**Dynamic Programming and Recursion**

**The intuition behind dynamic programming is that we trade space for time**, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Let's try to understand this by taking an example of Fibonacci numbers.

Fibonacci (n) = 1; if n = 0  
Fibonacci (n) = 1; if n = 1  
Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: **1, 1, 2, 3, 5, 8, 13, 21...** and so on!

A code for it using pure recursion:

int fib (int n) {

if (n < 2)

return 1;

return fib(n-1) + fib(n-2);

}

Using Dynamic Programming approach with memoization:

void fib () {

fibresult[0] = 1;

fibresult[1] = 1;

for (int i = 2; i<n; i++)

fibresult[i] = fibresult[i-1] + fibresult[i-2];

}

Majority of the Dynamic Programming problems can be categorized into two types:

**1. Optimization problems.  
2. Combinatorial problems.**

The optimization problems expect you to select a feasible solution, so that the value of the required function is minimized or maximized. Combinatorial problems expect you to figure out the number of ways to do something, or the probability of some event happening.

Every Dynamic Programming problem has a schema to be followed:

* Show that the problem can be broken down into optimal sub-problems.
* Recursively define the value of the solution by expressing it in terms of optimal solutions for smaller sub-problems.
* Compute the value of the optimal solution in bottom-up fashion.
* Construct an optimal solution from the computed information.

**Bottom up vs. Top Down:**

* **Bottom Up** - I'm going to learn programming. Then, I will start practicing. Then, I will start taking part in contests. Then, I'll practice even more and try to improve. After working hard like crazy, I'll be an amazing coder.
* **Top Down** - I will be an amazing coder. How? I will work hard like crazy. How? I'll practice more and try to improve. How? I'll start taking part in contests. Then? I'll practicing. How? I'm going to learn programming.

Not a great example, but I hope I got my point across. In Top Down, you start building the big solution right away by explaining how you build it from smaller solutions. In Bottom Up, you start with the small solutions and then build up.

Memoization is very easy to code and might be your first line of approach for a while. Though, with dynamic programming, you don't risk blowing stack space, you end up with lots of liberty of when you can throw calculations away. The downside is that you have to come up with an ordering of a solution which works.

**One can think of dynamic programming as a table-filling algorithm: you know the calculations you have to do, so you pick the best order to do them in and ignore the ones you don't have to fill in.**

Let's look at a sample problem:

Let us say that you are given a number **N,** you've to find the number of different ways to write it as the sum of 1, 3 and 4.

For example, if N = 5, the answer would be 6.

* 1 + 1 + 1 + 1 + 1
* 1 + 4
* 4 + 1
* 1 + 1 + 3
* 1 + 3 + 1
* 3 + 1 + 1

**Sub-problem:** DPn be the number of ways to write **N** as the sum of 1, 3, and 4.  
**Finding recurrence:** Consider one possible solution, n = x1 + x2 + ... xn. If the last number is 1, the sum of the remaining numbers should be n - 1. So, number of sums that end with 1 is equal to DPn-1.. Take other cases into account where the last number is 3 and 4. The final recurrence would be:

DPn = DPn-1 + DPn-3 + DPn-4.

Take care of the base cases. DP0 = DP1 = DP2 = 1, and DP3 = 2.

Implementation:

DP[0] = DP[1] = DP[2] = 1; DP[3] = 2;

for (i = 4; i <= n; i++) {

DP[i] = DP[i-1] + DP[i-3] + DP[i-4];

}

The technique above, takes a bottom up approach and uses memoization to not compute results that have already been computed.

Dynamic Programming | Set 13 (Cutting a Rod):-

// A Dynamic Programming solution for Rod cutting problem

#include<stdio.h>

#include<limits.h>

// A utility function to get the maximum of two integers

int max(int a, int b) { return (a > b)? a : b;}

/\* Returns the best obtainable price for a rod of length n and

   price[] as prices of different pieces \*/

int cutRod(int price[], int n)

{

   int val[n+1];

   val[0] = 0;

   int i, j;

   // Build the table val[] in bottom up manner and return the last entry

   // from the table

   for (i = 1; i<=n; i++)

   {

       int max\_val = INT\_MIN;

       for (j = 0; j < i; j++)

         max\_val = max(max\_val, price[j] + val[i-j-1]);

       val[i] = max\_val;

   }

   return val[n];

}

/\* Driver program to test above functions \*/

int main()

{

    int arr[] = {1, 5, 8, 9, 10, 17, 17, 20};

    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Maximum Obtainable Value is %dn", cutRod(arr, size));

    getchar();

    return 0;

}

**Dynamic Programming | Set 7 (Coin Change)**

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.

