# Dynamic Programming

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

In this post, we will discuss first property (Overlapping Subproblems) in detail. The second property of Dynamic programming is discussed in next post i.e. [Set 2](http://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/).

1. Overlapping Subproblems
2. 2) Optimal Substructure

**1) Overlapping Subproblems:-**Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don’t have to recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, [Binary Search](http://en.wikipedia.org/wiki/Binary_search_algorithm) doesn’t have common subproblems. If we take example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

|  |
| --- |
| /\* simple recursive program for Fibonacci numbers \*/  int fib(int n)  {     if ( n <= 1 )        return n;     return fib(n-1) + fib(n-2);  } |

Recursion tree for execution of fib(5)

fib(5)

/ \

fib(4) fib(3)

/ \ / \

fib(3) fib(2) fib(2) fib(1)

/ \ / \ / \

fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)

/ \

fib(1) fib(0)

We can see that the function f(3) is being called 2 times. If we would have stored the value of f(3), then instead of computing it again, we could have reused the old stored value. There are following two different ways to store the values so that these values can be reused:  
a) Memoization (Top Down)  
b) Tabulation (Bottom Up)

**a) Memoization (Top Down):**The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later.

Following is the memoized version for nth Fibonacci Number.

/\* Java program for Memoized version \*/

public class Fibonacci

{

  final int MAX = 100;

  final int NIL = -1;

  int lookup[] = new int[MAX];

  /\* Function to initialize NIL values in lookup table \*/

  void \_initialize()

  {

    for (int i = 0; i < MAX; i++)

        lookup[i] = NIL;

  }

  /\* function for nth Fibonacci number \*/

  int fib(int n)

  {

    if (lookup[n] == NIL)

    {

      if (n <= 1)

          lookup[n] = n;

      else

          lookup[n] = fib(n-1) + fib(n-2);

    }

    return lookup[n];

  }

  public static void main(String[] args)

  {

    Fibonacci f = new Fibonacci();

    int n = 40;

    f.\_initialize();

    System.out.println("Fibonacci number is" + " " + f.fib(n));

  }

}

**b) Tabulation (Bottom Up):**The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on. So literally, we are building the solutions of subproblems bottom-up. Following is the tabulated version for nth Fibonacci Number.

public class Fibonacci

{

  int fib(int n)

  {

    int f[] = new int[n+1];

    f[0] = 0;

    f[1] = 1;

    for (int i = 2; i <= n; i++)

          f[i] = f[i-1] + f[i-2];

    return f[n];

  }

  public static void main(String[] args)

  {

    Fibonacci f = new Fibonacci();

    int n = 9;

    System.out.println("Fibonacci number is" + " " + f.fib(n));

  }

}

**Output:-** Fibonacci number is 34

Both Tabulated and Memoized store the solutions of subproblems. In Memoized version, table is filled on demand while in Tabulated version, starting from the first entry, all entries are filled one by one. Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. For example, [Memoized solution](https://www.ics.uci.edu/~eppstein/161/960229.html)of the [LCS problem](http://en.wikipedia.org/wiki/Longest_common_subsequence_problem)doesn’t necessarily fill all entries.

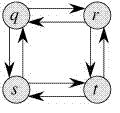
As we discussed in [Set 1](http://www.geeksforgeeks.org/?p=12635), following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming:  
1) Overlapping Subproblems  
2) Optimal Substructure

We have already discussed Overlapping Subproblem property in the [Set 1](http://www.geeksforgeeks.org/?p=12635). Let us discuss Optimal Substructure property here.

**2) Optimal Substructure:-**A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

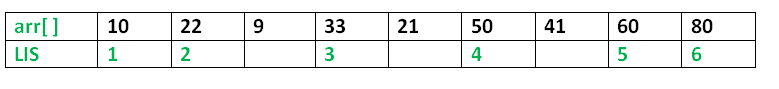
For example, the Shortest Path problem has following optimal substructure property:  
If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithms like [Floyd–Warshall](http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm) and [Bellman–Ford](http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm)are typical examples of Dynamic Programming.

