DP Techniques

To solve the subset sum problem using recursion, memoization, and dynamic programming in Java, let's follow these steps:

1. **Recursive Approach**: This is the brute force approach where we try every possible subset.
2. **Memoized Approach**: This approach uses recursion but stores the results of subproblems to avoid redundant calculations.
3. **Dynamic Programming Approach**: This approach builds the solution iteratively using a table to store the results of subproblems.

1. Recursive Approach

This approach checks every possible subset to see if any subset has a sum equal to the target sum.

java

```java
public class SubsetSum {
    // Recursive function to determine if a subset with the given sum exists
    public static boolean isSubsetSumRecursive(int[] arr, int n, int sum) {
        // Base cases
        if (sum == 0) {
            return true; // Empty subset always has a sum of 0
        }
        if (n == 0) {
            return false; // No items left to choose from and sum is not 0

        // If the last element is greater than sum, ignore it
        if (arr[n-1] > sum) {
            return isSubsetSumRecursive(arr, n-1, sum);
        }

        // Check if sum can be obtained by any of:
        // (1) including the last element
        // (2) excluding the last element
        return isSubsetSumRecursive(arr, n-1, sum) || isSubsetSumRecursive(arr, n-1, sum-
arr[n-1]);
    }

    public static void main(String[] args) {
        int[] arr = {3, 34, 4, 12, 5, 2};
        int sum = 9;
        int n = arr.length;
        System.out.println(isSubsetSumRecursive(arr, n, sum) ? 1 : 0);
    }
}
```

2. Memoized Approach

This approach uses a 2D array to store the results of subproblems to avoid redundant calculations.

java

```java
import java.util.Arrays;

public class SubsetSum {
    // Memoization table
    static Boolean[][] memo;

    // Helper function for memoized approach
    public static boolean isSubsetSumMemo(int[] arr, int n, int sum) {
        memo = new Boolean[n+1][sum+1];
        for (Boolean[] row : memo) {
            Arrays.fill(row, null); // Initialize memoization table with null values
        }
        return isSubsetSumMemoHelper(arr, n, sum);
    }

    public static boolean isSubsetSumMemoHelper(int[] arr, int n, int sum) {
        // Base cases
        if (sum == 0) {
            return true; // Empty subset always has a sum of 0
        }
        if (n == 0) {
            return false; // No items left to choose from and sum is not 0

        // Check if result is already computed
        if (memo[n][sum] != null) {
            return memo[n][sum];
        }

        // If the last element is greater than sum, ignore it
        if (arr[n-1] > sum) {
            memo[n][sum] = isSubsetSumMemoHelper(arr, n-1, sum);
        } else {
            // Check if sum can be obtained by any of:
            // (1) including the last element
            // (2) excluding the last element
            memo[n][sum] = isSubsetSumMemoHelper(arr, n-1, sum) ||
isSubsetSumMemoHelper(arr, n-1, sum-arr[n-1]);
        }
        return memo[n][sum];
    }

    public static void main(String[] args) {
        int[] arr = {3, 34, 4, 12, 5, 2};
        int sum = 9;
        int n = arr.length;
        System.out.println(isSubsetSumMemo(arr, n, sum) ? 1 : 0);
    }
}
```

3. Dynamic Programming Approach

This approach builds a table `dp` where `dp[i][j]` will be `true` if there is a subset of the first `i` elements with sum `j`.

java

```java
public class SubsetSum {
    public static boolean isSubsetSumDP(int[] arr, int n, int sum) {
        // Create a dp table with dimensions (n+1) x (sum+1)
        boolean[][] dp = new boolean[n+1][sum+1];

        // If sum is 0, answer is true (empty subset)
        for (int i = 0; i <= n; i++) {
            dp[i][0] = true;
```

```
        }

        // If no elements and sum is not 0, answer is false
        for (int j = 1; j <= sum; j++) {
            dp[0][j] = false;
        }

        // Fill the dp table
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= sum; j++) {
                if (arr[i-1] > j) {
                    dp[i][j] = dp[i-1][j]; // Exclude the element
                } else {
                    dp[i][j] = dp[i-1][j] || dp[i-1][j-arr[i-1]]; // Include or exclude the
element
                }
            }
        }

        // The answer will be in dp[n][sum]
        return dp[n][sum];
    }

    public static void main(String[] args) {
        int[] arr = {3, 34, 4, 12, 5, 2};
        int sum = 9;
        int n = arr.length;
        System.out.println(isSubsetSumDP(arr, n, sum) ? 1 : 0);
    }
}
```

Explanation:

1. **Recursive Approach**:
    - Base cases check if the sum is 0 or if no elements are left.
    - Recursively check if the sum can be obtained by including or excluding the current element.
2. **Memoized Approach**:
    - Use a memoization table to store results of subproblems.
    - Avoids redundant calculations by checking if the result for a particular subproblem is already computed.
3. **Dynamic Programming Approach**:
    - Use a 2D DP table where `dp[i][j]` represents if a subset with sum `j` can be formed using the first `i` elements.
    - Fill the table iteratively using the recursive relation.