**1) Optimal Substructure**  
To count total number solutions, we can divide all set solutions in two sets.  
1) Solutions that do not contain mth coin (or Sm).  
2) Solutions that contain at least one Sm.  
Let count(S[], m, n) be the function to count the number of solutions, then it can be written as sum of count(S[], m-1, n) and count(S[], m, n-Sm).

Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.

**2) Overlapping Subproblems**  
Following is a simple recursive implementation of the Coin Change problem. The implementation simply follows the recursive structure mentioned above.

|  |
| --- |
| // Recursive C program for  // coin change problem.  #include<stdio.h>    // Returns the count of ways we can  // sum S[0...m-1] coins to get sum n  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] );  }    // Driver program to test above function  int main()  {      int i, j;      int arr[] = {1, 2, 3};      int m = sizeof(arr)/sizeof(arr[0]);      printf("%d ", count(arr, m, 4));      getchar();      return 0;  } |

**Dynamic Programming Solution**

|  |
| --- |
| #include<stdio.h>    int count( int S[], int m, int n )  {      int i, j, x, y;        // We need n+1 rows as the table is consturcted 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;        // Fill rest of the table enteries in bottom up manner      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];  }    // Driver program to test above function  int main()  {      int arr[] = {1, 2, 3};      int m = sizeof(arr)/sizeof(arr[0]);      int n = 4;      printf(" %d ", count(arr, m, n));      return 0;  } |

Output:

4

Time Complexity: O(mn)

Following is a simplified version of method 2. The auxiliary space required here is O(n) only.

|  |
| --- |
| int count( int S[], int m, int n )  {      // table[i] will be storing the number of solutions for      // value i. We need n+1 rows as the table is consturcted      // in bottom up manner using the base case (n = 0)      int table[n+1];        // Initialize all table values as 0      memset(table, 0, sizeof(table));        // Base case (If given value is 0)      table[0] = 1;        // Pick all coins one by one and update the table[] values      // after the index greater than or equal to the value of the      // picked coin      for(int i=0; i<m; i++)          for(int j=S[i]; j<=n; j++)              table[j] += table[j-S[i]];        return table[n];  } |

**Dynamic Programming | Set 8 (Matrix Chain Multiplication)**

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ....

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Clearly the first parenthesization requires less number of operations.

*Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.*

**Input: p[] = {40, 20, 30, 10, 30}**

**Output: 26000**

There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.

Let the input 4 matrices be A, B, C and D. The minimum number of

multiplications are obtained by putting parenthesis in following way

(A(BC))D --> 20\*30\*10 + 40\*20\*10 + 40\*10\*30

**Input: p[] = {10, 20, 30, 40, 30}**

**Output: 30000**

There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.

Let the input 4 matrices be A, B, C and D. The minimum number of

multiplications are obtained by putting parenthesis in following way

((AB)C)D --> 10\*20\*30 + 10\*30\*40 + 10\*40\*30

**Input: p[] = {10, 20, 30}**

**Output: 6000**

There are only two matrices of dimensions 10x20 and 20x30. So there

is only one way to multiply the matrices, cost of which is 10\*20\*30

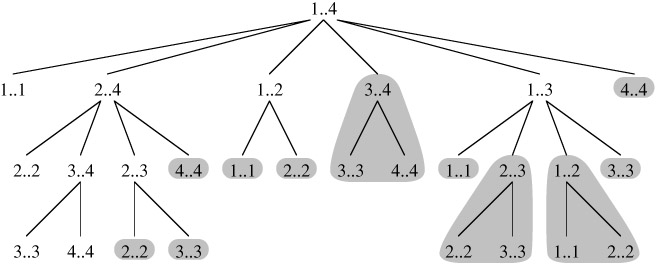
**1) Optimal Substructure:**  
A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n, we can place the first set of parenthesis in n-1 ways. For example, if the given chain is of 4 matrices. let the chain be ABCD, then there are 3 ways to place first set of parenthesis outer side: (A)(BCD), (AB)(CD) and (ABC)(D). So when we place a set of parenthesis, we divide the problem into subproblems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.

