Questions and Answers

1. Longest Palindromic Subsequence using Dynamic Programming

The Longest Palindromic Subsequence (LPS) problem is finding the longest subsequences of a string that is also a palindrome.The problem differs from the problem of finding the **longest palindromic substring**. Unlike substrings, **subsequences** are not required to occupy consecutive positions within the original string.
For example, consider the sequence ABBDCACB.

The idea is to use recursion to solve this problem. The idea is to compare the last character of the string `X[i…j]` with its first character. There are two possibilities:

1. If the string's last character is the same as the first character, include the first and last characters in palindrome and recur for the remaining substring `X[i+1, j-1]`.
2. If the last character of the string is different from the first character, return the maximum of the two values we get by
    - Removing the last character and recursing for the remaining substring `X[i, j-1]`.
    - Removing the first character and recursing for the remaining substring `X[i+1, j]`.

This yields the following recursive relation to finding the length of the longest repeated subsequence of a sequence `X`:

```
           | 1                                 (if i = j)
LPS[i…j] = | LPS[i+1…j-1] + 2                  (if X[i] = X[j])
           | max (LPS[i+1…j], LPS[i…j-1])      (if X[i] != X[j])
```

```python
def longest_palindrome(s):
    n = len(s)

    # Create a table to store results of subproblems
    table = [[0] * n for _ in range(n)]

    # All substrings of length 1 are palindromes
    for i in range(n):
        table[i][i] = 1

    start = 0  # Variable to store the start index of the longest pa
    max_length = 1  # Variable to store the length of the longest pa

    # Check for substrings of length 2
    for i in range(n - 1):
        if s[i] == s[i + 1]:
            table[i][i + 1] = 1
            start = i
            max_length = 2
```

```
    # Check for substrings of length 3 and greater
    for k in range(3, n + 1):
        for i in range(n - k + 1):
            j = i + k - 1  # Ending index of the substring
            if table[i + 1][j - 1] and s[i] == s[j]:
                table[i][j] = 1
                start = i
                max_length = k

    return s[start:start + max_length]

# Example usage:
s = "babad"
result = longest_palindrome(s)
print(result)
```

This implementation uses a 2D table where table[i][j] is True if the substring s[i:j+1] is a palindrome. The algorithm iterates through all possible substrings of different lengths, updating the table accordingly. Finally, it returns the longest palindromic substring found.

Note that this solution has a time complexity of O(n^2) and a space complexity of O(n^2).

```
function longestPalindrome(s):
    n = length of s

    // Create a table to store results of subproblems
    table = 2D array of size n x n, initialized with zeros

    // All substrings of length 1 are palindromes
    for i from 0 to n - 1:
        table[i][i] = 1

    start = 0  // Variable to store the start index of the longest palindrome
    maxLength = 1  // Variable to store the length of the longest palindrome

    // Check for substrings of length 2
```

```
    for i from 0 to n - 2:
        if s[i] equals s[i + 1]:
            table[i][i + 1] = 1
            start = i
            maxLength = 2

    // Check for substrings of length 3 and greater
    for k from 3 to n:
        for i from 0 to n - k:
            j = i + k - 1  // Ending index of the substring
            if table[i + 1][j - 1] equals 1 and s[i] equals s[j]:
                table[i][j] = 1
                start = i
                maxLength = k

    return substring s[start to start + maxLength - 1]
```

Note that this solution has a time complexity of O(n^2) and a space complexity of O(n^2).

2. Implement Diff Utility

Implement your diff utility, i.e., given two similar strings, efficiently list out all differences between them.

