



EXPERIMENT - 4

Algorithms Design and Analysis Lab

Aim

To implement dynamic programming techniques and analyse its time complexity.

Syeda Reeha Quasar

14114802719

4C7

EXPERIMENT – 4

Aim:

To implement dynamic programming techniques and analyse its time complexity.

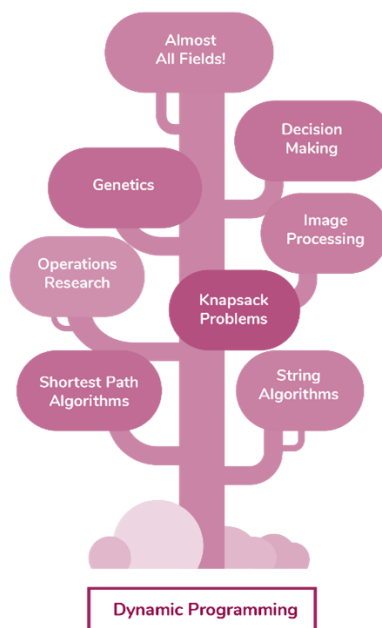
Theory:

Dynamic programming solves problems by combining the solution of sub problems. It is only applicable when sub problems are not independent, that is, they share sub sub-Problems. Each time a new sub problem is solved, its solution is stored such that other sub problems sharing the stored sub problem can use the stored value instead of doing a recalculation, thereby saving work compared to applying the divide-and-conquer principle on the same problem which would have recalculated everything.

A dynamic programming solution has three components:

- Formulate the answer as a recurrence relation or recursive algorithm.
- Show that the number of different instances of your recurrence is bounded by a polynomial.
- Specify an order of evaluation for the recurrence so you always have what you need.

Applications of Dynamic Programming



4.1 LCS (Longest Common Subsequence)

The longest common subsequence problem is finding the longest sequence which exists in both the given strings.

Subsequence:

Let us consider a sequence $S = \langle s_1, s_2, s_3, s_4, \dots, s_n \rangle$. A sequence $Z = \langle z_1, z_2, z_3, z_4, \dots, z_m \rangle$ over S is called a subsequence of S , if and only if it can be derived from S deletion of some elements.

Common Subsequence:

Suppose, X and Y are two sequences over a finite set of elements. We can say that Z is a common subsequence of X and Y , if Z is a subsequence of both X and Y .

Longest Common Subsequence:

If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length. It is a classic computer science problem, the basis of data comparison programs such as the diff-utility, and has applications in bioinformatics.

Naïve Method:

Let X be a sequence of length m and Y a sequence of length n . Check for every subsequence of X whether it is a subsequence of Y , and return the longest common subsequence found. There are 2^m subsequences of X . Testing sequences whether or not it is a subsequence of Y takes $O(n)$ time. Thus, the naïve algorithm would take $O(n2^m)$ time.

Dynamic Programming:

Let $X = \langle x_1, x_2, x_3, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, y_3, \dots, y_n \rangle$ be the sequences. To compute the length of an element the following algorithm is used. In this

procedure, table **C[m, n]** is computed in row major order and another table **B[m,n]** is computed to construct optimal solution.

Recursive Equation:

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases}$$

Pseudo code:

Let X and Y are Sequences whose LCS has to be found.

LCS-Length-Table-Formulation (X, Y)

$m := \text{length}(X)$

$n := \text{length}(Y)$

for $i = 1$ to m do

$C[i, 0] := 0$

for $j = 1$ to n do

$C[0, j] := 0$

for $i = 1$ to m do

for $j = 1$ to n do

if $x_i = y_j$

$C[i, j] := C[i - 1, j - 1] + 1$

$B[i, j] := 'D'$

else

if $C[i - 1, j] \geq C[i, j - 1]$

$C[i, j] := C[i - 1, j] + 1$

$B[i, j] := 'U'$

else

$C[i, j] := C[i, j - 1] + 1$

$B[i, j] := 'L'$

return C and B

Algorithm: Print-LCS (B, X, i, j)

if $i = 0$ and $j = 0$

return

if $B[i, j] = 'D'$

Print-LCS(B, X, $i-1$, $j-1$)

Print(x_i)

else if $B[i, j] = 'U'$

Print-LCS(B, X, $i-1$, j)

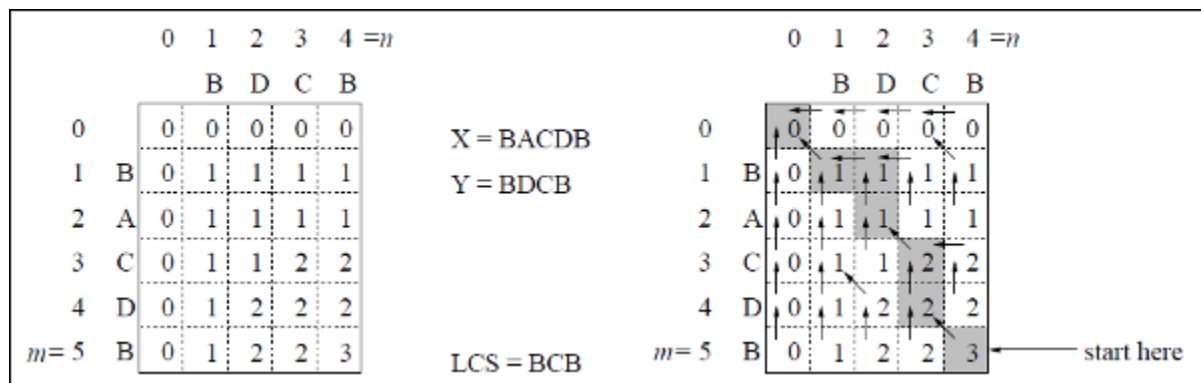
else

Print-LCS(B, X, i , $j-1$)

Sample Example:

We have two strings $X = \text{BACDB}$ and $Y = \text{BDCB}$ to find the longest common subsequence.

Following the algorithm LCS-Length-Table-Formulation (as stated above), we have calculated table C (shown on the left hand side) and table B (shown on the right hand side). In table B, instead of 'D', 'L' and 'U', we are using the diagonal arrow, left arrow and up arrow, respectively. After generating table B, the LCS is determined by function LCS-Print. The result is BCB.



Result and Analysis

To populate the table, the outer **for** loop iterates **m** times and the inner **for** loop iterates **n** times. Hence, the complexity of the algorithm is $O(m, n)$, where **m** and **n** are the length of two strings.