On the other hand, the Longest Path problem doesn’t have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following unweighted graph given in the [CLRS book](http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11866). There are two longest paths from q to t: q→r→t and q→s→t. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q→r→t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q→s→t→r and the longest path from r to t is r→q→s→t



# Longest Increasing Subsequence: - Let us discuss Longest Increasing Subsequence (LIS) problem as an example problem that can be solved using Dynamic Programming.

# The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}.



# More Examples:-

# Input : arr[] = {3, 10, 2, 1, 20}

# Output : Length of LIS = 3

# The longest increasing subsequence is 3, 10, 20

# Input : arr[] = {3, 2}

# Output : Length of LIS = 1

# The longest increasing subsequences are {3} and {2}

# Input : arr[] = {50, 3, 10, 7, 40, 80}

# Output : Length of LIS = 4

# The longest increasing subsequence is {3, 7, 40, 80}

**Optimal Substructure:**  
Let arr[0..n-1] be the input array and L(i) be the length of the LIS ending at index i such that arr[i] is the last element of the LIS.  
Then, L(i) can be recursively written as:  
L(i) = 1 + max( L(j) ) where 0 < j < i and arr[j] < arr[i]; or  
L(i) = 1, if no such j exists.  
To find the LIS for a given array, we need to return max(L(i)) where 0 < i < n.  
Thus, we see the LIS problem satisfies the optimal substructure property as the main problem can be solved using solutions to subproblems.

Following is a simple recursive implementation of the LIS problem. It follows the recursive structure discussed above.

class LIS

{

   static int max\_ref; // stores the LIS

   /\* To make use of recursive calls, this function must return two things:

   1) Length of LIS ending with element arr[n-1]. We use max\_ending\_here for this purpose.

   2) Overall maximum as the LIS may end with an element before arr[n-1] max\_ref is used this purpose.The value of LIS of full array of size n is stored in max\_ref which is our final result \*/

   static int \_lis(int arr[], int n)

   {

       // base case

       if (n == 1)

           return 1;

       // 'max\_ending\_here' is length of LIS ending with arr[n-1]

       int res, max\_ending\_here = 1;

        /\* Recursively get all LIS ending with arr[0], arr[1] ...

        arr[n-2]. If   arr[i-1] is smaller than arr[n-1], and max ending with

arr[n-1] needs to be updated, then update it \*/

        for (int i = 1; i < n; i++)

        {

            res = \_lis(arr, i);

            if (arr[i-1] < arr[n-1] && res + 1 > max\_ending\_here)

                max\_ending\_here = res + 1;

        }

      //Compare max\_ending\_here with the overall max. And update the overall max if needed

        if (max\_ref < max\_ending\_here)

           max\_ref = max\_ending\_here;

        // Return length of LIS ending with arr[n-1]

        return max\_ending\_here;

   }

    // The wrapper function for \_lis()

    static int lis(int arr[], int n)

    {

        // The max variable holds the result

         max\_ref = 1;

        // The function \_lis() stores its result in max

        \_lis( arr, n);

        // returns max

        return max\_ref;

    }

    public static void main(String args[])

    {

        int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };

        int n = arr.length;

        System.out.println("Length of lis is "+ lis(arr, n) + "n");

    }

 }

Length of lis is 5

**Overlapping Subproblems:-** Considering the above implementation, following is recursion tree for an array of size 4. lis(n) gives us the length of LIS for arr[].

lis(4)

/ |

lis(3) lis(2) lis(1)

/ /

lis(2) lis(1) lis(1)

/

lis(1)

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 tabluated implementation for the LIS problem.

class LIS

{

    /\* lis() returns the length of the longest increasing

       subsequence in arr[] of size n \*/

    static int lis(int arr[],int n)

    {

          int lis[] = new int[n];

          int i,j,max = 0;

          /\* Initialize LIS values for all indexes \*/

           for ( i = 0; i < n; i++ )

              lis[i] = 1;