The diff utility is a data comparison tool that calculates and displays the differences between the two texts. It tries to determine the smallest set of deletions and insertions and create one text from the other. Diff is line-oriented rather than character-oriented, unlike edit distance.
e.g.
string X = XMJYAUZ
string Y = XMJAATZ
Output: X M J -Y A -U +A +T Z
(- indicates that character is deleted from Y but it was present in X)
(+ indicates that character is inserted in Y but it was not present in X)

We can use the Longest Common Subsequence (LCS) to solve this problem. The idea is to find the longest sequence of characters present in both original sequences in the same order. From the longest common subsequence, it is only a small step to get the diff-like output:

If a character is absent in the subsequence but present in the first original sequence, it must have been deleted (indicated by the - marks).
If it is absent in the subsequence but present in the second original sequence, it must have been inserted (indicated by the + marks).

```
function diffUtility(str1, str2):
    m = length of str1
    n = length of str2

    // Create a table to store results of subproblems
    table = 2D array of size (m + 1) x (n + 1), initialized with zeros

    // Base case: Initialization
    for i from 0 to m:
        table[i][0] = i
    for j from 0 to n:
        table[0][j] = j

    // Fill the table using bottom-up dynamic programming
    for i from 1 to m:
        for j from 1 to n:
            if str1[i - 1] equals str2[j - 1]:
                table[i][j] = table[i - 1][j - 1]
            else:
                table[i][j] = 1 + min(table[i - 1][j],       // Deletion
                                      table[i][j - 1],       // Insertion
                                      table[i - 1][j - 1])   // Substitution

    return table[m][n]  // Edit distance between str1 and str2
```

The diff utility problem is a classic dynamic programming problem that involves finding the minimum number of single-character edit.

Explanation:

- table[i][j] represents the edit distance between the first i characters of str1 and the first j characters of str2.
- The base case initializes the table for the empty string to any substring. The idea is that the edit distance from an empty string to a non-empty string is the length of the non-empty string.
- The main dynamic programming loop fills the table by considering the three possible operations: deletion, insertion, and substitution.
- If the current characters in both strings are equal, no operation is needed (table[i][j] = table[i-1][j-1]).
- If the characters are not equal, the edit distance is incremented by 1, and the minimum of the three possible operations is considered (table[i][j] = 1 + min(...)).
- The final result is stored in table[m][n], representing the edit distance between the entire strings str1 and str2.

```
function diffUtilityWithEditDetails(str1, str2):
    m = length of str1
    n = length of str2

    // Create a table to store results of subproblems
    table = 2D array of size (m + 1) x (n + 1), initialized with zeros
    operations = 2D array of size (m + 1) x (n + 1), initialized with empty strings

    // Base case: Initialization
    for i from 0 to m:
        table[i][0] = i
        operations[i][0] = "Delete " + str1[i - 1]
    for j from 0 to n:
        table[0][j] = j
        operations[0][j] = "Insert " + str2[j - 1]

    // Fill the table and operations using bottom-up dynamic programming
    for i from 1 to m:
        for j from 1 to n:
            if str1[i - 1] equals str2[j - 1]:
                table[i][j] = table[i - 1][j - 1]
                operations[i][j] = operations[i - 1][j - 1]
            else:
                deletion = table[i - 1][j] + 1
                insertion = table[i][j - 1] + 1
                substitution = table[i - 1][j - 1] + 1

                minOperation = min(deletion, insertion, substitution)

                table[i][j] = minOperation

                if minOperation equals deletion:
                    operations[i][j] = operations[i - 1][j] + "\nDelete " + str1[i -
1]
                else if minOperation equals insertion:
                    operations[i][j] = operations[i][j - 1] + "\nInsert " + str2[j -
1]
                else:
                    operations[i][j] = operations[i - 1][j - 1] + "\nSubstitute " +
str1[i - 1] + " with " + str2[j - 1]

    return table[m][n], operations[m][n]  // Edit distance and detailed operations
```