LCS:

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;

void lcs(string s1, string s2, int m, int n)
{
    int dpTable[m + 1][n + 1];

    for (int i = 0; i <= m; i++) // creating dp table bottom up
    {
        for (int j = 0; j <= n; j++)
        {
            if (i == 0 || j == 0)
                dpTable[i][j] = 0;
            else if (s1[i - 1] == s2[j - 1])
                dpTable[i][j] = dpTable[i - 1][j - 1] + 1;
            else
                dpTable[i][j] = max(dpTable[i - 1][j], dpTable[i][j - 1]);
        }
    }

    int index = dpTable[m][n];
    char lcsAlgo[index + 1];
    lcsAlgo[index] = '\0';

    int i = m, j = n;
    while (i > 0 && j > 0)
    {
        if (s1[i - 1] == s2[j - 1])
```

```
        {
            lcsAlgo[index - 1] = s1[i - 1];
            i--;
            j--;
            index--;
        }

        else if (dpTable[i - 1][j] > dpTable[i][j - 1])
            i--;
        else
            j--;
    }
    cout << "s1 : " << s1 << "\nS2 : " << s2 << "\nlcs: " << lcsAlgo << "\n";
}

int main()
{
    // string s1 = "ACADB";
    // string s2 = "CBDA";
    string s1, s2;
    cout << "Please Enter the Strings for finding LCS" << endl;
    cin >> s1 >> s2;
    int m = s1.size();
    int n = s2.size();


    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    lcs(s1, s2, m, n);

    auto end = chrono::high_resolution_clock::now();


    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;
    cout << "Time difference is: " << time_taken << setprecision(6) << endl;
    return 0;
}
```

Output:



```
Please Enter the Strings for finding LCS
abcdefgh
ajkdbacacdefgh
s1 : abcdefgh
s2 : ajkdbacacdefgh
lcs: abcdefgh
Time difference is: 6.8233e-05

...Program finished with exit code 0
Press ENTER to exit console.
```



```
Please Enter the Strings for finding LCS
abc
defjkl
s1 : abc
s2 : defjkl
lcs:
Time difference is: 8.2993e-05

...Program finished with exit code 0
Press ENTER to exit console.
```


LCS Batch Analysis:

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;

double lcs(string s1, string s2, int m, int n)
{
    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    int dpTable[m + 1][n + 1];

    for (int i = 0; i <= m; i++) // creating dp table bottom up
    {
        for (int j = 0; j <= n; j++)
        {
            if (i == 0 || j == 0)
                dpTable[i][j] = 0;
            else if (s1[i - 1] == s2[j - 1])
                dpTable[i][j] = dpTable[i - 1][j - 1] + 1;
            else
                dpTable[i][j] = max(dpTable[i - 1][j], dpTable[i][j - 1]);
        }
    }

    int index = dpTable[m][n];
    char lcsAlgo[index + 1];
    lcsAlgo[index] = '\0';

    int i = m, j = n;
    while (i > 0 && j > 0)
    {
        if (s1[i - 1] == s2[j - 1])
        {
            lcsAlgo[index - 1] = s1[i - 1];
            i--;
            j--;
            index--;
        }
    }
}
```

```
    }

    else if (dpTable[i - 1][j] > dpTable[i][j - 1])
        i--;
    else
        j--;
}
auto end = chrono::high_resolution_clock::now();

double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
time_taken *= 1e-9;
return time_taken;
}

string randomStr(const int len) {
    static const char letters[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    string s;
    s.reserve(len);
    for (int i = 0; i < len; ++i) {
        s += letters[rand() % (sizeof(letters) - 1)];
    }
    return s;
}

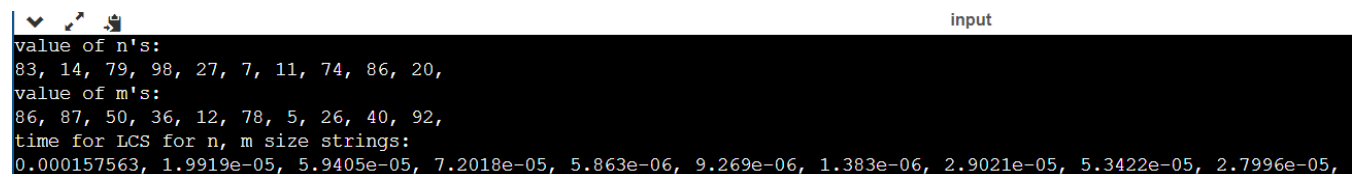
void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

int main()
{
    double times[10];
    int ns[10];
    int ms[10];
    for (int x = 0; x < 10; x++)
```

```
{
    int n = rand() % 100;
    int m = rand() % 100;
    ns[x] = n;
    ms[x] = m;
    string s1 = randomStr(n);
    string s2 = randomStr(m);
    times[x] = lcs(s1, s2, n, m);
}
cout << "value of n's: " << endl;
printArray(ns, 10);
cout << "value of m's: " << endl;
printArray(ms, 10);
cout << "time for LCS for n, m size strings: " << endl;
printArray(times, 10);
return 0;
}
```

Output:



```
input
value of n's:
83, 14, 79, 98, 27, 7, 11, 74, 86, 20,
value of m's:
86, 87, 50, 36, 12, 78, 5, 26, 40, 92,
time for LCS for n, m size strings:
0.000157563, 1.9919e-05, 5.9405e-05, 7.2018e-05, 5.863e-06, 9.269e-06, 1.383e-06, 2.9021e-05, 5.3422e-05, 2.7996e-05,
```

```
value of n's:
83, 14, 79, 98, 27, 7, 11, 74, 86, 20,
value of m's:
86, 87, 50, 36, 12, 78, 5, 26, 40, 92,
```

```
time for LCS for n, m size strings:
0.000157563, 1.9919e-05, 5.9405e-05, 7.2018e-05, 5.863e-06, 9.269e-06, 1.383e-06, 2.9021e-05, 5.3422e-05, 2.7996e-05,
```

value of n's:

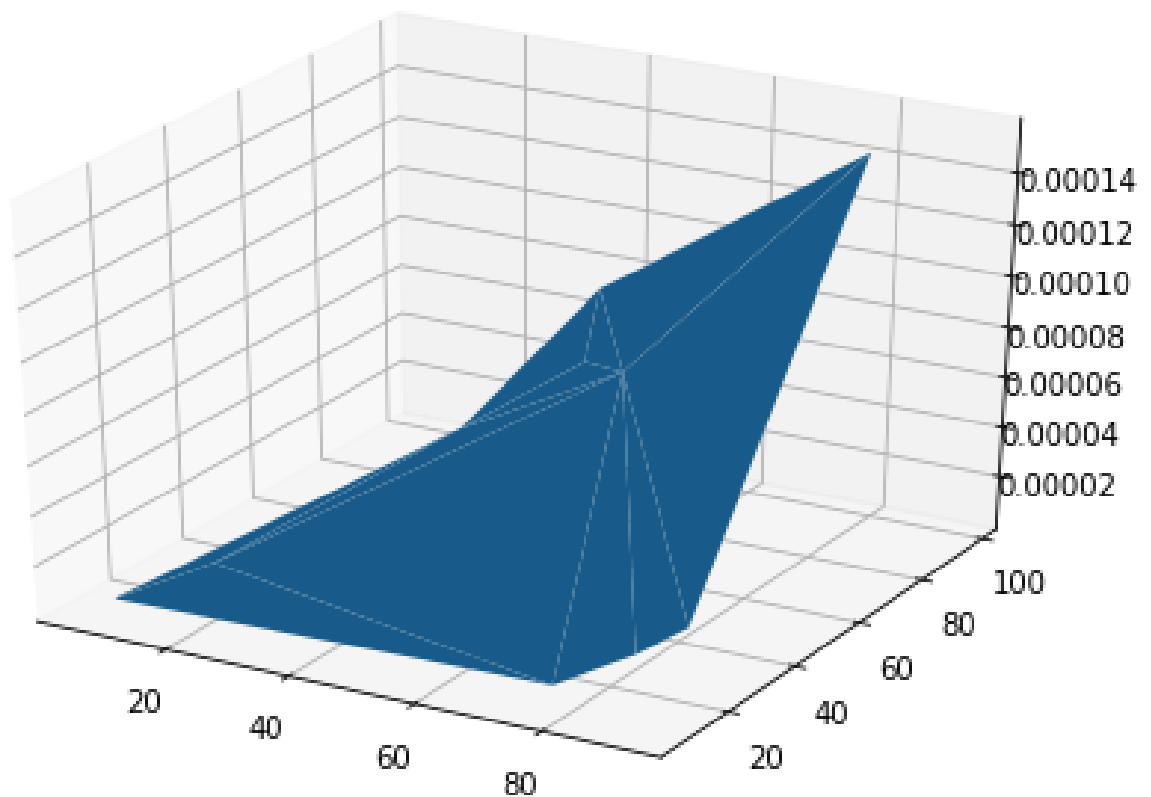
83, 14, 79, 98, 27, 7, 11, 74, 86, 20

value of m's:

