

CHAROTAR UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEVANG PATEL INSTITUTE OF ADVANCE TECHNOLOGY AND RESEARCH

Department of Computer Science & Engineering

CS328: Design and Analysis of Algorithms

Name: Rushik Rajeshbhai Rathod

ID: 20DCS103

Semester: 5

Academic Year: 2022

Aim 1: Implement and analyze algorithms given below.**1.1 Factorial (Iterative and Recursive).****Code - Iterative:**

```
#include <stdio.h>
#include <iostream>
using namespace std;

int main()
{
    long long n, fact = 1, i = 0, count = 0;
    cout << "Enter a number: ";
    cin >> n;
    for (i = 1; i <= n; i++)
    {
        fact = fact * i;
        count++;
    }
    cout << "Factorial: " << fact << endl;
    cout << "Count: " << count << endl;
    return 0;
}
```

Code - Recursive:

```
#include <stdio.h>
#include <iostream>
using namespace std;

int count = 0;

int factorial(int n)
{
    if (n < 0)
        return 0;
    if (n == 0)
        return 1;
    else
    {
        count++;
        return n * factorial(n - 1);
    }
}
```

```

int main()
{
    long long n;
    cout << "Enter a number: ";
    cin >> n;
    cout << "Factorial: " << factorial(n) << endl;
    cout << "Count: " << count << endl;
    return 0;
}

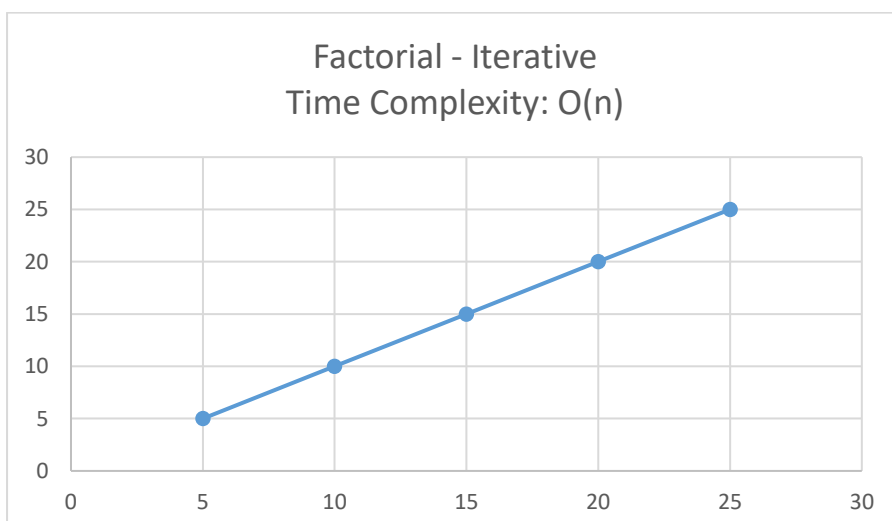
```

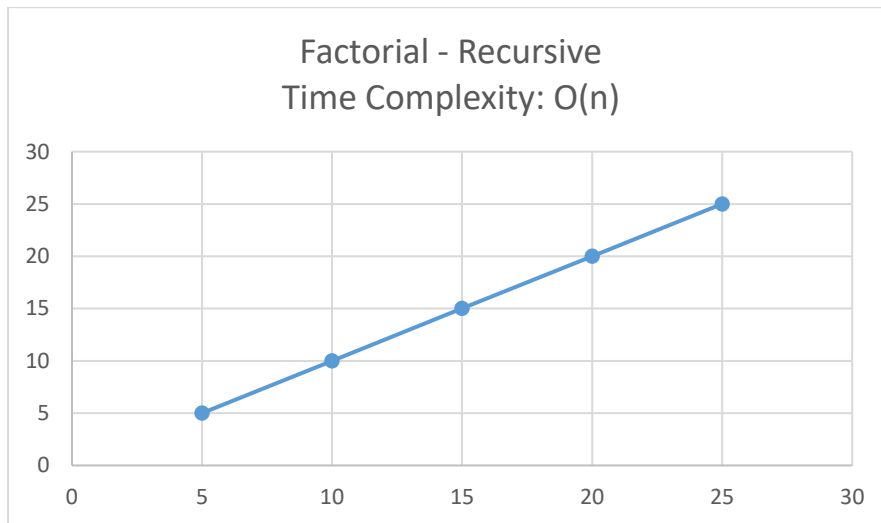
Table:**Iterative:**

Array Elements	Count
5	5
10	10
15	15
20	20
25	25

Recursive:

Array Elements	Count
5	5
10	10
15	15
20	20
25	25

Graph:**Factorial - Iterative:**

Factorial - Recursive:**Conclusion:**

Iterative code to find the factorial of a number traverses through all the elements starting from '1 to n' only once, which results in the $O(n)$ time complexity;

Recursive function to find the factorial of a number calls itself resulting in traversing all the elements starting from 'n to 1' and giving time complexity of $O(n)$.