Top-down approach

```
function diffUtilityTopDown(str1, str2):
    memo = 2D array of size (length of str1 + 1) x (length of str2 + 1), initialized
with -1
    return diffUtilityTopDownHelper(str1, str2, length of str1, length of str2, memo)

function diffUtilityTopDownHelper(str1, str2, i, j, memo):
    // Base case: If either of the strings is empty, return the length of the other
string
    if i == 0:
        return j
    if j == 0:
        return i

    // Check if the result is already memoized
    if memo[i][j] ≠ -1:
        return memo[i][j]

    // If the last characters are equal, no operation is needed
    if str1[i - 1] equals str2[j - 1]:
        memo[i][j] = diffUtilityTopDownHelper(str1, str2, i - 1, j - 1, memo)
        return memo[i][j]

    // Calculate the minimum of three operations: deletion, insertion, and
substitution
    deletion = 1 + diffUtilityTopDownHelper(str1, str2, i - 1, j, memo)
    insertion = 1 + diffUtilityTopDownHelper(str1, str2, i, j - 1, memo)
    substitution = 1 + diffUtilityTopDownHelper(str1, str2, i - 1, j - 1, memo)

    memo[i][j] = min(deletion, insertion, substitution)
    return memo[i][j]
```

The diffUtilityTopDown function initializes the memoization table and calls the helper function. The actual work is done in the diffUtilityTopDownHelper function, which recursively calculates the edit distance considering deletion, insertion, and substitution operations.
The memo table is used to store already computed results to avoid redundant calculations.

Without Recursion

```
function diffUtilityTopDown(str1, str2):
    m = length of str1
    n = length of str2
    memo = 2D array of size (m + 1) x (n + 1), initialized with -1
```

```
    // Base case: Initialization
    for i from 0 to m:
        memo[i][0] = i
    for j from 0 to n:
        memo[0][j] = j

    // Fill the memoization table using top-down dynamic programming
    for i from 1 to m:
        for j from 1 to n:
            // If the last characters are equal, no operation is needed
            if str1[i - 1] equals str2[j - 1]:
                memo[i][j] = memo[i - 1][j - 1]
            else:
                // Calculate the minimum of three operations: deletion, insertion,
and substitution
                deletion = 1 + memo[i - 1][j]
                insertion = 1 + memo[i][j - 1]
                substitution = 1 + memo[i - 1][j - 1]
                memo[i][j] = min(deletion, insertion, substitution)

    return memo[m][n]
```

3. Find the size of the largest square submatrix of 1's present in a binary matrix

Given an M × N binary matrix, find the size of the largest square submatrix of 1's present. For example, the size of the largest square submatrix of 1's is 3 in the following matrix:

```
  0   0   1   0   1   1
  0   1   1   1   0   0
  0   0   1   1   1   1
  1   1   0   1   1   1
  1   1   1   1   1   1
  1   1   0   1   1   1
  1   0   1   1   1   1
  1   1   1   0   1   1
```

The idea is to use dynamic programming to solve this problem. The problem has optimal substructure. The size of the largest square submatrix ending at a cell M[i][j] will be 1 plus the

minimum among the largest square submatrix ending at M[i][j-1], M[i-1][j] and M[i-1][j-1]. The result will be the maximum of all square submatrix ending at M[i][j] for all possible values of i and j.

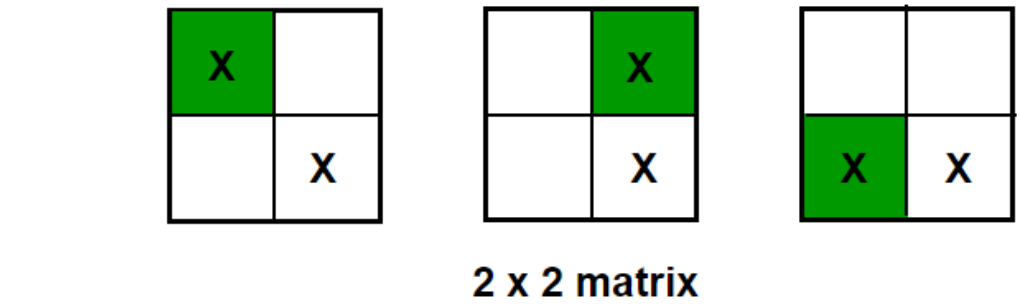
How does this works?