86, 87, 50, 36, 12, 78, 5, 26, 40, 92

time for LCS for n, m size strings:

0.000157563, 1.9919e-05, 5.9405e-05, 7.2018e-05, 5.863e-06, 9.269e-06, 1.383e-06,
2.9021e-05, 5.3422e-05, 2.7996e-05



Viva Questions

1. Give any application of LCS problem.

Ans.

It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

The LCS has been used to study various areas (see [2, 3]), such as text analysis, pattern recognition, file comparison, efficient tree matching [4], etc. Biological applications of the LCS and similarity measurement are varied, from sequence alignment [5] in comparative genomics [6], to phylogenetic construction and analysis, to rapid search in huge biological sequences [7], to compression and efficient storage of the rapidly expanding genomic data sets [8, 9], to re-sequencing a set of strings given a target string [10], an important step in efficient genome assembly.

2. Can LCS problem be solved by Greedy strategy?

Ans.

Yes, LCS can be solved using Greedy Strategy.

3. Find the LCS for input Sequences "ABCDGH" and "AEDFHR"

Ans.

LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.

4. Find X= BDCABA and Y= ABCBDAB. Find the length of LCS

Ans.

The length of LCS is 4

BCAB,BCBA,BDAB

ALGO:

LCS(m,n)

```
{ 1 + L(m-1,n-1) if A[m] = B[n]    // if first character match then take it
  max( LCS(m-1,n), LCS(m,n-1) ) else
}
```

BDCABA & ABCBDAB : First character mismatch

then **two cases** :

1) BDCABA & BCBADAB : we hv removed 1st char from second string..

Here **1st character match**,

$LCS = 1 + LCS(DCABA \& CBADAB)$

Now **D & C mismatch** again take two(can apply algo)..

Here to do quickly we observe manually that longest possible now is 3 characters(we can use algo also..) - So, i get CAB ,CBA,DAB

which when combined with B gives BCAB,BCBA,BDAB

2) DCABA & ABCBDAB : Here we hv removed first B from 1st sequence

Now, D & A first characters again mismatch.

two parts again:

2a) CABA & ABCBDAB : no common subsequence of length 4 possible here

2b) DCABA & BCBADAB : similarly here too - no possibility for a subsequence of length 4

5. Find the LCS for input Sequences “abcdghijklm” and “bcdehjklsmnd”

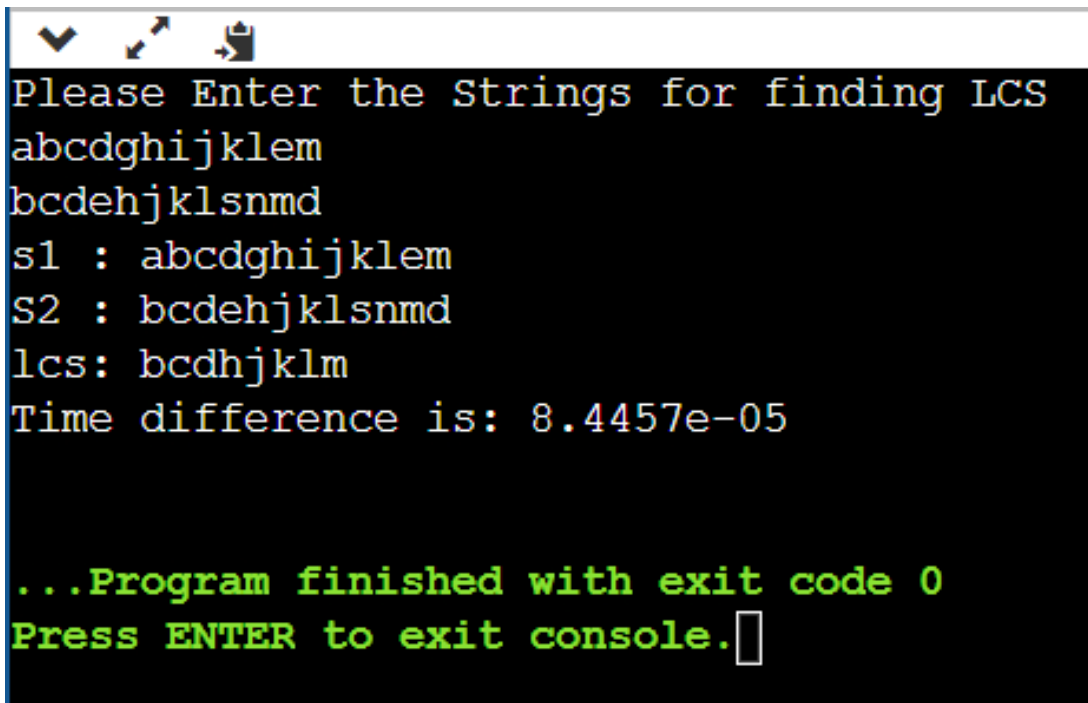
Ans.

s1 : abcdghijklm

S2 : bcdehjklsnmd

lcs: bcdhjklm

Time difference is: 8.4457e-05



```
Please Enter the Strings for finding LCS
abcdghijklm
bcdehjklsnmd
s1 : abcdghijklm
S2 : bcdehjklsnmd
lcs: bcdhjklm
Time difference is: 8.4457e-05

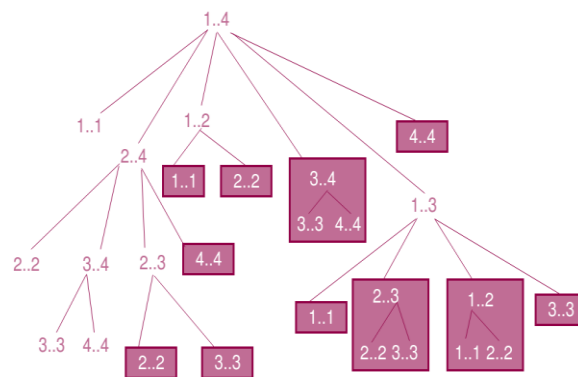
...Program finished with exit code 0
Press ENTER to exit console.
```

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.

$$(ABC)D = (AB)(CD) = A(BCD) = \dots$$
$$(AB)C = (10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500 \text{ operations}$$

$$A(BC) = (30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000 \text{ operations.}$$

Given an array $p[]$ which represents the chain of matrices such that the i th matrix A_i is of dimension $p[i-1] \times p[i]$. We need to write a function `MatrixChainOrder()` that should return the minimum number of multiplications needed to multiply the chain.



$$(AB)C = (10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500 \text{ operations}$$

$$A(BC) = (30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000 \text{ operations.}$$

Given an array $p[]$ which represents the chain of matrices such that the i th matrix A_i is of dimension $p[i-1] \times p[i]$. We need to write a function `MatrixChainOrder()` that should return the minimum number of multiplications needed to multiply the chain.