Minimum number of multiplication needed to multiply a chain of size n = Minimum of all n-1 placements (these placements create subproblems of smaller size)

**2) Overlapping Subproblems**  
Following is a recursive implementation that simply follows the above optimal substructure property.

|  |
| --- |
| /\* A naive recursive implementation that simply    follows the above optimal substructure property \*/  #include<stdio.h>  #include<limits.h>    // Matrix Ai has dimension p[i-1] x p[i] for i = 1..n  int MatrixChainOrder(int p[], int i, int j)  {      if(i == j)          return 0;      int k;      int min = INT\_MAX;      int count;        // place parenthesis at different places between first      // and last matrix, recursively calculate count of      // multiplications for each parenthesis placement and      // return the minimum count      for (k = i; k <j; k++)      {          count = MatrixChainOrder(p, i, k) +                  MatrixChainOrder(p, k+1, j) +                  p[i-1]\*p[k]\*p[j];            if (count < min)              min = count;      }        // Return minimum count      return min;  }    // Driver program to test above function  int main()  {      int arr[] = {1, 2, 3, 4, 3};      int n = sizeof(arr)/sizeof(arr[0]);        printf("Minimum number of multiplications is %d ",                            MatrixChainOrder(arr, 1, n-1));        getchar();      return 0;  } |

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for a matrix chain of size 4. The function MatrixChainOrder(p, 3, 4) is called two times. We can see that there are many subproblems being called more than once.



Since same suproblems are called again, this problem has Overlapping Subprolems property. So Matrix Chain Multiplication problem has both properties of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array m[][] in bottom up manner.

**Dynamic Programming Solution**  
Following is C/C++ implementation for Matrix Chain Multiplication problem using Dynamic Programming.

|  |
| --- |
| // See the Cormen book for details of the following algorithm  #include<stdio.h>  #include<limits.h>    // Matrix Ai has dimension p[i-1] x p[i] for i = 1..n  int MatrixChainOrder(int p[], int n)  {        /\* For simplicity of the program, one extra row and one         extra column are allocated in m[][].  0th row and 0th         column of m[][] are not used \*/      int m[n][n];        int i, j, k, L, q;        /\* m[i,j] = Minimum number of scalar multiplications needed         to compute the matrix A[i]A[i+1]...A[j] = A[i..j] where         dimension of A[i] is p[i-1] x p[i] \*/        // cost is zero when multiplying one matrix.      for (i=1; i<n; i++)          m[i][i] = 0;        // L is chain length.      for (L=2; L<n; L++)      {          for (i=1; i<n-L+1; i++)          {              j = i+L-1;              m[i][j] = INT\_MAX;              for (k=i; k<=j-1; k++)              {                  // q = cost/scalar multiplications                  q = m[i][k] + m[k+1][j] + p[i-1]\*p[k]\*p[j];                  if (q < m[i][j])                      m[i][j] = q;              }          }      }        return m[1][n-1];  }    int main()  {      int arr[] = {1, 2, 3, 4};      int size = sizeof(arr)/sizeof(arr[0]);        printf("Minimum number of multiplications is %d ",                         MatrixChainOrder(arr, size));        getchar();      return 0;  } |

Output:

Minimum number of multiplications is 18

Time Complexity: O(n^3)  
Auxiliary Space: O(n^2)

**Dynamic Programming | Set 4 (Longest Common Subsequence)**

We have discussed Overlapping Subproblems and Optimal Substructure properties in [Set 1](https://www.geeksforgeeks.org/?p=12635) and [Set 2](https://www.geeksforgeeks.org/?p=12819) respectively. We also discussed one example problem in [Set 3](https://www.geeksforgeeks.org/?p=12832). Let us discuss Longest Common Subsequence (LCS) problem as one more example problem that can be solved using Dynamic Programming.

*LCS Problem Statement:* 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”. So a string of length n has 2^n different possible subsequences.

It is a classic computer science problem, the basis of [diff](http://en.wikipedia.org/wiki/Diff) (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

**Examples:**  
LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.  
LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

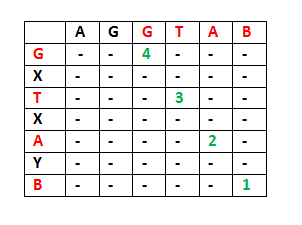
The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

**1) Optimal Substructure:**   
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])

Examples:  
1) Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as:  
L(“AGGTAB”, “GXTXAYB”) = 1 + L(“AGGTA”, “GXTXAY”)



2) Consider the input strings “ABCDGH” and “AEDFHR. Last characters do not match for the strings. So length of LCS can be written as:  
L(“ABCDGH”, “AEDFHR”) = MAX ( L(“ABCDG”, “AEDFH**R**”), L(“ABCDG**H**”, “AEDFH”) )