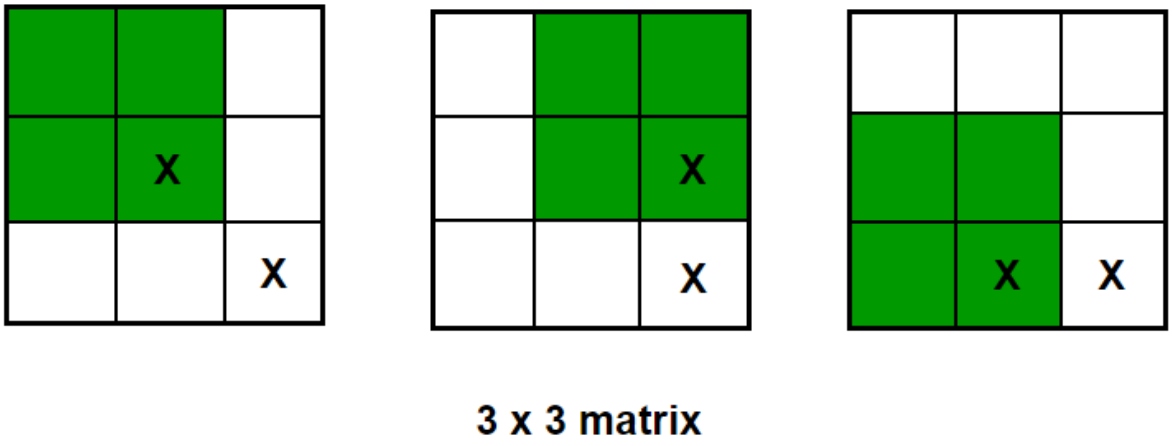
Let's consider any 2×2 matrix. For it to be a 2×2 matrix, each of the top, left, and top-left neighbor of its bottom-right corner has to be a 1×1 square matrix.

Similarly, for a 3×3 matrix, each top, left, and top-left neighbor of its bottom-right corner has to be a 2×2 square matrix.

In general, for any n × n square matrix, each of its neighbors at the top left and top-left corner should at least have the size of (n-1) × (n-1). The reverse of this statement is also true. If the size of the square submatrix ending at top, left, and top-left neighbors of any cell in the given matrix is at least n-1, then we can get n × n submatrix from that cell. That is the reason behind picking up the smallest neighboring square and adding 1 to it.

The following figure might help in visualizing things better:





The recursive solution exhibits overlapping subproblems. If we draw the solution's recursion tree, we can see that the same subproblems are repeatedly computed. We know that problems with optimal substructure and overlapping subproblems can be solved using dynamic programming, in which subproblem solutions are memoized rather than computed repeatedly. The memoized version follows the top-down approach since we first break the problem into subproblems and then calculate and store values. We can also solve this problem in a bottom-up manner. In the bottom-up approach, we solve smaller subproblems first, then solve larger subproblems from them.

```
function largestSquareSubmatrix(matrix):
    rows = number of rows in matrix
    cols = number of columns in matrix
    memo = 2D array of size (rows + 1) x (cols + 1), initialized with -1
    maxSquareSize = 0

    // Base case: The largest square size ending at the matrix boundary is 0
    for i from 0 to rows:
        memo[i][0] = 0
    for j from 0 to cols:
        memo[0][j] = 0

    // Fill the memoization table using top-down dynamic programming
    for i from 1 to rows:
        for j from 1 to cols:
            // If the current cell is 1, calculate the largest square size
            if matrix[i-1][j-1] == 1:
                memo[i][j] = 1 + min(memo[i-1][j], memo[i][j-1], memo[i-1][j-1])
                maxSquareSize = max(maxSquareSize, memo[i][j])
            else:
                memo[i][j] = 0  // If the current cell is 0, the largest square size
is 0

    return maxSquareSize
```

- matrix is the binary matrix.
- memo is a 2D array used for memoization.
- The largestSquareSubmatrix function initializes the memoization table and iteratively fills it using a nested loop.
- The memo[i][j] represents the size of the largest square submatrix ending at position (i, j) in the original matrix.
- maxSquareSize keeps track of the overall maximum square size encountered during the process.

- The min(memo[i-1][j], memo[i][j-1], memo[i-1][j-1]) calculates the size of the largest square ending at the current position based on the sizes of the squares ending at its three adjacent positions.