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

int dp[100][100];

int matrixChainMemo(int *p, int i, int j){
    if (i == j)
        return 0;

    if (dp[i][j] != -1)
        return dp[i][j];

    dp[i][j] = INT_MAX;

    for (int k = i; k < j; k++) {
        dp[i][j] = min(dp[i][j], matrixChainMemo(p, i, k) +
matrixChainMemo(p, k + 1, j) + p[i - 1] * p[k] * p[j]);
    }
    return dp[i][j];
}

int MatrixChainOrder(int *p, int n){
    return matrixChainMemo(p, 1, n - 1);
}

int main(){
    int arr[] = {10, 20, 30, 40};
```

```
int n = sizeof(arr) / sizeof(arr[0]);
memset(dp, -1, sizeof dp);

auto start = chrono::high_resolution_clock::now();

ios_base::sync_with_stdio(false);

cout << "Minimum number of multiplications is " << MatrixChainOrder(arr,
n) << endl;

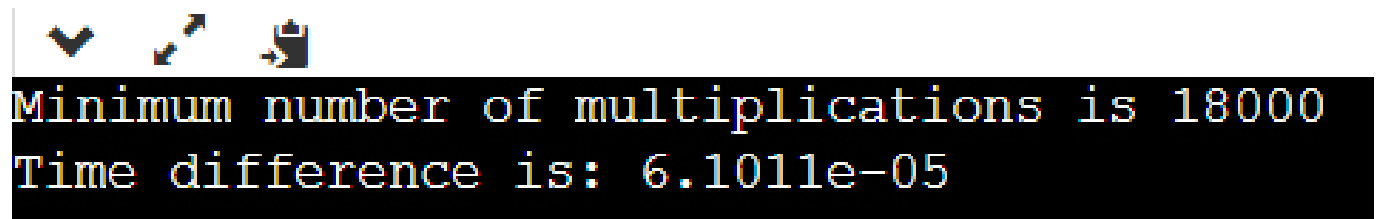
auto end = chrono::high_resolution_clock::now();

double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
time_taken *= 1e-9;

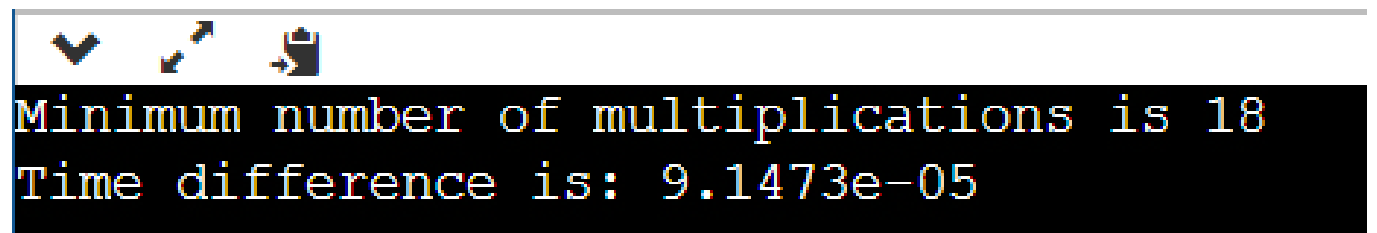
cout << "Time difference is: " << time_taken << setprecision(6) << endl;

return 0;
}
```

Output:

A terminal window with a black background and yellow text. It shows the output of the program: "Minimum number of multiplications is 18000" and "Time difference is: 6.1011e-05".

```
Minimum number of multiplications is 18000
Time difference is: 6.1011e-05
```

A terminal window with a black background and yellow text. It shows the output of the program: "Minimum number of multiplications is 18" and "Time difference is: 9.1473e-05".

```
Minimum number of multiplications is 18
Time difference is: 9.1473e-05
```

Batch Analysis

Source Code

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

int dp[100][100];

int matrixChainMemo(int *p, int i, int j){
    if (i == j)
        return 0;

    if (dp[i][j] != -1)
        return dp[i][j];

    dp[i][j] = INT_MAX;

    for (int k = i; k < j; k++) {
        dp[i][j] = min(
            dp[i][j], matrixChainMemo(p, i, k) + matrixChainMemo(p, k + 1, j) +
            p[i - 1] * p[k] * p[j]);
    }
    return dp[i][j];
}

int MatrixChainOrder(int *p, int n){
    int i = 1, j = n - 1;
    return matrixChainMemo(p, i, j);
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}
```

```
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

int main(){
    double times[10];
    int ns[10];
    for (int x = 0; x < 10; x++)    {
        memset(dp, -1, sizeof dp);
        int n = rand() % 100;
        ns[x] = n;

        int arr[n];
        for (int i = 0; i < n; i++) {
            arr[i] = rand() % 100;
        }

        auto start = chrono::high_resolution_clock::now();
        ios_base::sync_with_stdio(false);

        MatrixChainOrder(arr, n);

        auto end = chrono::high_resolution_clock::now();

        double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

        times[x] = time_taken;
    }

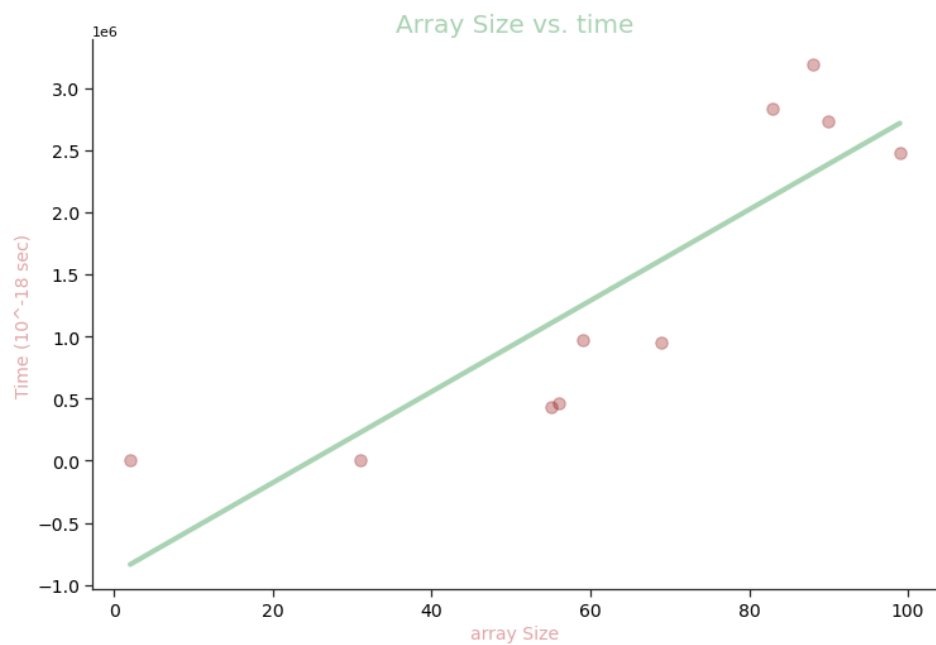
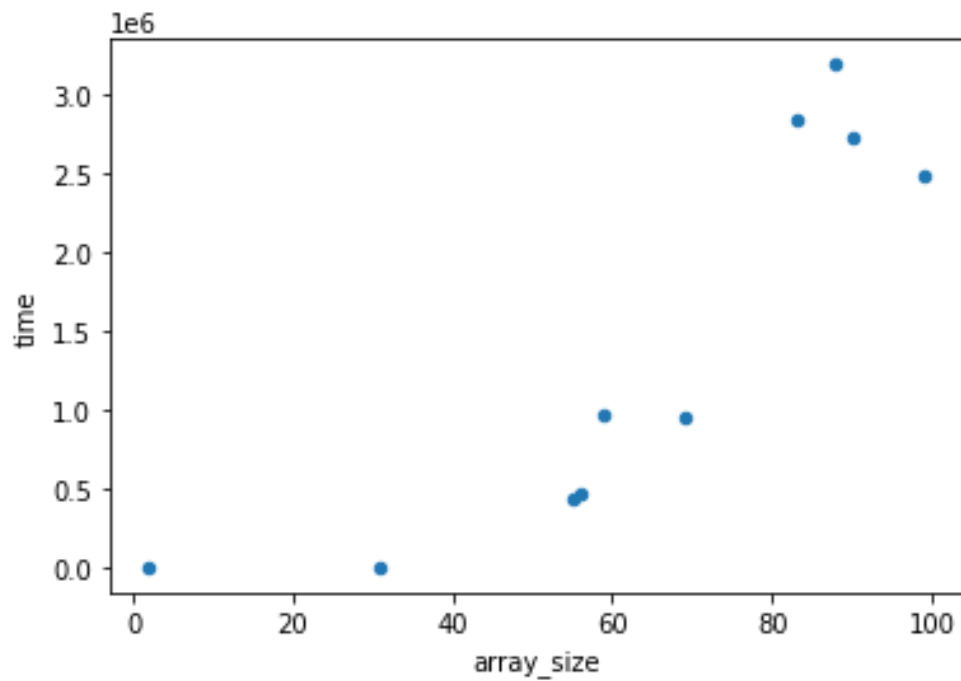
    cout << "value of n's: " << endl;
    printArray(ns, 10);

    cout << "time for each n: " << endl;
    printArray(times, 10);

    return 0;
}
```

