**Practical 5**

**Aim: - Dynamic Programming**

5.1 Find Binomial Coefficient using Dynamic Programming (Implement)

5.2 0/1 Knapsack Problem (Implement)

5.3 Matrix Chain Multiplication (Implement)

5.4 Longest Common Subsequence (Implement)

**Dynamic Programming:**

* Dynamic Programming was invented by Richard Bellman, 1950.
* It is a very general technique for solving optimization problems.
* Using Dynamic Programming requires that the problem can be divided into overlapping similar sub-problems.
* A recursive relation between the larger and smaller sub problems is used to fill out a table. The algorithm remembers solutions of the sub-problems and so does not have to recalculate the solutions.
* Dynamic Programming requires:
* Problem divided into overlapping sub-problems
* Sub-problem can be represented by a table
* Principle of optimality, recursive relation between smaller and larger problems

**5.1 Find Binomial Coefficient using Dynamic Programming (Implement)**

**Theory:-**

A [binomial coefficient](http://en.wikipedia.org/wiki/Binomial_coefficient) C (n, k) can be defined as the coefficient of X^k in the expansion of (1 + X) ^n.

A binomial coefficient C (n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k-element subsets (or k-combinations) of an n-element set.

For Example:   
Write a function that takes two parameters n and k and returns the value of Binomial Coefficient C (n, k).For example, your function should return 6 for n = 4 and k = 2, and it should return 10 for n = 5 and k = 2.

Optimal Substructure would be:  
The value of C (n, k) can be recursively calculated using following standard formula for Binomial Coefficients.

* C (n, k) = C (n-1, k-1) + C (n-1, k)
* C (n, 0) = C (n, n) = 1

**Algorithm: -**

Read n and k

print Bionomial(n,k)

Binomial(int n,int k):

if k==0 OR n==k then

return 1

return Bionomial(n-1,k-1)+Bionomial(n-1,k)

OR

 Binomial(n, k)

for i ← 0 to n do

for i = 0 to min(i, k) do

if j==0 or j==i then C[i, j] ← 1

else C[i, j] ← C[i-1, j-1] + C[i-1, j]

return C[n, k]

**Program: -**

**Code: -**

#include<iostream>

#include<iomanip>

using namespace std;

int main()

{

int n,k;

cout<<"Enter the value of n and k : ";

cin>>n>>k;

int a[n+1][k+1];

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

{

for(int j=0;j<=k;j++)

{

if(i<j)

{

a[i][j]=0;

}

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

{

a[i][j]=1;

//cout<<a[i][j]<<" ";

}

else

{

a[i][j]=a[i-1][j-1]+a[i-1][j];

}

}

}

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

for(int j=0;j<=k && j<=i;j++) {

cout<<setw(2)<<a[i][j]<<" ";

}

cout<<endl;

}

if(n>k) {

cout<<endl<<endl<<"|-------------------|"<<endl;

cout<<"| Answer is : "<<a[n][k]<<" |"<<endl;

cout<<"|-------------------|"<<endl;

}

else {

cout<<endl<<endl<<"|------------------------|"<<endl;

cout<<"| Answer is : "<<"invalid"<<" |"<<endl;

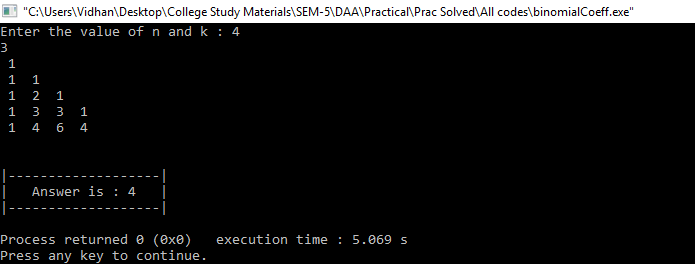
cout<<"|------------------------|"<<endl;

}

return 0;

}

**Output: -**



**Conclusion: -**

We can evaluate any Binomial Co-efficient by using the above code. The output of the code generates a Parse tree. The code has the time complexity of O (n, k).

**5.2 0/1 Knapsack Problem (Implement)**

**Theory: -**

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).

**Algorithm: -**

Sort the n objects from large to small based on the ratios vi/wi. We assume the arrays w[1...n] and v[i...n] store the respective weights and values after sorting.