1.2 Euclidean algorithm.

Code:

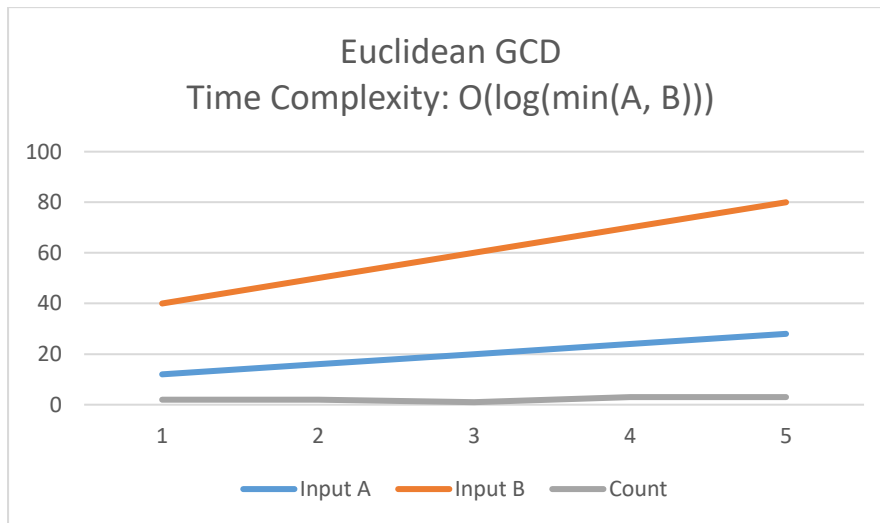
```
#include <iostream>
using namespace std;

int counter = 0;
GCD(int a, int b)
{
    if (a == 0)
        return b;
    else
    {
        counter++;
        GCD(b % a, a);
    }
}

int main()
{
    int num1, num2;
    cin >> num1 >> num2;
    cout << "GCD: " << GCD(min(num1, num2), max(num1, num2));
    cout << "\nCount: " << counter;
    return 0;
}
```

Table:

Input A	Input B	Count
12	40	2
16	50	2
20	60	1
24	70	3
28	80	3

Graph:**Conclusion:**

In the Euclidean's theorem, the log of the minimum of both the numbers will be the time complexity.

1.3 Matrix addition and Matrix multiplication (Iterative).

Code:**Matrix addition:**

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

int main()
{
    int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int b[2][3] = {{7, 8, 9}, {1, 2, 3}};
    int c[2][3];

    int count = 0;
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            count++;
            c[i][j] = a[i][j] + b[i][j];
        }
    }
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cout << c[i][j] << endl;
        }
    }
    cout << "Count: " << count;
    return 0;
}
```

Matrix multiplication:

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

int main()
{
    cout << "Enter m*n for matrix 1: " << endl;
    int m, n;
    cin >> m >> n;

    cout << "Enter x*y for matrix 1: " << endl;
    int x, y;
```

```
    cin >> x >> y;
    int a[m][n];
    int b[x][y];
    int ans[m][y];

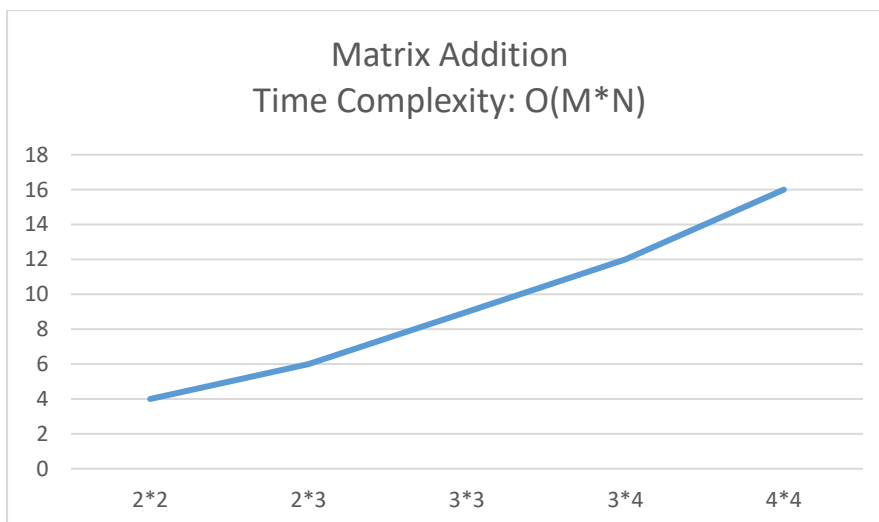
    int count = 0;
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cin >> a[i][j];
        }
    }
    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < y; j++)
        {
            cin >> b[i][j];
        }
    }
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < y; j++)
        {
            ans[i][j] = 0;
        }
    }
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < y; j++)
        {
            for (int k = 0; k < n; k++)
            {
                count++;
                ans[i][j] = ans[i][j] + (a[i][k] * b[k][j]);
            }
        }
    }
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < y; j++)
        {
            cout << ans[i][j] << " ";
        }
    }
    cout << "Count: " << count;
    return 0;
}
```