Output:

```
input
value of n's:
83, 88, 56, 69, 90, 59, 2, 99, 55, 31,
time for each n:
2.83879e+06, 3.19062e+06, 460596, 953485, 2.72973e+06, 974584, 235, 2.48403e+06, 432541, 76248,
```



Viva Questions

1. What is the recurrence relation for MCM problem?

Ans.

$$B(i, j) = B(i, k) + B(k + 1, j) + p[i - 1] * p[k] * p[j]$$

2. Let A1, A2, A3, A4, A5 be five matrices of dimensions 2x3, 3x5, 5x2, 2x4, 4x3 respectively. Find the minimum number of scalar multiplications required to find the product A1 A2 A3 A4 A5 using the basic matrix multiplication method?

Ans.

78

3. Consider the two matrices P and Q which are 10 x 20 and 20 x 30 matrices respectively. What is the number of multiplications required to multiply the two matrices?

Ans.

The number of multiplications required is $10 * 20 * 30$.

4. Consider the matrices P, Q and R which are 10 x 20, 20 x 30 and 30 x 40 matrices respectively. What is the minimum number of multiplications required to multiply the three matrices?

Ans.

The minimum number of multiplications are 18000. This is the case when the matrices are parenthesized as $(P * Q) * R$.

5. Can this problem be solved by greedy approach. Justify?

Ans.

Unfortunately, **there is no good "greedy choice"** for Matrix Chain Multiplication, meaning that for any choice that's easy to compute, there is always some input sequence of matrices for which your greedy algorithm will not find the optimum parenthesization.

At each step only least value is selected among all elements of in the array $p[x, y]$, so that the multiplication cost is kept minimum at each step. This greedy approach ensures that the solution is optimal with least cost involved and the output is a fully parenthesized product of matrices.

Optimal Binary Search

an **optimal binary search tree (OBST)**, sometimes called a **weight-balanced binary tree**, is a binary search tree which provides the smallest possible search time for a given sequence of accesses (or access probabilities).

Optimal BSTs are generally divided into two types:

- Static
- Dynamic

In the **static optimality** problem, the tree cannot be modified after it has been constructed. In this case, there exists some particular layout of the nodes of the tree which provides the smallest expected search time for the given access probabilities. Various algorithms exist to construct or approximate the statically optimal tree given the information on the access probabilities of the elements.

- ❖ OBST is one special kind of advanced tree.
- ❖ It focus on how to reduce the cost of the search of the BST.
- ❖ It may not have the lowest height !
- ❖ It needs 3 tables to record probabilities, cost, and root.
- ❖ It has n keys (representation k_1, k_2, \dots, k_n) in sorted order (so that $k_1 < k_2 < \dots < k_n$), and we wish to build a binary search tree from these keys. For each k_i , we have a probability p_i that a search will be for k_i .
- ❖ In contrast of, some searches may be for values not in k_i , and so we also have $n+1$ "dummy keys" d_0, d_1, \dots, d_n representing not in k_i .
- ❖ In particular, d_0 represents all values less than k_1 , and d_n represents all values greater than k_n , and for $i=1, 2, \dots, n-1$, the dummy key d_i represents all values between k_i and k_{i+1} .

*** The dummy keys are leaves (external nodes), and the data keys mean internal nodes.**

Step1: The structure of an OBST

To characterize the optimal substructure of OBST, we start with an observation about sub trees. Consider any sub tree of a BST. It must contain keys in a contiguous range k_i, \dots, k_j , for some $1 \leq i \leq j \leq n$. In addition, a sub tree that contains keys k_i, \dots, k_j must also have as its leaves the dummy keys d_{i-1}, \dots, d_j

- ❖ We need to use the optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to sub problems. Given keys k_i, \dots, k_j , one of these keys, say k_r ($i \leq r \leq j$), will be the root of an optimal sub tree containing these keys. The left sub tree of the root k_r will contain the keys (k_i, \dots, k_{r-1}) and the dummy keys $(d_{i-1}, \dots, d_{r-1})$, and the right sub tree will contain the keys (k_{r+1}, \dots, k_j) and the dummy keys (d_r, \dots, d_j) . As long as we examine all candidate roots k_r , where $i \leq r \leq j$, and we determine all optimal binary search trees containing k_i, \dots, k_{r-1} and those containing k_{r+1}, \dots, k_j , we are guaranteed that we will find an OBST.
- ❖ There is one detail worth noting about "empty" sub trees. Suppose that in a sub tree with keys k_i, \dots, k_j , we select k_i as the root. By the above argument, k_i 's left sub tree contains the keys k_i, \dots, k_{i-1} . It is natural to interpret this sequence as containing no keys. It is easy to know that sub trees also contain dummy keys. The sequence has no actual keys but does contain the single dummy key d_{i-1} . Symmetrically, if we select k_j as the root, then k_j 's right sub tree contains the keys k_{j+1}, \dots, k_j ; this right sub tree contains no actual keys, but it does contain the dummy key d_j .