This algorithm has a time complexity of O(rows * cols) as it processes each cell in the matrix only once.

bottom -up

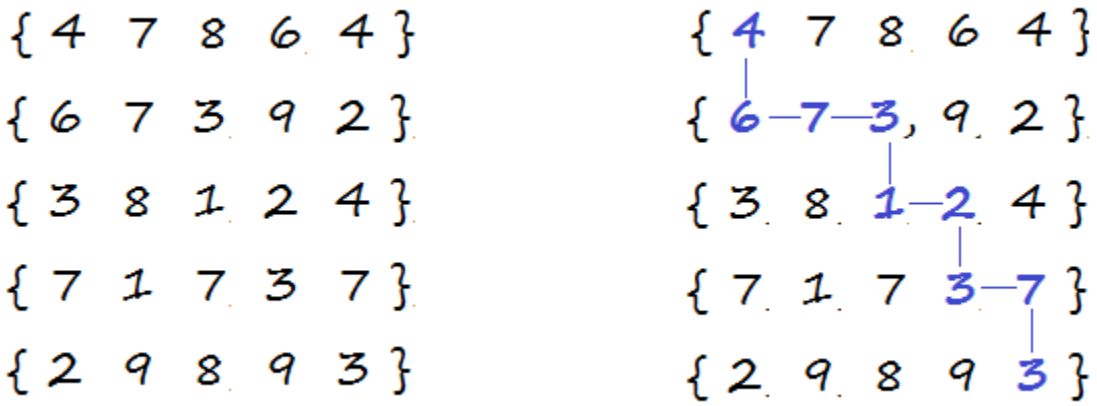
```
function largestSquareSubmatrix(matrix):
    rows = number of rows in matrix
    cols = number of columns in matrix
    dp = 2D array of size (rows + 1) x (cols + 1), initialized with 0
    maxSquareSize = 0

    // Fill the DP table using bottom-up dynamic programming
    for i from 1 to rows:
        for j from 1 to cols:
            // If the current cell is 1, calculate the largest square size
            if matrix[i-1][j-1] == 1:
                dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
                maxSquareSize = max(maxSquareSize, dp[i][j])

    return maxSquareSize
```

- matrix is the binary matrix.
- dp is a 2D array used for dynamic programming.
- The largestSquareSubmatrix function iteratively fills the dp table using a nested loop.
- The dp[i][j] represents the size of the largest square submatrix ending at position (i, j) in the original matrix.
- maxSquareSize keeps track of the overall maximum square size encountered during the process.
- The min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) calculates the size of the largest square ending at the current position based on the sizes of the squares ending at its three adjacent positions.

  This bottom-up approach starts from the smaller subproblems and builds up the solution for larger submatrices until the entire matrix is processed. The time complexity of this algorithm is also O(rows * cols).

4. Find minimum cost to reach the last cell of a matrix from its first cell

Given an M × N matrix of integers where each cell has a cost associated with it, find the minimum cost to reach the last cell (M-1, N-1) of the matrix from its first cell (0, 0). We can only move one unit right or one unit down from any cell, i.e., from cell (i, j), we can move to (i, j+1) or (i+1, j).



The idea is to use recursion. The problem has optimal substructure. That means the problem can be broken down into smaller, simple "subproblems", which can further be divided into yet simpler, smaller subproblems until the solution becomes trivial. We can recursively define the problem as:

Cost to reach cell (m, n) = cost[m][n] + min (cost to reach cell (m, n-1),
cost to reach cell (m, n-1))

As we can see, the same subproblems (highlighted in the same color) are getting computed repeatedly. As the recursion grows deeper, more and more of this type of unnecessary repetition occurs. We know that problems having optimal substructure and overlapping subproblems can be solved by dynamic programming, in which subproblem solutions are memoized rather than computed repeatedly. Each time we calculate the minimum cost of reaching any cell (i, j), we save it. If we are ever asked to compute it again, give the saved answer and do not recompute it.

