts `sumSubset` before returning it (depending on problem requirements).

## Usage:

To use the `Solution` class and the `subsetSums` method:



java

```

public class Main {
    public static void main(String[] args) {
        Solution solution = new Solution();
        List<Integer> arr = Arrays.asList(5, 2, 1);
        List<Integer> result = solution.subsetSums(arr, arr.size());
        System.out.println("The sum of each subset is: " + result);
    }
}

```

This approach ensures that all subsets, including those that are duplicates, are correctly calculated and included in the final result.



Using dynamic programming (DP) to calculate subset sums can indeed be modified from the subset sum problem where we check if a subset can form a given target sum. However, to get all subset sums efficiently, it would still involve a form of  $2^n$  complexity due to the nature of subsets generation. Nonetheless, DP can still optimize certain aspects. Here's how you can do it:

## DP Approach for Subset Sums:

### 1. Define the DP Table:

- Use a boolean DP table `dp[i][j]` where `i` is the index of the array and `j` is the possible sum.
- `dp[i][j]` will be `true` if there exists a subset of the first `i` elements with sum `j`.

### 2. Initialization:

- `dp[0][0]` is `true` because a subset with sum 0 is always possible (the empty subset).

### 3. DP Transition:

- For each element in the array, update the DP table to include/exclude the current element for all possible sums.

### 4. Extract Subset Sums:

- After filling the DP table, extract all sums that can be formed.

Here's how the implementation would look in Java:

java

```

import java.util.*;

```

```

class Solution {
    public List<Integer> subsetSums(List<Integer> arr, int n) {
        // Find the total sum of the array elements
        int totalSum = arr.stream().mapToInt(Integer::intValue).sum();

        // DP table to store subset sum possibilities
        boolean[][] dp = new boolean[n + 1][totalSum + 1];

        // Initialize the DP table
        for (int i = 0; i <= n; i++) {
            dp[i][0] = true; // Sum 0 is always possible
        }

        // Fill the DP table
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= totalSum; j++) {
                dp[i][j] = dp[i - 1][j]; // Exclude the current element
                if (j >= arr.get(i - 1)) {
                    dp[i][j] |= dp[i - 1][j - arr.get(i - 1)]; // Include the current
                    element
                }
            }
        }

        // Extract all possible subset sums
        List<Integer> result = new ArrayList<>();
        for (int j = 0; j <= totalSum; j++) {
            if (dp[n][j]) {
                result.add(j);
            }
        }

        Collections.sort(result); // Sort the result if needed
        return result;
    }
}

```

## Explanation:

### 1. Initialization:

- We initialize `dp[i][0]` to `true` for all `i` because a sum of 0 is always possible with the empty subset.

### 2. DP Transition:

- For each element in the array, update the DP table:
  - `dp[i][j]` is `true` if `dp[i-1][j]` is `true` (excluding the current element).
  - If `j >= arr.get(i-1)`, `dp[i][j]` is `true` if `dp[i-1][j - arr.get(i-1)]` is `true` (including the current element).

### 3. Extract Subset Sums:

- After filling the DP table, iterate through the last row to find all sums that can be formed using subsets of the array.

## Usage:

To use the `Solution` class and the `subsetSums` method:

java

```
public class Main {  
    public static void main(String[] args) {  
        Solution solution = new Solution();  
        List<Integer> arr = Arrays.asList(5, 2, 1);  
        List<Integer> result = solution.subsetSums(arr, arr.size());  
        System.out.println("The sum of each subset is: " + result);  
    }  
}
```

## Complexity:

- **Time Complexity:**  $O(n * \text{totalSum})$ , where `totalSum` is the sum of all elements in the array. This is still exponential in the worst case, but it can be more efficient in practice for smaller sums.
- **Space Complexity:**  $O(n * \text{totalSum})$ , due to the DP table used to store subset sum possibilities.

This approach ensures all possible subset sums are calculated and stored efficiently using dynamic programming, avoiding the explicit generation of all subsets.

ChatGPT can make mistakes. Check important info.