Step2: A recursive solution

- ❖ We need to define the value of an optimal solution recursively. We pick our sub problem domain as finding an OBST containing the keys k_i, \dots, k_j , where $i \geq 1$, $j \leq n$, and $j \geq i-1$. (It is when $j=i-1$ that there are no actual keys; we have just the dummy key d_{i-1} .)
- ❖ Let us define $e[i, j]$ as the expected cost of searching an OBST containing the keys k_i, \dots, k_j . Ultimately, we wish to compute $e[1, n]$.
- ❖ The easy case occurs when $j=i-1$. Then we have just the dummy key d_{i-1} . The expected search cost is $e[i, i-1] = q_{i-1}$.
- ❖ When $j \geq i$, we need to select a root k_r from among k_i, \dots, k_j and then make an OBST with keys k_i, \dots, k_{r-1} its left sub tree and an OBST with keys k_{r+1}, \dots, k_j its right sub tree. By the time, what happens to the expected search cost of a sub tree when it becomes a sub tree of a node? The answer is that the depth of each node in the sub tree increases by 1.

- ❖ By the second statement, the expected search cost of this sub tree increases by the sum of all the probabilities in the sub tree. For a sub tree with keys k_i, \dots, k_j let us denote this sum of probabilities as

$$w(i, j) = (i \sim j) \sum p_i + (i+1 \sim j) \sum q_i$$

Thus, if k_r is the root of an optimal sub tree containing keys k_i, \dots, k_j , we have

$$E[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$$

Nothing that $w(i, j) = w(i, r-1) + p_r + w(r+1, j)$

- ❖ We rewrite $e[i, j]$ as

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j)$$

The recursive equation as above assumes that we know which node k_r to use as the root. We choose the root that gives the lowest expected search cost, giving us our final recursive formulation:

$$E[i, j] =$$

case1: if $i \leq j, i \leq r \leq j$

$$E[i, j] = \min\{e[i, r-1] + e[r+1, j] + w(i, j)\}$$

case2: if $j = i-1$; $E[i, j] = q_{i-1}$

- ❖ The $e[i, j]$ values give the expected search costs in OBST. To help us keep track of the structure of OBST, we define $\text{root}[i, j]$, for $1 \leq i \leq j \leq n$, to be the index r for which k_r is the root of an OBST containing keys k_i, \dots, k_j .

Step3: Computing the expected search cost of an OBST

- ❖ We store the $e[i, j]$ values in a table $e[1..n+1, 0..n]$. The first index needs to run to $n+1$ rather than n because in order to have a sub tree containing only the dummy key d_n , we will need to compute and store $e[n+1, n]$. The second index needs to start from 0 because in order to have a sub tree containing only the dummy key d_0 , we will need to compute and store $e[1, 0]$. We will use only the entries $e[i, j]$ for which $j \geq i-1$. we also use a table $\text{root}[i, j]$, for recording the root of the sub tree containing keys k_i, \dots, k_j . This table uses only the entries for which $1 \leq i \leq j \leq n$.
 - ❖ We will need one other table for efficiency. Rather than compute the value of $w(i, j)$ from scratch every time we are computing $e[i, j]$. We keep these values in a table $w[1..n+1, 0..n]$. For the base case, we compute $w[i, i-1] = q_{i-1}$ for $1 \leq i \leq n$.
 - ❖ For $j \geq i$, we compute :
- $$w[i, j] = w[i, j-1] + p_i + q_j$$

Pseudo code:**OPTIMAL—BST(p,q,n)**

```

For i    1 to n+1
    do e[i,i-1] <- qi-1
    do w[i,i-1] <- qi-1
For l    1 to n
    do for i <- 1 to n-l+1
        do j <- i+l-1
        e[i,j] <- ∞
        w[i,j] <- w[i,j-1]+pi+qj
        For r <- i to j
            do t <- e[i,r-1]+e[r+1,j]+w[i,j]
            if t<e[i,j]
                then e[i,j] <- t
                root [i,j] <- r
Return e and root

```

Sample Example:

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The initial tables look like this:

		main table							root table				
		0	1	2	3	4			0	1	2	3	4
1		0	0.1				1			1			
2			0	0.2			2				2		
3				0	0.4		3					3	
4					0	0.3	4						4
5						0	5						

Let us compute $C(1, 2)$:

$$C(1, 2) = \min \left\{ \begin{array}{l} k=1: C(1, 0) + C(2, 2) + \sum_{s=1}^2 p_s = 0 + 0.2 + 0.3 = 0.5 \\ k=2: C(1, 1) + C(3, 2) + \sum_{s=1}^2 p_s = 0.1 + 0 + 0.3 = 0.4 \end{array} \right\} = 0.4.$$

Thus, out of two possible binary trees containing the first two keys, **A** and **B**, the root of the optimal tree has index 2 (i.e., it contains **B**), and the average number of comparisons in a successful search in this tree is 0.4.

We will arrive at the following final tables using the recurrence equations:

main table					
	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table					
	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

Thus, the average number of key comparisons in the optimal tree is equal to 1.7. Since $R(1, 4) = 3$, the root of the optimal tree contains the third key, i.e., **C**. Its left sub tree is made up of keys **A** and **B**, and its right sub tree contains just key **D**. To find the specific structure of these sub trees, we find first their roots by consulting the root table again. Since $R(1, 2) = 2$, the root of the optimal tree containing **A** and **B** is **B**, with **A** being its left child (and the root of the one-node tree: $R(1, 1) = 1$). Since $R(4, 4) = 4$, the root of this one-node optimal tree is its only key **D**.

Result and Analysis:

Every time we work on an entry $e[i, j]$ with $j - i = k$, we know that all the entries $e[i_0, j_0]$ with $j_0 - i_0 < k$ have already been computed. Note that the recursive formula we use to compute $e[i, j]$ only involves entries $e[i_0, j_0]$ with $j_0 - i_0 < k$. So they are all ready, and we can compute $e[i, j]$ in time $O(j - i)$. It is easy to check that the total running time is $\Theta(n^3)$.

Source Code:

```
#include <bits/stdc++.h>
using namespace std;

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

int optimalSearchTree(int keys[], int freq[], int n){
```

```

int cost[n][n];

// 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];
}

int sum(int freq[], int i, int j){
    int s = 0;
    for (int k = i; k <= j; k++)
        s += freq[k];

    return s;
}

int main(){

    srand((unsigned)time(0));

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