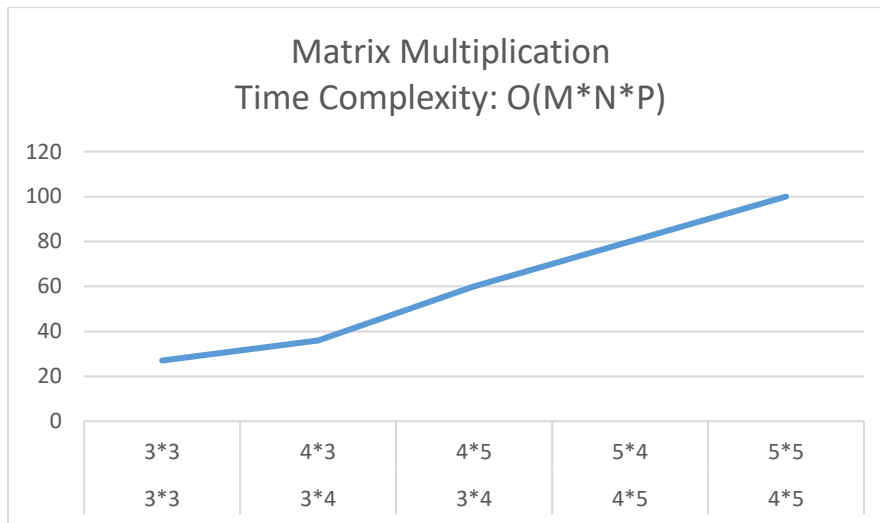

Table:**Matrix addition:**

Dimension (M*N)	Count
2*2	4
2*3	6
3*3	9
3*4	12
4*4	16

Matrix multiplication:

Matrix A: (M*N)	Matrix B: (N*P)	Count
3*3	3*3	27
3*4	4*3	36
3*4	4*5	60
4*5	5*4	80
4*5	5*5	100

Graph:**Matrix addition:**

Matrix multiplication:**Conclusion:**

In matrix addition theorem, the outer for loop will traverse through 'm' number of elements which is equal to the number of rows and the inner loop will run 'n' number of times which is equal to the number of columns in a matrix. As a result, we get the time complexity of $O(m*n)$.

In matrix multiplication theorem, the number of columns of first matrix must be same as the number of rows of matrix B. For example, the dimension of two matrices should be A: $2*3$ and B: $3*4$. Here, in this algorithm we have to run three for loops, as a result the time complexity of matrix multiplication algorithm will be $O(m*n*p)$, where m = number of rows of matrix A, n = number of columns of matrix A = number of rows of matrix B, p = number of columns of matrix B.

1.4 Find a subset of a given set $S = \{s_1, s_2, \dots, s_n\}$ of n positive integers whose sum is equal to a given positive integer d . For example, if $S = \{1, 2, 5, 6, 8\}$ and $d = 9$ there are two solutions $\{1, 2, 6\}$ and $\{1, 8\}$. A suitable message is to be displayed if the given problem instance doesn't have a solution.

Code:

```
#include <iostream>
#include <vector>

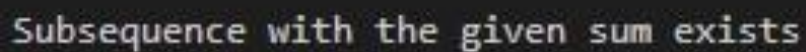
using namespace std;

bool subsequenceSum(vector<int> const &arr, int n, int k)
{
    if (k == 0)
    {
        return true;
    }
    if (n < 0 || k < 0)
    {
        return false;
    }
    bool include = subsequenceSum(arr, n - 1, k - arr[n]);
    bool exclude = subsequenceSum(arr, n - 1, k);
    return include || exclude;
}

int main()
{
    vector<int> A = {1, 2, 5, 6, 8};
    int k = 9;

    int n = A.size();
    if (subsequenceSum(A, n - 1, k))
    {
        cout << "\nSubsequence with the given sum exists\n"
              << endl;
    }
}
```

```
    else
    {
        cout << "\nSubsequence with the given sum does not exist" << endl;
    }
    return 0;
}
```

Output:A screenshot of a terminal window with a dark background. The text "Subsequence with the given sum exists" is displayed in a light-colored, monospaced font.**Conclusion:**

From this practical, I learnt about creating a dynamic programming strategy to check if a given subsequence can yield a specified sum. A recursive program is created and is used to check the possibilities that were fruitful for us.