Each approach improves on the previous one in terms of efficiency, and the DP approach provides an optimal solution within the given constraints.

To solve the problem of partitioning an array into two subsets with equal sums, we can approach it similarly to the subset sum problem. Here's how we can solve it using a recursive approach, memoized approach, and dynamic programming approach in Java.

1. Recursive Approach

The idea is to check if we can find a subset with sum equal to half of the total sum of the array. If the total sum is odd, we can't partition it into two equal subsets.

java

```java
public class PartitionEqualSubsetSum {
    public static boolean canPartitionRecursive(int[] nums) {
        int totalSum = 0;
        for (int num : nums) {
            totalSum += num;
        }
        // If the total sum is odd, we cannot partition it into two equal subsets
        if (totalSum % 2 != 0) {
            return false;
        }
        int target = totalSum / 2;
        return canPartitionRecursiveHelper(nums, nums.length - 1, target);
    }

    private static boolean canPartitionRecursiveHelper(int[] nums, int n, int target) {
        // Base cases
        if (target == 0) {
            return true; // Found a subset with the required sum
        }
        if (n < 0 || target < 0) {
            return false; // No more elements to consider or target is negative
        }

        // Recursively check if we can find a subset with the given sum
        // either including or excluding the current element
        return canPartitionRecursiveHelper(nums, n - 1, target) ||
               canPartitionRecursiveHelper(nums, n - 1, target - nums[n]);
    }

    public static void main(String[] args) {
        int[] nums1 = {1, 5, 11, 5};
        int[] nums2 = {1, 2, 3, 5};
        System.out.println(canPartitionRecursive(nums1)); // Output: true
        System.out.println(canPartitionRecursive(nums2)); // Output: false
    }
}
```

2. Memoized Approach

We use a 2D array to store the results of subproblems and avoid redundant calculations.

java

```java
import java.util.Arrays;

public class PartitionEqualSubsetSum {
    public static boolean canPartitionMemo(int[] nums) {
        int totalSum = 0;
        for (int num : nums) {
            totalSum += num;
        }
        if (totalSum % 2 != 0) {
            return false;
        }
        int target = totalSum / 2;
        Boolean[][] memo = new Boolean[nums.length][target + 1];
        return canPartitionMemoHelper(nums, nums.length - 1, target, memo);
```

```
    }

    private static boolean canPartitionMemoHelper(int[] nums, int n, int target, Boolean[][]
memo) {
        if (target == 0) {
            return true;
        }
        if (n < 0 || target < 0) {
            return false;
        }
        if (memo[n][target] != null) {
            return memo[n][target];
        }
        memo[n][target] = canPartitionMemoHelper(nums, n - 1, target, memo) ||
                          canPartitionMemoHelper(nums, n - 1, target - nums[n]);
        return memo[n][target];
    }

    public static void main(String[] args) {
        int[] nums1 = {1, 5, 11, 5};
        int[] nums2 = {1, 2, 3, 5};
        System.out.println(canPartitionMemo(nums1)); // Output: true
        System.out.println(canPartitionMemo(nums2)); // Output: false
    }
}
```

3. Dynamic Programming Approach

We use a boolean DP table where `dp[i][j]` will be `true` if a subset with sum `j` can be formed using the first `i` elements.

java

```
public class PartitionEqualSubsetSum {
    public static boolean canPartitionDP(int[] nums) {
        int totalSum = 0;
        for (int num : nums) {
            totalSum += num;
        }
        if (totalSum % 2 != 0) {
            return false;
        }
        int target = totalSum / 2;
        boolean[][] dp = new boolean[nums.length + 1][target + 1];

        for (int i = 0; i <= nums.length; i++) {
            dp[i][0] = true; // Sum 0 can always be formed with 0 elements
        }

        for (int i = 1; i <= nums.length; i++) {
            for (int j = 1; j <= target; j++) {
                if (nums[i - 1] > j) {
                    dp[i][j] = dp[i - 1][j]; // Exclude the current element
                } else {
                    dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i - 1]]; // Include or
exclude the current element
                }
            }
        }

        return dp[nums.length][target];
    }

    public static void main(String[] args) {
        int[] nums1 = {1, 5, 11, 5};
        int[] nums2 = {1, 2, 3, 5};
        System.out.println(canPartitionDP(nums1)); // Output: true
        System.out.println(canPartitionDP(nums2)); // Output: false
```

```
    }
}
```