The following bottom-up approach computes, for each 0 <= i < M and 0 <= j < N, the minimum costs of reaching cell (i, j) from cell (0, 0), using the costs of smaller values i and j already computed. It has the same asymptotic runtime as Memoization but no recursion overhead.

```
function minCostToReachLastCell(matrix):
    rows = number of rows in matrix
    cols = number of columns in matrix
    memo = 2D array of size (rows + 1) x (cols + 1), initialized with -1
```

```
    // Base case: The cost to reach the first cell is the value in the first cell
    memo[0][0] = matrix[0][0]

    // Initialize the first row and first column in the memo table
    for i from 1 to rows:
        memo[i][0] = infinity
    for j from 1 to cols:
        memo[0][j] = infinity

    // Call the recursive helper function to calculate the minimum cost
    return minCostToReachLastCellHelper(matrix, rows-1, cols-1, memo)

function minCostToReachLastCellHelper(matrix, i, j, memo):
    // Check if the result is already memoized
    if memo[i][j] ≠ -1:
        return memo[i][j]

    // Calculate the cost to reach the current cell
    cost = matrix[i][j] + min(
        minCostToReachLastCellHelper(matrix, i-1, j, memo),  // Move up
        minCostToReachLastCellHelper(matrix, i, j-1, memo)   // Move left
    )

    memo[i][j] = cost  // Memoize the result
    return cost
```

- matrix is the input matrix representing the costs at each cell.
- memo is a 2D array used for memoization.
- The minCostToReachLastCell function initializes the memoization table and calls the recursive helper function.
- The minCostToReachLastCellHelper function recursively calculates the minimum cost to reach the last cell by considering the minimum cost of reaching the cell above and the cell to the left.
- The base case initializes the cost to reach the first cell, and the first row and first column in the memo table are initialized to infinity to ensure they are properly updated during the recursive calls.

5. 0–1 Knapsack Problem

In the 0–1 Knapsack problem, we are given a set of items, each with a weight and a value, and we need to determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Input:

```
value = [ 20, 5, 10, 40, 15, 25 ]
weight = [ 1, 2, 3, 8, 7, 4 ]
int W = 10
```

Output: Knapsack value is 60

```
value = 20 + 40 = 60
weight = 1 + 8 = 9 < W
```

The idea is to use recursion to solve this problem. For each item, there are two possibilities:

1. Include the current item in the knapsack and recur for remaining items with knapsack's decreased capacity. If the capacity becomes negative, do not recur or return -INFINITY.
2. Exclude the current item from the knapsack and recur for the remaining items.

Finally, return the maximum value we get by including or excluding the current item. The base case of the recursion would be when no items are left, or capacity becomes 0.

The above solution has an optimal substructure, i.e., the optimal solution can be constructed efficiently from optimal solutions of its subproblem. It also has overlapping subproblems, i.e., the problem can be broken down into subproblems, and each subproblem is repeated several times. To reuse the subproblem solutions, we can apply dynamic programming, in which subproblem solutions are memoized rather than computed over and over again.

```
function knapsack(weights, values, capacity):
    n = length of weights
    memo = 2D array of size (n + 1) x (capacity + 1), initialized with zeros

    // Build the memoization table bottom-up
    for i from 1 to n:
        for w from 1 to capacity:
            if weights[i-1] > w:
                memo[i][w] = memo[i-1][w]  // Current item doesn't fit in the
knapsack
            else:
```

```
                // Choose the maximum value between including and excluding the
current item
                memo[i][w] = max(memo[i-1][w], values[i-1] + memo[i-1][w -
weights[i-1]])

    return memo[n][capacity]  // Maximum value that can be obtained
```

- `weights` is an array representing the weights of the items.
- `values` is an array representing the values of the items.
- `capacity` is the maximum weight the knapsack can hold.
- `n` is the number of items.
- `memo` is a 2D array used for memoization.

The idea is to fill the memo table iteratively, considering each item and each possible weight capacity. The value at memo[i][w] represents the maximum value that can be obtained by considering the first i items and having a knapsack capacity of w.