Aim 2: Implement and analyze algorithms given below.**2.1 Bubble sort.****Code:**

```
#include <iostream>
using namespace std;

void display(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
}

int bubbleSort(int arr[], int n)
{
    static int count;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - 1 - i; j++)
        {
            count++;
            if (arr[j] > arr[j + 1])
            {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
    return count;
}

int main()
{
    int n;
    cout << "Enter the size of array: ";
    cin >> n;

    int arr[n];
    cout << "Enter array elements: ";
    for(int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    cout << "Array: ";
```

```

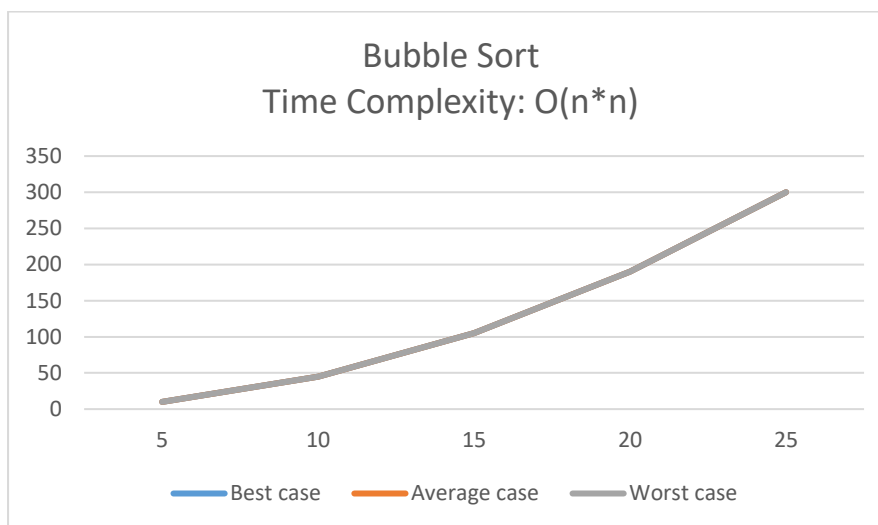
display(arr, n);
cout << endl;
int count = bubbleSort(arr, n);

cout << "Sorted: ";
display(arr, n);
cout << endl;
cout << "Count: " << count;
return 0;
}

```

Table:

Array Elements	Best case	Average case	Worst case
5	10	10	10
10	45	45	45
15	105	105	105
20	190	190	190
25	300	300	300

Graph:**Conclusion:**

In bubble sort algorithm, the outer loop runs for n times and the inner loop runs for $n-1-i$ times in best and worst case scenario. As a result, bubble sort consumes the time complexity of $O(n*n)$.

In average case scenario, assuming that the half of the array is already sorted. Now the number of swaps performed will be $n/2$ and the number of comparison will be n resulting in the time complexity of $O(n/2 * n)$ which is almost $O(n*n)$.

2.2 Selection sort.

Code:

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

int counter = 0;

void swap(int *xp, int *yp)
{
    counter++;
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    for (i = 0; i < n - 1; i++)
    {
        min_idx = i;
        for (j = i + 1; j < n; j++)
        {
            counter++;
            if (arr[j] < arr[min_idx])
            {
                min_idx = j;
            }
        }
        swap(&arr[min_idx], &arr[i]);
    }
}

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

int main()
{
    int n;
    cout << "Enter size of array: ";
    cin >> n;
    int arr[n];
```

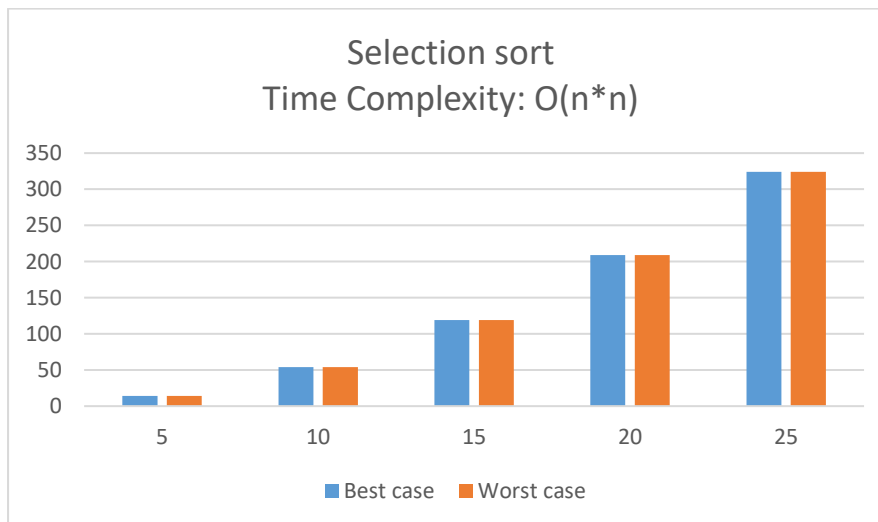
```