Explanation:

1. **Recursive Approach**:
   - Check if we can find a subset with sum equal to half of the total sum of the array.
   - Recursively check if the target can be achieved by including or excluding the current element.
2. **Memoized Approach**:
   - Use a memoization table to store results of subproblems to avoid redundant calculations.
   - Similar to the recursive approach but with caching.
3. **Dynamic Programming Approach**:
   - Use a DP table where `dp[i][j]` indicates if a subset with sum `j` can be formed using the first `i` elements.
   - Fill the table iteratively based on whether the current element is included or excluded.

Each approach builds on the previous one in terms of efficiency, with the DP approach providing an optimal solution within the given constraints.

java

```java
// We are using isSubsetSumRecursive function to find whether the subset of sum
// target/2 exists in the given array or not such that if it exists then we can
// guarantee that sum/2 is there so sum can be split into two equal subsets and
// all these canPartitionRecursive and canPartitionMemo were doing the exact task
// as isSubsetSumRecursive.

public class PartitionEqualSubsetSum {
    public static boolean canPartitionRecursive(int[] nums) {
        int totalSum = 0;
        for (int num : nums) {
            totalSum += num;
        }
        // If the total sum is odd, we cannot partition it into two equal subsets
        if (totalSum % 2 != 0) {
            return false;
        }
        int target = totalSum / 2;
        return isSubsetSumRecursive(nums, nums.length, target);
    }

    // Recursive function to determine if a subset with the given sum exists
    private static boolean isSubsetSumRecursive(int[] arr, int n, int sum) {
        // Base cases
        if (sum == 0) {
            return true; // Empty subset always has a sum of 0
        }
        if (n == 0) {
            return false; // No items left to choose from and sum is not 0
        }
        // If the last element is greater than sum, ignore it
```

```
        if (arr[n-1] > sum) {
            return isSubsetSumRecursive(arr, n-1, sum);
        }
        // Check if sum can be obtained by any of:
        // (1) including the last element
        // (2) excluding the last element
        return isSubsetSumRecursive(arr, n-1, sum) || isSubsetSumRecursive(arr, n-1, sum-
arr[n-1]);
    }

    public static void main(String[] args) {
        int[] nums1 = {1, 5, 11, 5};
        int[] nums2 = {1, 2, 3, 5};
        System.out.println(canPartitionRecursive(nums1)); // Output: true
        System.out.println(canPartitionRecursive(nums2)); // Output: false
    }
}
```

java

```
// We are using isSubsetSumRecursive function to find whether the subset of sum
// target/2 exists in the given array or not such that if it exists then we can
// guarantee that sum/2 is there so sum can be split into two equal subsets and
// all these canPartitionRecursive and canPartitionMemo were doing the exact task
// as isSubsetSumRecursive.

import java.util.Arrays;

public class PartitionEqualSubsetSum {
    public static boolean canPartitionMemo(int[] nums) {
        int totalSum = 0;
        for (int num : nums) {
            totalSum += num;
        }
        // If the total sum is odd, we cannot partition it into two equal subsets
        if (totalSum % 2 != 0) {
            return false;
        }
        int target = totalSum / 2;
        Boolean[][] memo = new Boolean[nums.length + 1][target + 1];
        for (Boolean[] row : memo) {
            Arrays.fill(row, null); // Initialize memoization table with null values
        }
        return isSubsetSumMemo(nums, nums.length, target, memo);
    }

    // Memoized function to determine if a subset with the given sum exists
    private static boolean isSubsetSumMemo(int[] arr, int n, int sum, Boolean[][] memo) {
        // Base cases
        if (sum == 0) {
            return true; // Empty subset always has a sum of 0
        }
        if (n == 0) {
            return false; // No items left to choose from and sum is not 0
        }
        // Check if result is already computed
        if (memo[n][sum] != null) {
            return memo[n][sum];
        }
        // If the last element is greater than sum, ignore it
        if (arr[n-1] > sum) {
            memo[n][sum] = isSubsetSumMemo(arr, n-1, sum, memo);
        } else {
            // Check if sum can be obtained by any of:
            // (1) including the last element
            // (2) excluding the last element
            memo[n][sum] = isSubsetSumMemo(arr, n-1, sum, memo) || isSubsetSumMemo(arr, n-1,
sum-arr[n-1], memo);
        }
        return memo[n][sum];
    }
```

```
    public static void main(String[] args) {
        int[] nums1 = {1, 5, 11, 5};
        int[] nums2 = {1, 2, 3, 5};
        System.out.println(canPartitionMemo(nums1)); // Output: true
        System.out.println(canPartitionMemo(nums2)); // Output: false
    }
}
```