```

```
int keys[] = {10, 12, 20};
int freq[] = {34, 8, 50};

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

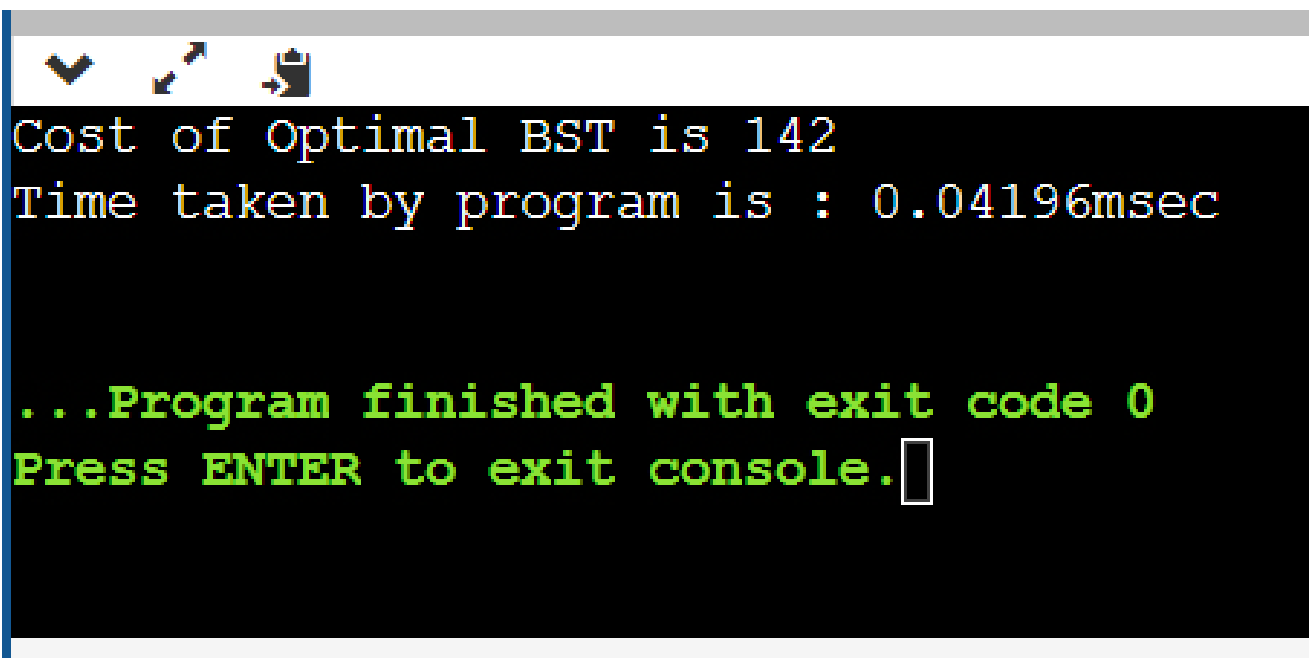
cout << "Cost of Optimal BST is " << optimalSearchTree(keys, freq, n);

auto end = chrono::high_resolution_clock::now();
double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
time_taken *= 1e-9 * 1000;

cout << "\nTime taken by program is : " << time_taken << setprecision(6);
cout << "msec" << endl;

return 0;
}
```

Output:



```
Cost of Optimal BST is 142
Time taken by program is : 0.04196msec

...Program finished with exit code 0
Press ENTER to exit console.
```

Viva Questions

1. What is OBST?

Ans.

As we know that in binary search tree, the nodes in the left subtree have lesser value than the root node and the nodes in the right subtree have greater value than the root node. The frequency and key-value determine the overall cost of searching a node.

2. Give any application of OBST.

Ans.

A binary search tree is one of the most important data structures in computer science. One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion

3. Can OBST be solved using Greedy Approach?

Ans.

Yes, we try to optimize the cost.

4. What is balanced BST?

Ans.

A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.

To check if a tree is height-balanced, get the height of left and right subtrees. Return true if difference between heights is not more than 1 and left and right subtrees are balanced, otherwise return false.

5. What is the best case for OBST?

Ans.

In best case, The binary search tree is a balanced binary search tree. Height of the binary search tree becomes $\log(n)$. So, Time complexity of BST Operations = **$O(\log n)$** .

Binomial Coefficient

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 a n -element set.

1) Optimal Substructure

The value of $C(n, k)$ can be recursively calculated using the following standard formula for Binomial Coefficients.

$$C(n, k) = C(n-1, k-1) + C(n-1, k)$$

$$C(n, 0) = C(n, n) = 1$$

2) Overlapping Subproblems

It should be noted that the above function computes the same subproblems again and again.

$$\begin{aligned} \binom{i}{n} &= \frac{i(i-1)(i-2)\cdots(i-(n-1))}{n(n-1)(n-2)(n-3)\cdots 2 \cdot 1} = \frac{1}{n!} \prod_{k=0}^{n-1} (i-k) = \left(\prod_{k=1}^n \frac{1}{k} \right) \prod_{k=0}^{n-1} (i-k) \\ \left(\prod_{k=1}^n \frac{1}{k} \right) \prod_{k=0}^{n-1} (i-k) &= \left(\prod_{k=1}^n \frac{1}{k} \right) \prod_{k=1}^n (i-(k-1)) = \prod_{k=1}^n \frac{i+1-k}{k} \Rightarrow \binom{i}{n} = \prod_{k=1}^n \frac{i-k+1}{k} \end{aligned}$$

1) Binomial Coefficients

In probability and statistics applications, you often need to know the total possible number of certain outcome combinations. For example, you may want to know how many ways a 2-card BlackJack hand can be dealt from a 52 card deck, or you may need to know the number of possible committees of 3 people that can be formed from a 12-person department, etc. The *binomial coefficient* (often referred to as " n choose k ") will provide the number of combinations of k things that can be formed from a set of n things. The binomial coefficient is written mathematically as:

$$\binom{n}{k}$$

which we refer to as " n choose k ".

Binomial coefficients can be defined recursively:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \qquad \binom{n}{0} = \binom{n}{n} = 1$$

Individually, write a *recursive* function named `choose(int n, int k)` that will compute and return the value of the binomial coefficient. Then compare your function to your partner's, and together (i) come up with a function implementation you both agree on, and (ii) write it as a C++ function on the computer.

Instructions:

Compared to a brute force recursive algorithm that could run exponential, the dynamic programming algorithm runs typically in quadratic time. The recursive algorithm ran in exponential time while the iterative algorithm ran in linear time. The space cost does increase, which is typically the size of the table. Frequently, the whole table does not have to store.

Computing a Binomial Coefficient

Computing binomial coefficients is non-optimization problem but can be solved using dynamic programming.

Binomial coefficients are represented by $C(n, k)$ or $\binom{n}{k}$ and can be used to represent the coefficients of a binomial:

$$(a + b)^n = C(n, 0)a^n + \dots + C(n, k)a^{n-k}b^k + \dots + C(n, n)b^n$$

The recursive relation is defined by the prior power

$$C(n, k) = C(n-1, k-1) + C(n-1, k) \text{ for } n > k > 0$$

$$\text{IC } C(n, 0) = C(n, n) = 1$$

Dynamic algorithm constructs a $n \times k$ table, with the first column and diagonal filled out using the IC.

Construct the table:

	k					
	0	1	2	...	k-1	k
0	1					
1	1	1				
2	1	2	1			
n	.					
.						
.						
k	1					1
.						
.						

.		
n-1	1	$C(n-1, k-1)$
n	1	$C(n, k)$

The table is then filled out iteratively, row by row using the recursive relation.

Pseudo code:

Binomial(n, k)

for $i \leftarrow 0$ to n do // fill out the table row wise

for $i = 0$ to $\min(i, k)$ do

if $j=0$ or $j=i$ then $C[i, j] \leftarrow 1$ // IC

else $C[i, j] \leftarrow C[i-1, j-1] + C[i-1, j]$ // recursive relation

return $C[n, k]$

Sample Example:

Ex: $C(3,2)$ Answer should be 3.