So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

**2) Overlapping Subproblems:**  
Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

|  |
| --- |
| /\* A Naive recursive implementation of LCS problem \*/  #include<bits/stdc++.h>    int max(int a, int b);    /\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/  int lcs( char \*X, char \*Y, int m, int n )  {     if (m == 0 || n == 0)       return 0;     if (X[m-1] == Y[n-1])       return 1 + lcs(X, Y, m-1, n-1);     else       return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));  }    /\* Utility function to get max of 2 integers \*/  int max(int a, int b)  {      return (a > b)? a : b;  }    /\* Driver program to test above function \*/  int main()  {    char X[] = "AGGTAB";    char Y[] = "GXTXAYB";      int m = strlen(X);    int n = strlen(Y);      printf("Length of LCS is %dn", lcs( X, Y, m, n ) );      return 0;  } |

Output:

Length of LCS is 4

Time complexity of the above naive recursive approach is O(2^n) in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0.  
Considering the above implementation, following is a partial recursion tree for input strings “AXYT” and “AYZX”

lcs("AXYT", "AYZX")

/

lcs("AXY", "AYZX") lcs("AXYT", "AYZ")

/ /

lcs("AX", "AYZX") lcs("AXY", "AYZ") lcs("AXY", "AYZ") lcs("AXYT", "AY")

In the above partial recursion tree, lcs(“AXY”, “AYZ”) is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabulated implementation for the LCS problem.

|  |
| --- |
| /\* Dynamic Programming C/C++ implementation of LCS problem \*/  #include<bits/stdc++.h>    int max(int a, int b);    /\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/  int lcs( char \*X, char \*Y, int m, int n )  {     int L[m+1][n+1];     int i, j;       /\* Following steps build L[m+1][n+1] in bottom up fashion. Note        that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] \*/     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];  }    /\* Utility function to get max of 2 integers \*/  int max(int a, int b)  {      return (a > b)? a : b;  }    /\* Driver program to test above function \*/  int main()  {    char X[] = "AGGTAB";    char Y[] = "GXTXAYB";      int m = strlen(X);    int n = strlen(Y);      printf("Length of LCS is %dn", lcs( X, Y, m, n ) );      return 0;  } |

Time Complexity of the above implementation is O(mn) which is much better than the worst case time complexity of Naive Recursive implementation.

Printing Longest Common Subsequence

/\* Dynamic Programming implementation of LCS problem \*/

/\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/

void lcs( char \*X, char \*Y, int m, int n )

{

   int L[m+1][n+1];

   /\* Following steps build L[m+1][n+1] in bottom up fashion. Note

      that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] \*/

   for (int i=0; i<=m; i++)

   {

     for (int 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]);

     }

   }

   // Following code is used to print LCS

   int index = L[m][n];

   // Create a character array to store the lcs string

   char lcs[index+1];

   lcs[index] = '\0'; // Set the terminating character

   // Start from the right-most-bottom-most corner and

   // one by one store characters in lcs[]

   int i = m, j = n;

   while (i > 0 && j > 0)

   {

      // If current character in X[] and Y are same, then

      // current character is part of LCS

      if (X[i-1] == Y[j-1])

      {

          lcs[index-1] = X[i-1]; // Put current character in result

          i--; j--; index--;     // reduce values of i, j and index

      }

      // If not same, then find the larger of two and

      // go in the direction of larger value

      else if (L[i-1][j] > L[i][j-1])

         i--;

      else

         j--;

   }

   // Print the lcs

   cout << "LCS of " << X << " and " << Y << " is " << lcs;

}

/\* Driver program to test above function \*/

int main()

{

  char X[] = "AGGTAB";

  char Y[] = "GXTXAYB";

  int m = strlen(X);

  int n = strlen(Y);

  lcs(X, Y, m, n);

  return 0;

}

/\* Space optimized C++ implementation of LCS problem \*/

#include<bits/stdc++.h>

using namespace std;

/\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/

int lcs(string &X, string &Y)

{

    // Find lengths of two strings

    int m = X.length(), n = Y.length();

    int L[2][n+1];

    // Binary index, used to index current row and

    // previous row.

    bool bi;

    for (int i=0; i<=m; i++)

    {

        // Compute current binary index

        bi = i&1;

        for (int j=0; j<=n; j++)

        {

            if (i == 0 || j == 0)

                L[bi][j] = 0;

            else if (X[i] == Y[j-1])

                L[bi][j] = L[1-bi][j-1] + 1;

            else

                L[bi][j] = max(L[1-bi][j], L[bi][j-1]);

        }

    }

    /\* Last filled entry contains length of LCS

       for X[0..n-1] and Y[0..m-1] \*/

    return L[bi][n];

}

/\* Driver program to test above function \*/

int main()

{

    string X = "AGGTAB";

    string Y = "GXTXAYB";

    printf("Length of LCS is %d\n", lcs(X, Y));

    return 0;

}