Initialize array x[1...n] to zeroes

Weight=0 ; i=1

While(i<=n and weight<w) do

* if weight +w[i]<=w then x[i]=1
* else x[i]=(w-weight/w[i]
* weight=weight + x[i]\*w[i] and i++

**Program: -**

**Code: -**

#include<iostream>

using namespace std;

struct item

{

int wt;

int val;

int id;

};

int main()

{

int n,w;

cout<<"Enter no of Items : ";

cin>>n;

item t[n];

cout<<"Enter weight and value : ";

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

{

cin>>t[i].wt;

cin>>t[i].val;

t[i].id=0;

}

cout<<"Enter total weight : ";

cin>>w;

int tab[n+1][w+1];

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

{

for(int j=0;j<=w;j++)

{

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

{

tab[i][j]=0;

}

else

{

if(j>=t[i-1].wt)

{

tab[i][j]=max(t[i-1].val+tab[i-1][j-t[i-1].wt],tab[i-1][j]);

}

else

{

tab[i][j]=tab[i-1][j];

}

}

}

}

int x=n;

int y=w;

while(x>0 && y>0)

{

if(tab[x][y]!=tab[x-1][y])

{

t[x-1].id=1;

x--;

y-=t[x-1].wt;

}

else

{

x--;

}

}

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

{

for(int j=0;j<=w;j++)

{

cout<<tab[i][j]<<" ";

}

cout<<endl;

}

cout<<endl;

cout<<"[ ";

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

{

if(i==n-1)

cout<<t[i].id;

else

cout<<t[i].id<<", ";

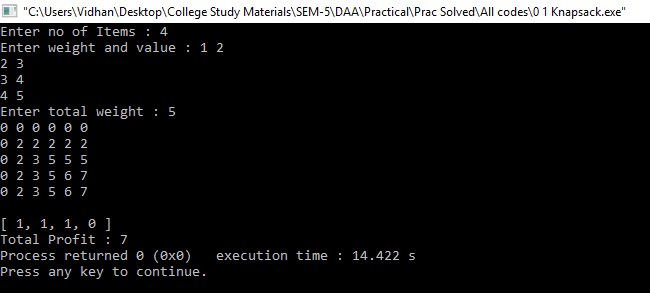
}

cout<<" ]";

cout<<endl<<"Total Profit : "<<tab[n][w];

}

**Output: -**



**Conclusion: -**

0/1 Knapsack problem is solved by using the concept of Dynamic Programming. It has the time complexity of O (W n).

**5.3 Matrix Chain Multiplication (Implement)**

**Theory: -**

The chain matrix multiplication problem is perhaps the most popular example of dynamic programming used in the upper undergraduate course (or review basic issues of dynamic programming in advanced algorithm's class).

The chain matrix multiplication problem involves the question of determining the optimal sequence for performing a series of operations. This general class of problem is important in complier design for code optimization and in databases for query optimization. We will study the problem in a very restricted instance, where the dynamic programming issues are clear. Suppose that our problem is to multiply a chain of n matrices A1 A2 ... An.

**Algorithm: -**

Matrix-Chain (array p [1 ... n], int n) {

Array s [1 ... n − 1, 2 ... n];

FOR i = 1 TO n DO m [i, i] = 0;

FOR L = 2 TO n DO {

FOR i = 1 TO n − L + 1 do

    j = i + L − 1;

    m [i, j] = infinity;

 FOR k = i TO j − 1 DO

   q = m [i, k] + m [k + 1, j] + p [i − 1] p[k] p[j];

                               IF (q < m [i, j])

                                         m [i, j] = q;

                                          s [i, j] = k;

          return m [1, n] (final cost) and s (splitting markers);

**Program: -**

**Code: -**

#include<iostream>

#define max\_int 10000

using namespace std;

int s[10][10];

void matrix(int p[], int n){

int l,i, j, temp;

int m[n][n];

cout<<"n"<<n<<endl;

for(i=0;i<n;i++){

for(j=0;j<n;j++){

m[i][j] = 0;

s[i][j] = 0;

}

}

for(l=2;l<n;l++){

for(i=1;i<n-l+1;i++){

j=i+l-1;

m[i][j] = max\_int;

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

temp = m[i][k] + m[k+1][j] + p[i-1] \* p[k] \* p[j];

if(temp<m[i][j]){

m[i][j] = temp;

s[i][j] = k;

}

}

}

}

cout<<"\nMatrix:\n";

for(int i=1;i<n;i++){

for(int k=1;k<n;k++){

cout<<m[i][k];

}

cout<<endl;

}

cout<<"\nK values:\n"<<endl;

for(i=1;i<n;i++){

for(int k=1;k<n;k++){

cout<<s[i][k];

}

cout<<endl;

}

}

void solution(int i,int j)

{

if (i == j)

cout<<"A"<<i;

else

{

cout<<"(";

solution(i,s[i][j]);

solution(s[i][j] + 1, j);

cout<<")";

}

}

int main(){

int n,p[5],m,i,k,j;

cout<<"Enter no. of matrix:"<<endl;

cin>>m;

cout<<"Enter no. of p:";

cin>>k;

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

cin>>p[i];

matrix(p,k);

cout<<"\nSequence:";