The recurrence relation is defined as follows:

- If the weight of the current item is greater than the current capacity, we can't include the item, so we inherit the value from the cell above: `memo[i][w] = memo[i-1][w]`.
- Otherwise, we choose the maximum value between including and excluding the current item: `memo[i][w] = max(memo[i-1][w], values[i-1] + memo[i-1][w - weights[i-1]])`.

Finally, the result is stored in memo[n][capacity], representing the maximum value that can be obtained considering all items and the given knapsack capacity.

Partition Problem using Dynamic Programming

Given a set of positive integers, check if it can be divided into two subsets with equal sum.

```
Consider S = {3, 1, 1, 2, 2, 1}

We can partition  S  into two partitions, each having a sum of 5.

S1 = {1, 1, 1, 2}
S2 = {2, 3}

Note that this solution is not unique. Here's another solution.

S1 = {3, 1, 1}
S2 = {2, 2, 1}
```

The partition problem is a special case of the **Subset Sum Problem**, which itself is a special case of the **Knapsack Problem**. The idea is to calculate the sum of all elements in the set, say sum. If sum is odd, we can't divide the array into two sets. If sum is even, check if a subset with sum/2 exists or not. Following is the algorithm to find the subset sum:
Consider each item in the given array one by one, and for each item, there are two possibilities:

1. Include the current item in the subset and recur for the remaining items with the remaining total.
2. Exclude the current item from the subset and recur for the remaining items.

Finally, return true if we get a subset by including or excluding the current item; otherwise, return false. The recursion's base case would be when no items are left, or the sum becomes negative. We return true when the sum becomes 0, i.e., the subset is found.
The problem has an optimal substructure and also exhibits overlapping subproblems, i.e., the problem can be split into smaller subproblems, and the same subproblems will get computed again and again. We can easily prove this by drawing a recursion tree of the above code.

Dynamic programming can solve this problem by saving subproblem solutions in memory rather than computing them again and again. The idea is to solve smaller subproblems first, then solve larger subproblems from them. The following bottom-up approach computes T[i][j], for each 1 <= i <= n and 1 <= j <= sum, which is true if subset with sum j can be found using items up to first i items. It uses the value of smaller values i and j already computed. It has the same asymptotic runtime as Memoization but no recursion overhead.

```
function canPartition(nums):
    totalSum = sum(nums)

    // If the total sum is odd, it can't be divided into two equal subsets
    if totalSum % 2 != 0:
        return false

    targetSum = totalSum / 2
    n = length of nums

    // Create a 2D array to store the state of subproblems
    dp = 2D array of size (n + 1) x (targetSum + 1), initialized with false

    // Base case: An empty subset can always achieve a sum of 0
    for i from 0 to n:
        dp[i][0] = true

    // Fill the DP table using bottom-up dynamic programming
    for i from 1 to n:
        for j from 1 to targetSum:
            // If the current number is greater than the target sum, exclude it
            if nums[i-1] > j:
                dp[i][j] = dp[i-1][j]
```

```
            else:
                // Include or exclude the current number to achieve the target sum
                dp[i][j] = dp[i-1][j] or dp[i-1][j - nums[i-1]]

    return dp[n][targetSum]
```

- nums is the array of positive integers.
- totalSum is the sum of all elements in the array.
- targetSum is the target sum that each subset needs to achieve.
- dp is a 2D array used for dynamic programming.

The algorithm checks if the total sum is even (indicating the possibility of dividing into two equal subsets) and then proceeds to fill the DP table. The state dp[i][j] represents whether it's possible to achieve a sum of j using the first i elements of the array.

The recurrence relation is defined as follows:

- If the current number is greater than the target sum, exclude it: dp[i][j] = dp[i-1][j].
- Otherwise, include or exclude the current number to achieve the target sum: dp[i][j] = dp[i-1][j] or dp[i-1][j - nums[i-1]].

The final result is dp[n][targetSum], which indicates whether it's possible to divide the set into two subsets with equal sum.

https://medium.com/techie-delight/top-50-dynamic-programming-practice-problems-4208fed71aa3