           /\* Compute optimized LIS values in bottom up manner \*/

           for ( i = 1; i < n; i++ )

              for ( j = 0; j < i; j++ )

                         if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)

                    lis[i] = lis[j] + 1;

           /\* Pick maximum of all LIS values \*/

           for ( i = 0; i < n; i++ )

              if ( max < lis[i] )

                 max = lis[i];

            return max;

    }

    public static void main(String args[])

    {

        int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };

            int n = arr.length;

            System.out.println("Length of lis is " + lis( arr, n ) + "n" );

    }

}

**Output**: - Length of lis is 5

Note that the time complexity of the above Dynamic Programming (DP) solution is O(n^2) and there is a O(nLogn) solution for the LIS problem. We have not discussed the O(n Log n) solution here as the purpose of this post is to explain Dynamic Programming with a simple example.

**Longest Common Subsequence :-** *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.

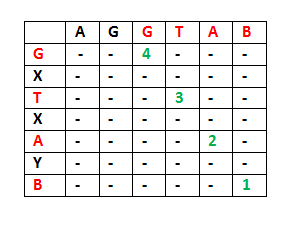
The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

**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”, “AEDFHR”), L(“ABCDGH”, “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 in java\*/

public class LongestCommonSubsequence

{

  /\* 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;

  }

  public static void main(String[] args)

  {

    LongestCommonSubsequence lcs = new LongestCommonSubsequence();

    String s1 = "AGGTAB";

    String s2 = "GXTXAYB";

    char[] X=s1.toCharArray();

    char[] Y=s2.toCharArray();

    int m = X.length;

    int n = Y.length;

    System.out.println("Length of LCS is" + " " +

                                  lcs.lcs( X, Y, m, n ) );

  }

}

**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 Java implementation of LCS problem \*/

public class LongestCommonSubsequence

{

  /\* 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[][] = new int[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]);

      }

    }

  return L[m][n];

  }

  /\* Utility function to get max of 2 integers \*/

  int max(int a, int b)

  {

    return (a > b)? a : b;

  }

  public static void main(String[] args)

  {

    LongestCommonSubsequence lcs = new LongestCommonSubsequence();

    String s1 = "AGGTAB";

    String s2 = "GXTXAYB";

    char[] X=s1.toCharArray();

    char[] Y=s2.toCharArray();

    int m = X.length;

    int n = Y.length;

    System.out.println("Length of LCS is" + " " + lcs.lcs( X, Y, m, n ) );

  }

}

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

Count number of ways to cover a distance:- Given a distance ‘dist, count total number of ways to cover the distance with 1, 2 and 3 steps.

Input: n = 3 Output: 4

Below are the four ways :-

1 step + 1 step + 1 step

1 step + 2 step

2 step + 1 step

3 step

Input: n = 4

Output: 7

class GFG

{

    // Function returns count of ways to cover 'dist'

    static int printCountRec(int dist)

    {

        // Base cases

        if (dist<0)

            return 0;

        if (dist==0)

            return 1;

        // Recur for all previous 3 and add the results

        return printCountRec(dist-1) +

               printCountRec(dist-2) +

               printCountRec(dist-3);

    }

    // driver program

    public static void main (String[] args)

    {

        int dist = 4;

        System.out.println(printCountRec(dist));

    }

}

Output:- 7

The time complexity of above solution is exponential, a close upper bound is O(3n). If we draw the complete recursion tree, we can observer that many subproblems are solved again and again. For example, when we start from n = 6, we can reach 4 by subtracting one 2 times and by subtracting 2 one times. So the subproblem for 4 is called twice.  
Since same suproblems are called again, this problem has Overlapping Subprolems property. So min square sum problem has both properties (see [this](http://www.geeksforgeeks.org/archives/12635)and [this](http://www.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](http://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array count[] in bottom up manner.