$i=0$

$j=0$ to 0

$j=0 : C[0,0]=1$

$i=1$

$j=0$ to 1

$j=0 : C[1,0]=1$

$j=1 : C[1,1]=1$

$i=2$

$j=0$ to 2

$j=0 : C[2,0]=1$

$$j=1: C[2,1]=C[1,0]+C[1,1]=1+1=2$$

$$j=1: C[2,1]=1$$

i=3

j= 0 to 2 (min(i,k) => k=2,i=3)

$$j=0 : C[3,0]=1$$

$$j=1: C[3,1]= C[2,0]+ C[2,1]= 1+2=3$$

$$j=2: C[3,2]= C[2,1]+ C[2,2]= 2+1=3$$

return C[3,2] =>3

Result and Analysis:

The cost of the algorithm is filling out the table. Addition is the basic operation. Because $k \leq n$, the sum needs to be split into two parts because only the half the table needs to be filled out for $i < k$ and remaining part of the table is filled out across the entire row.

$A(n, k) = \text{sum for upper triangle} + \text{sum for the lower rectangle}$

$$= \sum_{i=1}^k \sum_{j=1}^{i-1} 1 + \sum_{i=1}^n \sum_{j=1}^k 1$$

$$= \sum_{i=1}^k (i-1) + \sum_{j=1}^n k$$

$$= (k-1)k/2 + k(n-k) \in \Theta(nk)$$

Source Code:

```
#include <bits/stdc++.h>
using namespace std;

int min(int a, int b);

int binomialCoeff(int n, int k){

    int C[n + 1][k + 1];
```

```
int i, j;

// Calculate value of Binomial Coefficient in bottom up manner
for (i = 0; i <= n; i++) {

    for (j = 0; j <= min(i, k); j++) {

        // Base Cases
        if (j == 0 || j == i)
            C[i][j] = 1;

        // Calculate value using previously stored values
        else
            C[i][j] = C[i - 1][j - 1] + C[i - 1][j];

    }
}
return C[n][k];
}

// A utility function to return minimum of two integers
int min(int a, int b) {
    return (a < b) ? a : b;
}

int main(){

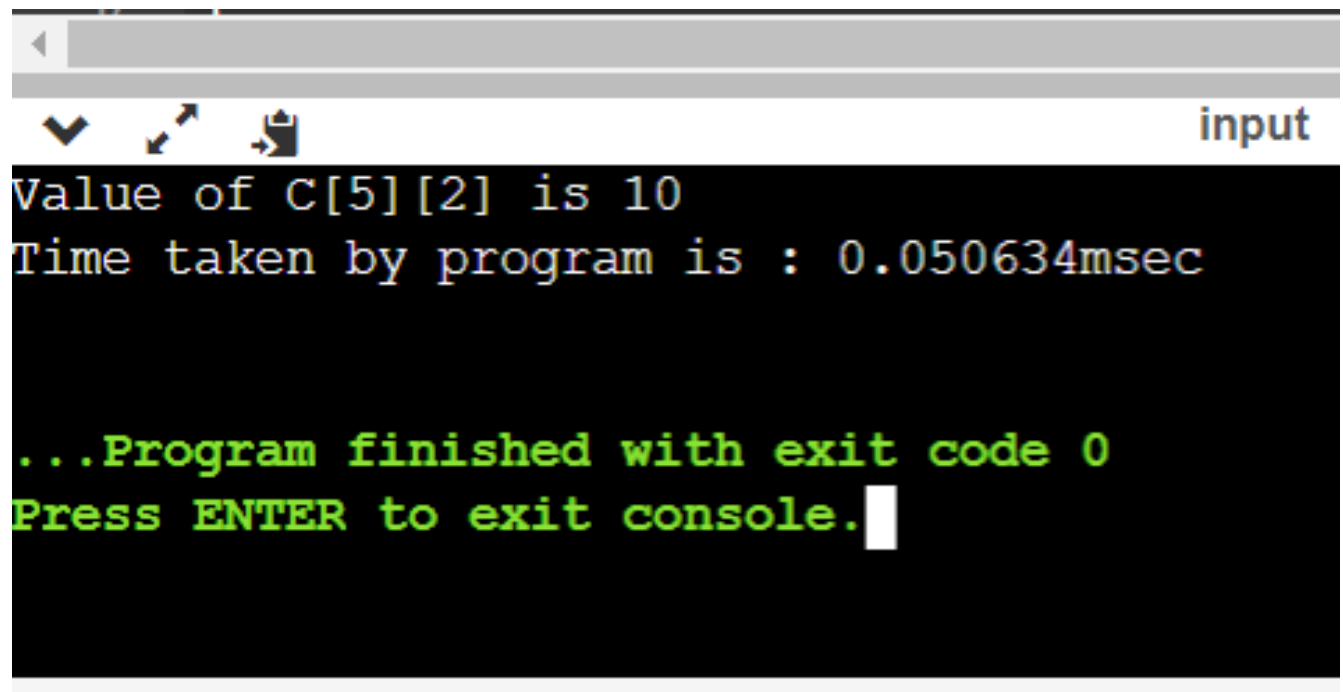
    srand((unsigned)time(0));

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    int n = 5, k = 2;

    cout << "Value of C[" << n << "][" << k << "] is " << binomialCoeff(n, k);

    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9 * 1000;
    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;
    return 0;
}
```

Output:A screenshot of a console window with a dark background. The window has a title bar at the top with a left arrow icon and a label 'input'. Below the title bar is a toolbar with three icons: a checkmark, a double-headed arrow, and a document with an arrow. The main area of the console displays the following text in a monospaced font: 'Value of C[5][2] is 10' in yellow, 'Time taken by program is : 0.050634msec' in yellow, '...Program finished with exit code 0' in green, and 'Press ENTER to exit console.' in green. A white cursor is positioned at the end of the last line.

```
Value of C[5][2] is 10
Time taken by program is : 0.050634msec

...Program finished with exit code 0
Press ENTER to exit console.
```

Viva Questions**1. What is Time Complexity of Binomial Coefficient?**

Ans.

$O(N^2 + Q)$, because we are precomputing the binomial coefficients up to nCn . This operation takes $O(N^2)$ time and then $O(1)$ time to answer each query.

2. What is Space Complexity of Binomial Coefficient?

Ans.

$O(N^2)$, for storing the precomputed results of binomial coefficients.

3. What is Binomial Coefficient?

Ans.

Binomial Coefficient is used heavily to solve combinatorics problems. Let's say you have some n different elements and you need to pick k elements. So, if you want to solve this problem you can easily write all the cases of choosing k elements out of n elements.

4. Why is Binomial Coefficient required?

Ans.

This problem can be easily solved using binomial coefficient. More than that, this problem of choosing k elements out of n different elements is one of the way to define binomial coefficient $\mathbf{n C k}$. Binomial coefficient can be easily calculated using the given formula:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Since now we are good at the basics, we should find ways to calculate this efficiently.

5. What is Naive Approach for finding Binomial Coefficient?

Ans.

This approach isn't too **naive** at all. Consider you are asked to find the number of ways of choosing 3 elements out of 5 elements. So you can easily find $\mathbf{n!}$, $\mathbf{k!}$ and $\mathbf{(n-k)!}$ and put the values in the given formula. This solution takes only **O(N) time** and **O(1) space**. But sometimes your factorial values may overflow so we need to take care of that. This approach is fine if we want to calculate a single binomial coefficient. But many times we need to calculate many binomial coefficients. So, it's better to have them precomputed. We will find out how to find the binomial coefficients efficiently.