DYNAMIC PROGRAMMING

This method of programming is generally used for optimisation problems just like greedy.In Dynamic Programming all possible solutions are considered and the best solution is taken unlike greedy method.Dynamic Programming is mainly an optimization over plain recursion.

Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.
Do try tracing the algorithms and Problems explained here to understand better how Dynamic Programming works.

Consider the following example where we shall find the nth fibonacci number.
Code using Recursion:

```
int fibo(int n)
{
      if(n<=1)
      {
      return n;
      }
      return fibo(n-2)+fibo(n-1);
}
```

Consider the above function is given an input of n=5.When the code starts executing, some functions are called again and again thus consuming more time. For example the function **fibo(1)**,**fibo(2)** are called multiple times and thus cosue more time. This could be prevented by storing the value after one function call and use the

stored value for further same function calls. This is where the concept of Dynamic Programming helps us.

Code Using Dynamic Programming:

```
int fibo(int n)
{
      f[0]=0;
      f[1]=1;
      for(int i=2;i<=n;i++)
      {
      f[i]=f[i-1]+f[i-2];
      }
      return f[n];
}
```

Here consider that f is an array and f[i] stores the ith fibonacci number.
Hence using the array f , we can store the ith fibonacci number and use it for calculating the i+1th and i+2th fibonacci number without the need to call fibo(i) again.
Thus here Dynamic Programming has helped us in decreasing the time complexity from exponential to linear.

We shall now see some standard dynamic programming problems that might help in solving many related problems.

COIN CHANGE PROBLEM

The problem statement is as follows:
Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn't matter.
For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

Incase the problem is approached using recursion, the code would look like this:

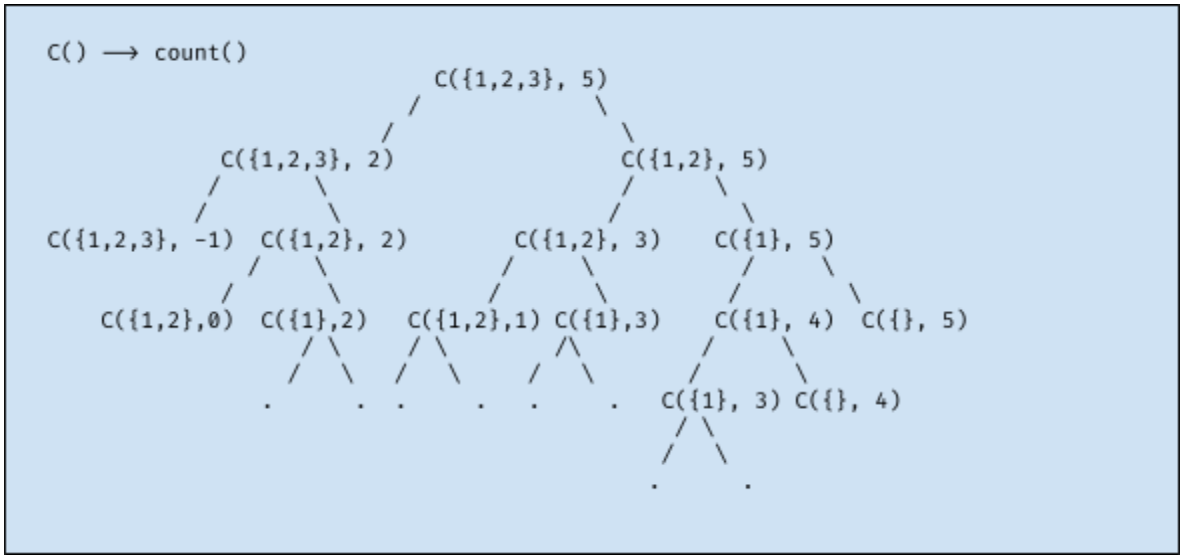
```
int count( int S[], int m, int n )
{
    // If n is 0 then there is 1 solution
    // (do not include any coin)
    if (n == 0)
        return 1;

    // If n is less than 0 then no
    // solution exists
    if (n < 0)
        return 0;

    // If there are no coins and n
    // is greater than 0, then no
    // solution exist
    if (m <= 0 && n >= 1)
        return 0;

    // count is sum of solutions (i)
    // including S[m-1] (ii) excluding S[m-1]
    return count( S, m - 1, n ) + count( S, m, n-S[m-1] );
}
```

Consider the example N=5 and S={1,2,3}. The recursion tree would look like the below image where in the function solves the same problem multiple times.The function C({1}, 3) is called two times. If we draw the complete tree, then we can see that there are many subproblems being called more than once.



