1. **C/C++ program for Memoized version for nth Fibonacci number**

**#include<stdio.h>**

**#define NIL -1**

**#define MAX 100**

**int lookup[MAX];**

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

**void \_initialize() {**

**int i;**

**for (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];**

**}**

**int main (){**

**int n = 40;**

**\_initialize();**

**printf("Fibonacci number is %d ", fib(n));**

**return 0;**

**}**

1. **C program for Tabulated version**

**#include<stdio.h>**

**int fib(int n){**

**int f[n+1];**

**int i;**

**f[0] = 0;   f[1] = 1;**

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

**f[i] = f[i-1] + f[i-2];**

**return f[n];**

**}**

**int main (){**

**int n = 9;**

**printf("Fibonacci number is %d ", fib(n));**

**return 0;**

**}**

1. **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;**

**}**

**Output:**

**Length of LCS is 4**

**Time Complexity of the above implementation is O(mn)**

1. **Dynamic Programming implementation of printing LCS**

**#include<iostream>**

**#include<cstring>**

**#include<cstdlib>**

**using namespace std;**

**/\* 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;**

**}**

1. **Dynamic Programming solution to find length of the longest common substring problem**

**#include<iostream>**

**#include<string.h>**

**using namespace std;**

**// A utility function to find maximum of two integers**

**int max(int a, int b)**

**{   return (a > b)? a : b; }**

**/\* Returns length of longest common substring of X[0..m-1] and Y[0..n-1] \*/**

**int LCSubStr(char \*X, char \*Y, int m, int n)**

**{**

***/\*Create a table to store lengths of longest common suffixes of substrings. Note that LCSuff[i][j] contains length of longest common suffix of X[0..i-1] and***

***Y[0..j-1]. The first row and first column entries have no logical meaning, they are used only for simplicity of program \*/***

**int LCSuff[m+1][n+1];**

**int result = 0;  // To store length of the longest common substring**

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

**for (int i=0; i<=m; i++)**

**{**

**for (int j=0; j<=n; j++)**

**{**

**if (i == 0 || j == 0)**

**LCSuff[i][j] = 0;**

**else if (X[i-1] == Y[j-1])**

**{**

**LCSuff[i][j] = LCSuff[i-1][j-1] + 1;**

**result = max(result, LCSuff[i][j]);**

**}**

**else LCSuff[i][j] = 0;**

**}**

**}**

**return result;**

**}**

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

**int main()**

**{**

**char X[] = "OldSite:GeeksforGeeks.org";**

**char Y[] = "NewSite:GeeksQuiz.com";**

**int m = strlen(X);**

**int n = strlen(Y);**

**cout << "Length of Longest Common Substring is "**

**<< LCSubStr(X, Y, m, n);**

**return 0;**

**}**

**Output:**

**Length of Longest Common Substring is 10**

**Time Complexity: O(m\*n)  
Auxiliary Space: O(m\*n)**

1. **A Dynamic Programming based solution for 0-1 Knapsack problem**

**#include<stdio.h>**

**// A utility function that returns maximum of two integers**

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

**// Returns the maximum value that can be put in a knapsack of capacity W**

**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];**

**}**

**int main()**

**{**

**int val[] = {60, 100, 120};**

**int wt[] = {10, 20, 30};**

**int W = 50;**

**int n = sizeof(val)/sizeof(val[0]);**

**printf("%d", knapSack(W, wt, val, n));**

**return 0;**

**}**

**Output:**

**220**

**Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.**

1. **Dynamic Programming code for Matrix Chain Multiplication problem**

**#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)**

1. **Dynamic Programming code for Optimal Binary Search Tree problem**

**#include <stdio.h>**

**#include <limits.h>**

**// A utility function to get sum of array elements**

**// freq[i] to freq[j]**

**int sum(int freq[], int i, int j);**

**/\* A Dynamic Programming based function that calculates**

**minimum cost of a Binary Search Tree. \*/**

**int optimalSearchTree(int keys[], int freq[], int n)**

**{**

**/\* Create an auxiliary 2D matrix to store results**

**of subproblems \*/**

**int cost[n][n];**

**/\* cost[i][j] = Optimal cost of binary search tree**

**that can be formed from keys[i] to keys[j].**

**cost[0][n-1] will store the resultant cost \*/**

**// For a single key, cost is equal to frequency of the key**

**for (int i = 0; i < n; i++)**

**cost[i][i] = freq[i];**

**// Now we need to consider chains of length 2, 3, ... .**

**// L is chain length.**

**for (int L=2; L<=n; L++)**

**{**

**// i is row number in cost[][]**

**for (int i=0; i<=n-L+1; i++)**

**{**

**// Get column number j from row number i and**

**// chain length L**

**int j = i+L-1;**

**cost[i][j] = INT\_MAX;**

**// Try making all keys in interval keys[i..j] as root**

**for (int r=i; r<=j; r++)**

**{**

**// c = cost when keys[r] becomes root of this subtree**

**int c = ((r > i)? cost[i][r-1]:0) +**

**((r < j)? cost[r+1][j]:0) +**

**sum(freq, i, j);**

**if (c < cost[i][j])**

**cost[i][j] = c;**

**}**

**}**

**}**

**return cost[0][n-1];**

**}**

**// A utility function to get sum of array elements**

**// freq[i] to freq[j]**

**int sum(int freq[], int i, int j)**

**{**

**int s = 0;**

**for (int k = i; k <=j; k++)**

**s += freq[k];**

**return s;**

**}**

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

**int main()**

**{**

**int keys[] = {10, 12, 20};**

**int freq[] = {34, 8, 50};**

**int n = sizeof(keys)/sizeof(keys[0]);**

**printf("Cost of Optimal BST is %d ",**

**optimalSearchTree(keys, freq, n));**

**return 0;**

**}**

**Output:**

**Cost of Optimal BST is 142**

**Notes**

**1) The time complexity of the above solution is O(n^4). The time complexity can be easily reduced to O(n^3) by pre-calculating sum of frequencies instead of calling sum() again and again.**

**2) In the above solutions, we have computed optimal cost only. The solutions can be easily modified to store the structure of BSTs also. We can create another auxiliary array of size n to store the structure of tree. All we need to do is, store the chosen ‘r’ in the innermost loop.**