    cout << "Enter array elements: ";
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    int arrsize = sizeof(arr) / sizeof(arr[0]);
    selectionSort(arr, arrsize);
    cout << endl << "Sorted array: ";
    printArray(arr, arrsize);
    cout << "Counter: " << counter;
    return 0;
}

```

Table:

Array Elements	Best case	Worst case
5	14	14
10	54	54
15	119	119
20	209	209
25	324	324

Graph:**Conclusion:**

In selection sort algorithm, the outer for loop will run for n times and the inner for loop will run for almost n times, which results in the time complexity of $O(n*n)$.

2.3 Insertion sort.

Code:

```
#include <stdio.h>
#include <iostream>
using namespace std;

int count = 0;

void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            count++;
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
        count++;
    }
}

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

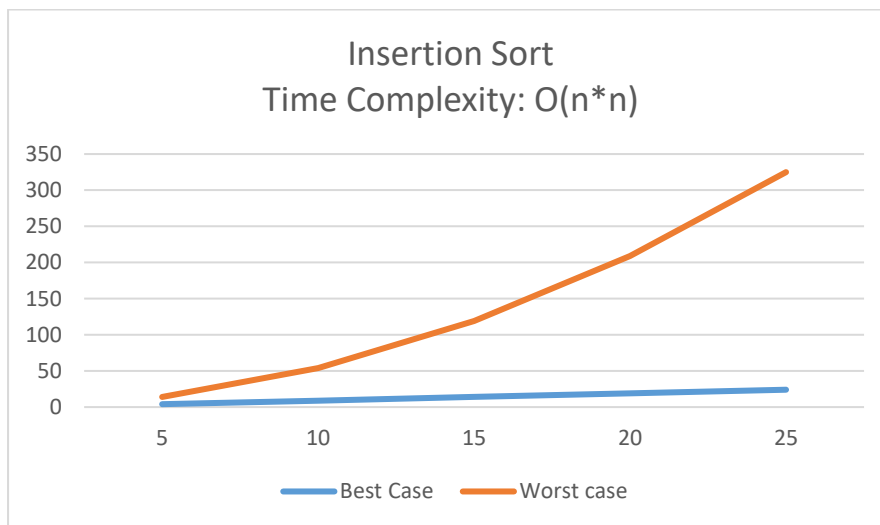
int main()
{
    int n;
    cout << "Enter the length: ";
    cin >> n;
    int arr[n];
    cout << "Enter an array elements: ";
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    cout << "Array: ";
    printArray(arr, n);
}
```

```
insertionSort(arr, n);  
cout << "Sorted: ";  
printArray(arr, n);  
cout << "Count: " << count << endl;  
return 0;  
}
```

Table:

Array Elements	Best Case	Worst case
5	4	14
10	9	54
15	14	119
20	19	209
25	24	325

Graph:**Conclusion:**

In the insertion sort algorithm, the outer for loop will run for $n-1$ times means total $n-1$ passes will be done. Additionally, during the entire execution of the algorithm n elements will be compared for n times which results in the time complexity of $O(n*n)$.

Aim 3: Divide and Conquer strategy.**Implement and perform analysis of worst case of Merge sort and Quick sort.****Compare both algorithms.****Merge sort:****Code:**

```
#include <iostream>
using namespace std;

int counter = 0;

void merge(int arr[], int p, int q, int r)
{
    counter++;
    int n1 = q - p + 1;
    int n2 = r - q;

    int L[n1], M[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[p + i];

    for (int j = 0; j < n2; j++)
        M[j] = arr[q + 1 + j];

    int i = 0, j = 0, k = p;

    while (i < n1 && j < n2)
    {
        counter++;
        if (L[i] <= M[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = M[j];
            j++;
        }
        k++;
    }
}
```

```
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}

while (j < n2)
{
    arr[k] = M[j];
    j++;
    k++;
}

void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        counter++;
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

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

int main()
{
    int n;
    cout << "Enter size of array: ";
    cin >> n;
    int arr[n];