In the subset sum problem, we are trying to determine if a subset with a given sum exists in the array. In the DP table for this problem, the value at each index `dp[i][j]` represents whether it is possible to form the sum `j` using the first `i` elements of the array.

Detailed Explanation:

- `dp[i][j] = true` means that there exists a subset of the first `i` elements that sums up to `j`.
- `dp[i][j] = false` means that there is no subset of the first `i` elements that sums up to `j`.

Initialization:

- `dp[0][0] = true`: An empty subset can achieve the sum of 0.
- `dp[i][0] = true` for all `i`: Any subset (including an empty one) can achieve the sum of 0.
- `dp[0][j] = false` for all `j > 0`: No subset of 0 elements (an empty set) can achieve a non-zero sum.

Transition:

For each element `nums[i-1]` and for each possible sum `j`:

- If the element is excluded: `dp[i][j] = dp[i-1][j]`
- If the element is included: `dp[i][j] = dp[i-1][j] || dp[i-1][j-nums[i-1]]` (only if `j >= nums[i-1]`)

Example:

For `nums = [1, 5, 11, 5]` and `target = 11`:

- `dp[0][0] = true` (base case)
- `dp[1][0] = true` (base case)
- `dp[1][1] = true` because `1` can form the sum `1`
- `dp[2][6] = true` because `1 + 5` can form the sum `6`
- ...

The final answer will be found at `dp[n][target]`, where `n` is the number of elements in the array and `target` is the required sum. If `dp[n][target]` is `true`, then a subset with the given sum exists; otherwise, it does not.

Code Example:

```java
public class SubsetSum {
    public static boolean isSubsetSumDP(int[] nums, int target) {
        int n = nums.length;
        boolean[][] dp = new boolean[n + 1][target + 1];

        // Initialize base cases
        for (int i = 0; i <= n; i++) {
            dp[i][0] = true; // A subset with sum 0 is always possible
        }

        // Fill the dp table
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= target; j++) {
                if (nums[i - 1] <= j) {
                    dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i - 1]];
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        return dp[n][target];
    }

    public static void main(String[] args) {
        int[] nums = {1, 5, 11, 5};
        int target = 11;
        System.out.println(isSubsetSumDP(nums, target)); // Output: true
    }
}
```

This code initializes and fills the DP table according to the explained logic, ultimately determining if a subset with the desired sum exists.

In the knapsack problem, we aim to maximize the total value of items that can be included in a knapsack with a weight capacity constraint. In the DP table for this problem, the value at each index `dp[i][j]` represents the maximum value that can be achieved with the first `i` items and a total weight capacity of `j`.

Detailed Explanation:

- `dp[i][j]` is the maximum value that can be obtained using the first `i` items with a total weight not exceeding `j`.
- `dp[i][j] = 0` if either `i == 0` or `j == 0` because no items or zero capacity means zero value.

Initialization:

- `dp[0][j] = 0` for all `j`: With 0 items, the maximum value is always 0 regardless of the weight capacity.
- `dp[i][0] = 0` for all `i`: With 0 capacity, the maximum value is always 0 regardless of the number of items.

Transition:

For each item `i` with weight `w[i-1]` and value `v[i-1]`, and for each possible capacity `j`:

- If the item is excluded: `dp[i][j] = dp[i-1][j]`
- If the item is included (only if `j >= w[i-1]`): `dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i-1]] + v[i-1])`

Example:

For items with weights `[w1, w2, ...]`, values `[v1, v2, ...]`, and knapsack capacity `C`:

- `dp[0][0] = 0` (base case)
- `dp[1][w1] = v1` if `w1 <= C`
- `dp[2][w2] = max(v2, dp[1][w2])` if `w2 <= C`
- ...

The final answer will be found at `dp[n][C]`, where `n` is the number of items and `C` is the knapsack capacity. This value represents the maximum value that can be obtained with the given items and capacity constraints. If `dp[n][C]` is the highest value achievable, then it represents the optimal solution for the knapsack problem.