Approach Using Dynamic Programming:
Hence we need to prevent multiple function calls of the same sub problem again.Hence we shall construct a matrix int M[n+1][m] where n is the amount and m is the number of types of coins i the array. As a base case,when the amount available with us is 0, the solution for this subproblem would be 1 i.e to not include any coin. Then for a given amount **i** available with us at some coin S[j] we have two options:
1)Include the current coin S[j] and recur with remaining change i-S[j] with same number of coins
2)Exclude the current count S[j] and recur for the remaining coins.

After filling the entire matrix, as the required amount for which the answer is needed is n, M[n][m-1] would yield the required answer.

The code for the above problem would be as follows:

```
int count( int S[], int m, int n )
{
    int i, j, x, y;

    // We need n+1 rows as the table
    // is constructed in bottom up
    // manner using the base case 0
    // value case (n = 0)
    int table[n + 1][m];

    // Fill the enteries for 0
    // value case (n = 0)
    for (i = 0; i < m; i++)
        table[0][i] = 1;

    for (i = 1; i < n + 1; i++)
    {
        for (j = 0; j < m; j++)
        {
            // Count of solutions including S[j]
            x = (i-S[j] ≥ 0) ? table[i - S[j]][j] : 0;

            // Count of solutions excluding S[j]
            y = (j ≥ 1) ? table[i][j - 1] : 0;

            // total count
            table[i][j] = x + y;
        }
    }
    return table[n][m - 1];
}
```

LONGEST COMMON SUBSEQUENCE

The problem statement is as follows:

Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”.

Approach to solve the Problem:

Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y. Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).

If last characters of both sequences match (or X[m-1] == Y[n-1]) then
L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

If last characters of both sequences do not match (or X[m-1] ≠ Y[n-1]) then

L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))

Approach to solve Problem Using Dynamic Programming:

We shall maintain a matrix L of dimensions int L[m+1][n+1] where L[i][j] denotes the length of the longest subsequence for the strings X[0..i-1] and Y[0...j-1]. Hence once the matrix is filled, L[m][n] would be the required answer.

Combining both the above two approaches to construct a solution would give the following code:

```
int lcs( string X, string Y, int m, int n )
{
    int L[m + 1][n + 1];
    int i, j;
    for (i = 0; i ⩽ m; i++)
    {
        for (j = 0; j ⩽ n; j++)
        {
        if (i == 0 || j == 0)
            L[i][j] = 0;

        else if (X[i - 1] == Y[j - 1])
            L[i][j] = L[i - 1][j - 1] + 1;

        else
            L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }

    /* L[m][n] contains length of LCS
    for X[0..n-1] and Y[0..m-1] */
    return L[m][n];
}
```

0/1 KNAPSACK PROBLEM

The problem statement is as follows:

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).

The below is a example:

```
value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220
```

```
Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50
```

Here while encountering every item in the set of items available, we have two options:

1) The item is a part of the final solution/optimal set
2) The item is not a part of the final solution/optimal set

To choose between the two given options, we check the corresponding value if the item included or not included and the maximum one among the two is considered into the solution.

Hence we shall use the a matrix K[n+1][W+1] where K[i][j] denotes the maximum value until the ith when the available weight is j.

Incase the item is not included in the solution then value corresponding to that would be K[i-1][j] and if included then the value corresponding to that would be val[i-1]+K[i-1][j-W[i-1]].The maximum of these both values is taken as the value for K[i][j]. When the no of weights available or the weight available is 0, then the max value corresponding to it also would be 0(base case).

Using the explanation above, we would get the following code:

```
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[n + 1][W + 1];

    // Build table K[][] in bottom up manner
    for(i = 0; i ⩽ n; i++)
    {
        for(w = 0; w ⩽ W; w++)
        {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] ⩽ w)
                K[i][w] = max(val[i - 1] +
                              K[i - 1][w - wt[i - 1]],
                              K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }
    return K[n][W];
}
```

FURTHER READINGS:

- https://www.geeksforgeeks.org/dynamic-programming/#concepts
- https://www.youtube.com/watch?v=5dRGRueKU3M
- https://www.geeksforgeeks.org/solve-dynamic-programming-problem/