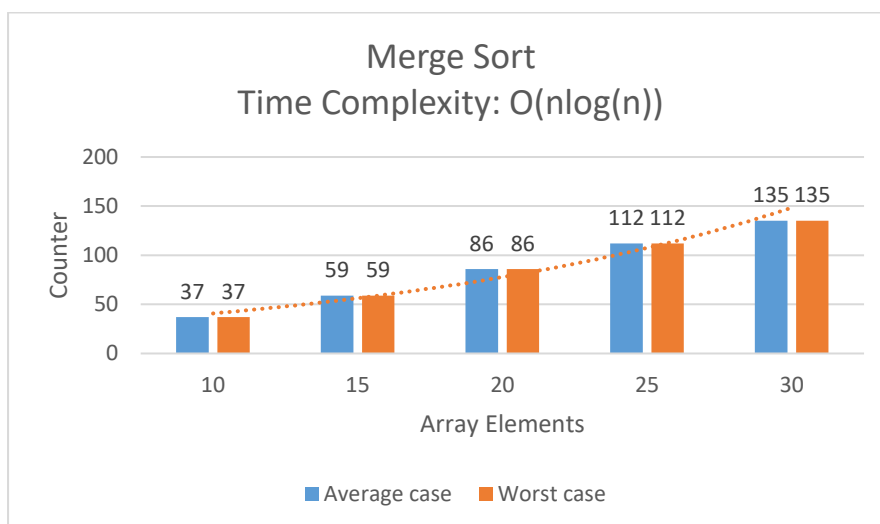
    for (int i = 0; i < n; i++)
    {
        arr[i] = i + 1;
    }
    int size = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, size - 1);
```

```
cout << "Sorted array: ";  
printArray(arr, size);  
cout << "Counter:" << counter;  
return 0;  
}
```

Table:

Array Elements	Average case	Worst case
10	37	37
15	59	59
20	86	86
25	112	112
30	135	135

Graph:**Conclusion:**

In the merge sort algorithm, the time complexity of all three cases will be same as all the elements will be divided till all the elements becomes single elements. Therefore the time complexity of merge sort will be $O(n \log(n))$ which is based on divide and conquer algorithm.

Quick sort:**Code:**

```
#include <cstdlib>
#include <stdio.h>
#include <iostream>
using namespace std;

int counter = 0;

void display(long long arr[], long long n)
{
    for (long long i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
}

void swap(long long arr[], long long pos1, long long pos2)
{
    long long temp;
    temp = arr[pos1];
    arr[pos1] = arr[pos2];
    arr[pos2] = temp;
}

long long partition(long long arr[], long long low, long long high, long
long pivot)
{
    long long i = low;
    long long j = low;

    while (i <= high)
    {
        counter++;
        if (arr[i] > pivot)
        {
            i++;
        }
        else
        {
            swap(arr, i, j);
            i++;
            j++;
        }
    }
    return j - 1;
}
```

```
void quickSort(long long arr[], long long low, long long high)
{
    if (low < high)
    {
        counter++;
        long long pivot = arr[high];
        long long pos = partition(arr, low, high, pivot);

        quickSort(arr, low, pos - 1);
        quickSort(arr, pos + 1, high);
    }
}

int main()
{
    long long n;
    cout << "Enter the size of array";
    cin >> n;
    long long arr[n];

    // for worst case
    // for (int i = 0; i < n; i++)
    // {
    //     arr[i] = i + 1;
    // }

    // for average case
    for (long long i = 0; i < n; i++)
    {
        arr[i] = rand();
    }

    cout << endl
         << "Array: ";
    display(arr, n);

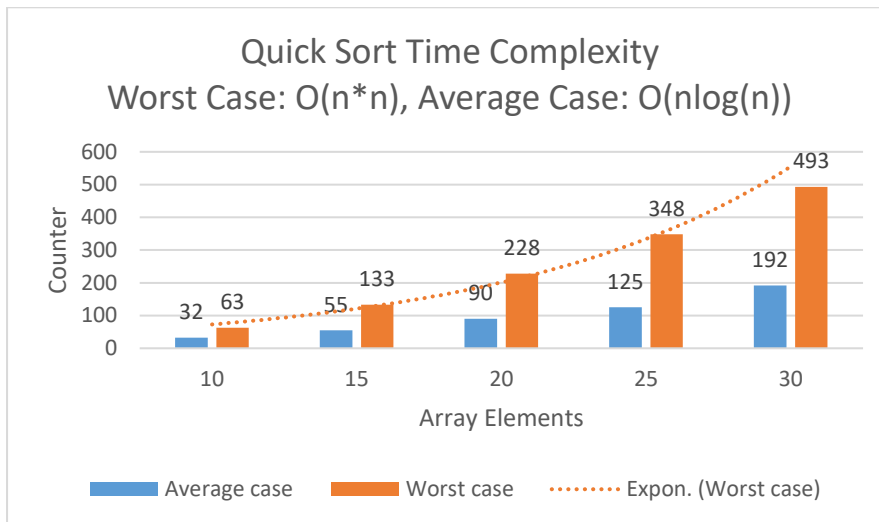
    quickSort(arr, 0, n - 1);
    cout << endl
         << "The sorted array is: ";
    display(arr, n);