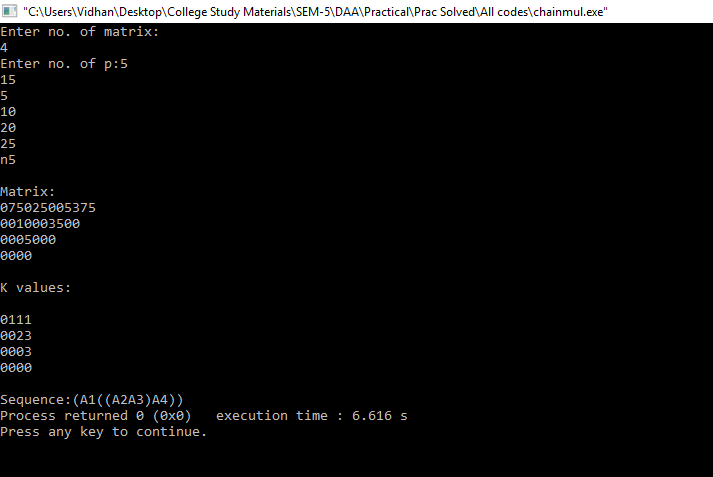
i=1,j=m;

solution(i,j);

return 0;

}

**Output: -**



**Conclusion: -**

Matrix chain multiplication uses concept of dynamic programming. It helps in performing matrix multiplication by simplifying the order of multiplication of matrices. We can optimize the matrix multiplication method and henceforth we save time and memory by using chain multiplication.

**5.4 Longest Common Subsequence (Implement)**

**Theory: -**

A subsequence of a string S, is a set of characters that appear in left-to-right order, but not necessarily consecutively.

Example

ACT T GCG

ACT, AT T C, T, and ACT T GC are all subsequences.

T T A is not a subsequence

A common subsequence of two strings is a subsequence that appears in both strings. A longest common subsequence is a common subsequence of maximal length.

Example

S1 = AAACCGT GAGT T AT T CGT T CT AGAA

S2 = CACCCCT AAGGT ACCT T T GGT T C

**LCS is**

**ACCT AGT ACTTTG**

**Algorithm: -**

LCS-LENGTH (X, Y)

m<- length[X]

n<- length[Y]

for i<- 1 to m

do c[i,0] <-0

for j<- 0 to n

do c[0,j] <-0

for i<- 1 to m

do for j<- 1 to n

do if xi=yi

 then c[i,j]<- c[i-1,j-1]+1

b[i,j] <-"↖ "

else if c[i-1,j]>=c[i,j-1]

then c[i,j]<-c[i-1,j]

b[i,j]<-" ↑"

 else c[i,j]<-c[i,j-1]

b[i,j]<-" ←"

return c and b

**Program: -**

**Code: -**

#include<iostream>

#include<stdio.h>

#include<string.h>

using namespace std;

int b[20][20]={0},c[20][20]={0};

void LCS\_length(char x[],char y[])

{

int m=strlen(x),n=strlen(y);

for(int i=1;i<=m;i++){

for(int j=1;j<=n;j++){

if(x[i-1]==y[j-1]){

c[i][j]=c[i-1][j-1]+1;

b[i][j]=3;

}

else{

if(c[i-1][j]>=c[i][j-1]) {

c[i][j]=c[i-1][j];

b[i][j]=1;

}

else{

c[i][j]=c[i][j-1];

b[i][j]=2;

}

}

}

}

}

void print\_LCS(char x[],int i,int j)

{

if(i<=0&&j<=0)

return;

if(b[i][j]==3) {

print\_LCS(x,i-1,j-1);

cout<<x[i-1];

}

else if(b[i][j]==1) {

print\_LCS(x,i-1,j);

}

else {

print\_LCS(x,i,j-1);

}

}

int main()

{

char x[20],y[20];

cout<<"Enter the string:"<<endl;

cin>>x;

cout<<"Enter the sub-string:"<<endl;

cin>>y;

LCS\_length(x,y);

for(int i=0;i<=strlen(x);i++) {

for(int j=0;j<=strlen(y);j++){

cout<<c[i][j]<<" ";

}

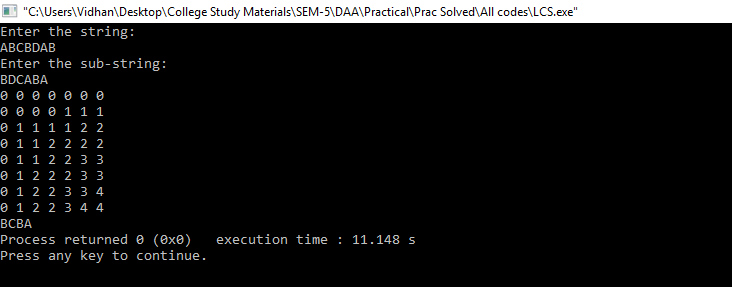
cout<<endl;

}

print\_LCS(x,strlen(x),strlen(y));

}

**Output: -**



**Conclusion: -**

Longest common sub-sequence uses concept of Dynamic Programming. From the given expression or string, we can compute sub-string or say sub-sequence by using concept of dynamic programming.