// A Dynamic Programming based Java program to count number of ways to cover a distance with 1, 2 and 3 steps

import java.io.\*;

class GFG

{

    // Function returns count of ways to cover 'dist'

    static int printCountDP(int dist)

    {

        int[] count = new int[dist+1];

        // Initialize base values. There is one way to cover 0 and 1 distances and two ways to cover 2 distance

        count[0] = 1;

        count[1] = 1;

        count[2] = 2;

        // Fill the count array in bottom up manner

        for (int i=3; i<=dist; i++)

            count[i] = count[i-1] + count[i-2] + count[i-3];

        return count[dist];

    }

    public static void main (String[] args)

    {

        int dist = 4;

        System.out.println(printCountDP(dist));

    }

}

**Output:-** 7

Asked in: [Amazon](http://practice.geeksforgeeks.org/company/Amazon/)

# Find the longest path in a matrix with given constraints: - Given a n\*n matrix where all numbers are distinct, find the maximum length path (starting from any cell) such that all cells along the path are in increasing order with a difference of 1.

We can move in 4 directions from a given cell (i, j), i.e., we can move to (i+1, j) or (i, j+1) or (i-1, j) or (i, j-1) with the condition that the adjacent cells have a difference of 1. Example:

Input: mat[][] = {{1, 2, 9}

{5, 3, 8}

{4, 6, 7}}

Output: 4 The longest path is 6-7-8-9.

The idea is simple, we calculate longest path beginning with every cell. Once we have computed longest for all cells, we return maximum of all longest paths. One important observation in this approach is many overlapping subproblems. Therefore this problem can be optimally solved using Dynamic Programming.

Below is Dynamic Programming based implementation that uses a lookup table dp[][] to check if a problem is already solved or not.

// Java program to find the longest path in a matrix with given constraints

class GFG

{

    public static int n = 3;

    // Function that returns length of the longest path beginning with mat[i][j] This function mainly uses lookup table dp[n][n]

    static int findLongestFromACell(int i, int j, int mat[][], int dp[][])

    {

        // Base case

        if (i<0 || i>=n || j<0 || j>=n)

            return 0;

        // If this subproblem is already solved

        if (dp[i][j] != -1)

            return dp[i][j];

        // Since all numbers are unique and in range from 1 to n\*n,

        // there is atmost one possible direction from any cell

        if (j<n-1 && ((mat[i][j] +1) == mat[i][j+1]))

            return dp[i][j] = 1 + findLongestFromACell(i,j+1,mat,dp);

        if (j>0 && (mat[i][j] +1 == mat[i][j-1]))

            return dp[i][j] = 1 + findLongestFromACell(i,j-1,mat,dp);

        if (i>0 && (mat[i][j] +1 == mat[i-1][j]))

            return dp[i][j] = 1 + findLongestFromACell(i-1,j,mat,dp);

        if (i<n-1 && (mat[i][j] +1 == mat[i+1][j]))

            return dp[i][j] = 1 + findLongestFromACell(i+1,j,mat,dp);

        // If none of the adjacent fours is one greater

        return dp[i][j] = 1;

    }

  // Function that returns length of the longest path beginning with any cell

    static int finLongestOverAll(int mat[][])

    {

        // Initialize result

        int result = 1;

        // Create a lookup table and fill all entries in it as -1

        int[][] dp = new int[n][n];

        for(int i=0;i<n;i++)

            for(int j=0;j<n;j++)

                dp[i][j] = -1;

        // Compute longest path beginning from all cells

        for (int i=0; i<n; i++)

        {

            for (int j=0; j<n; j++)

            {

                if (dp[i][j] == -1)

                    findLongestFromACell(i, j, mat, dp);

                //  Update result if needed

                result = Math.max(result, dp[i][j]);

            }

        }

        return result;

    }

    public static void main (String[] args)

    {

        int  mat[][] = { {1, 2, 9},

                         {5, 3, 8},

                         {4, 6, 7} };

System.out.print("Len of longest path is" + finLongestOverAll(mat));

    }

}

**Output:-** Length of the longest path is 4

Time complexity of the above solution is O(n2). It may seem more at first look. If we take a closer look, we can notice that all values of dp[i][j] are computed only once.