    cout << endl
         << "Counter is: " << counter << endl;

    return 0;
}
```

Table:

Array Elements	Average case	Worst case
10	32	63
15	55	133
20	90	228
25	125	348
30	192	493

Graph:**Conclusion:**

In the merge sort algorithm, the time complexity of all three cases will be same as all the elements will be divided till all the elements becomes single elements. Therefore the time complexity of merge sort will be $O(n \log(n))$ which is based on divide and conquer algorithm.

Concluding comparison:

Merge sort algorithm performs same number of comparisons regardless of the nature of the input array. As a result, worst case time complexity is same as the best case time complexity and average case time complexity.

In contrast, in quick sort algorithm, the choice of pivot element plays an important role. For instance, while choosing the right most element as a pivot and having the reversely sorted array will result in unbalanced partitions. Therefore, quick sort has worst time complexity of $O(n*n)$. In addition, the average case time complexity of quick sort is $O(n \log(n))$.

3.2 Implement the program to find X^Y using divide and conquer strategy and print the total number of multiplications required to find X^Y .

Test the program for following test cases:

Test Case	X	Y
1	2	6
2	7	25
3	5	34

Code:

```
#include <stdio.h>
#include <iostream>
using namespace std;

double power(long x, long y)
{
    if (y == 0)
        return 1;
    else if (y % 2 == 0)
    {
        return power(x * x, y / 2);
    }
    else
    {
        return x * power(x * x, y / 2);
    }
}

int main()
{
    long x, y;
    cout << "Enter the number and it's power : ";
    cin >> x >> y;
    double result = power(x, y);
    cout << "Result : " << result;
    return 0;
}
```

Output:**Test Case 1:**

```
PS F:\B.Tech. Sem 5\DAA Practical> cd "f:\B.Tech. Sem 5\DAA Practical"
Enter the number and it's power : 2 6
Result : 64
PS F:\B.Tech. Sem 5\DAA Practical> █
```

Test Case 2:

```
PS F:\B.Tech. Sem 5\DAA Practical> cd "f:\B.Tech. Sem 5\DAA Practical"
Enter the number and it's power : 7 25
Result : -6.15944e+016
PS F:\B.Tech. Sem 5\DAA Practical> █
```

Test Case 3:

```
PS F:\B.Tech. Sem 5\DAA Practical> cd "f:\B.Tech. Sem 5\DAA Practical"
Enter the number and it's power : 5 34
Result : -5.13066e+010
PS F:\B.Tech. Sem 5\DAA Practical> █
```

Conclusion:

From this practical, I learned the basics of divide and conquer strategy. I also tried to optimize the 'Power of element problem' using divide and conquer strategy.

Aim 4: Greedy approach.

4.1 A cashier at any mall needs to give change of an amount to customers many times in a day. Cashier has multiple number of coins available with different denominations which is described by a set C. Implement the program for a cashier to find the minimum number of coins required to find a change of a particular amount A. Output should be the total number of coins required of given denominations. Check the program for following test cases:

Test Case	Coin Denomination C	Amount A
1	₹1, ₹2, ₹3	₹ 5
2	₹18, ₹17, ₹5, ₹1	₹ 22
3	₹100, ₹25, ₹10, ₹5, ₹1	₹ 289

Is the output of Test case 2 is optimal? Write your observation.

Code:

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
    int n, amount;
    cout << "Enter the coins for denomination : ";
    cin >> n;
    int C[n];
    cout << "Enter denominations coins : ";
    for (int i = 0; i < n; i++)
    {
        cin >> C[i];
    }
}
```

```
    cout << "Enter amount : ";
    cin >> amount;

    sort(C, C + n, greater<int>());
    vector<int> solution;

    int x = C[0];
    int j = 0;

    while (amount > 0 && j < n)
    {
        if (amount >= x && amount > 0)
        {
            solution.push_back(x);
            amount -= x;
        }
        else
        {
            j++;
            x = C[j];
        }
    }

    cout << "Total coins required : " << solution.size();
    cout << "\nCoins are : ";
    for (auto x : solution)
    {
        cout << x << " ";
    }
    return 0;
}
```

Output:**Test Case 1:**

```
PS F:\B.Tech. Sem 5\DAA Practical> cd
Enter the coins for denomination : 3
Enter denominations coins : 1 2 3
Enter amount : 5
Total coins required : 2
Coins are : 3 2
```

Test Case 2:

```
PS F:\B.Tech. Sem 5\DAA Practical> cd
Enter the coins for denomination : 4
Enter denominations coins : 18 17 5 1
Enter amount : 22
Total coins required : 5
Coins are : 18 1 1 1 1
```

Test Case 3:

```
PS F:\B.Tech. Sem 5\DAA Practical> cd "f:\
Enter the coins for denomination : 5
Enter denominations coins : 100 25 10 5 1
Enter amount : 289
Total coins required : 10
Coins are : 100 100 25 25 25 10 1 1 1 1
```

Conclusion:

This problem is a variation of 'Coin Change Problem'. When the only coin present is of rupees 1, in that case, time complexity becomes $O(A)$ where A is the amount to be paid.

4.2 Let S be a collection of objects with profit-weight values. Implement the fractional knapsack problem for S assuming we have a sack that can hold objects with total weight W. Check the program for following test cases:

Test Case	S	Profit Weight Values	W
1	{A,B,C}	Profit:(1,2,5) Weight: (2,3,4)	5
2	{A,B,C,D,E,F,G}	Profit:(10,5,15,7,6,18,3) Weight: (2,3,5,7,1,4,1)	15
3	{A,B,C,D,E,F,G}	A:(12,4), B:(10,6), C:(8,5), D:(11,7), E:(14,3), F:(7,1), G:(9,6)	18

Code:

```
#include <stdio.h>

#include <iostream>
#include <vector>
#include <iterator>
#include <map>
#include <cmath>

#define int long long int
#define F first
#define S second
#define pb push_back

using namespace std;

int32_t main()
{
    int items;
    cout << "Enter number of items present : ";
    cin >> items;
    cout << "Enter total weight of sack : ";
    int weight;
    cin >> weight;
    vector<int> p(items), w(items);
```

```
cout << "Enter profits : ";
for (int i = 0; i < items; i++)
{
    cin >> p[i];
}

cout << "Enter weights : ";
for (int i = 0; i < items; i++)
{
    cin >> w[i];
}

multimap<double, int, greater<int>> map;
double pDivW;
for (int i = 0; i < items; i++)
{
    pDivW = round(double(p[i]) / double(w[i]) * 100) / 100;
    map.insert(make_pair(pDivW, i));
}

cout << "Pi/Wi : index\n";
double max_profit = 0;

while (weight)
{
    int index, key;
    double frac_weight;
    multimap<double, int>::iterator itr;
    itr = map.begin();

    while (itr != map.end())
    {
        cout << itr->first << " : " << itr->second << "\n";
        index = itr->second;
        key = itr->first;
        if (w[index] <= weight)
        {
            max_profit += p[index];
        }
    }
}
```

```
        weight -= w[index];
        itr++;
    }
    else
    {
        cout << weight << "\n\n";
        frac_weight = double(weight) /
                      double(w[index]);
        max_profit += (double(itr->first) * weight);
        weight -= weight;
        if (weight == 0)
            break;
    }
}
}
cout << "Maximum Profit : " << max_profit << endl << endl;
return 0;
}
```


Output:**Test Case 1:**

```
Enter number of items present : 3
Enter total weight of sack : 5
Enter profits : 1 2 5
Enter weights : 2 3 4
Pi/Wi : index
1.25 : 2
0.5 : 0
1
Maximum Profit : 5.5
```

Test Case 2:

```
Enter number of items present : 7
Enter total weight of sack : 15
Enter profits : 10 5 15 7 6 18 3
Enter weights : 2 3 5 7 1 4 1
Pi/Wi : index
6 : 4
5 : 0
4.5 : 5
3 : 2
3 : 6
1.67 : 1
2
Maximum Profit : 55.34
```

Test Case 3:

```
Enter number of items present : 7
Enter total weight of sack : 18
Enter profits : 12 10 8 11 14 7 9
Enter weights : 4 6 5 7 3 1 6
Pi/Wi : index
7 : 5
4.67 : 4
3 : 0
1.67 : 1
1.6 : 2
4
Maximum Profit : 49.4
```

Conclusion:

From this practical, I learned to solve fractional knapsack problem which is based on getting maximum profit items in the knapsack having fixed capacity.

4.3 Suppose you want to schedule N activities in a Seminar Hall. Start time and Finish time of activities are given by pair of (si,fi) for ith activity. Implement the program to maximize the utilization of Seminar Hall. (Maximum activities should be selected.)

Test Case	Number of activities (n)	(si, fi)
1	9	(1,2), (1,3), (1,4), (2,5), (3,7), (4,9), (5,6), (6,8), (7,9)
2	11	(1,4), (3,5), (0,6), (3,8), (5,7), (5,9), (6,10), (8,12), (8,11), (12,14), (2,13)

Code:

```
#include <stdio.h>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int n = 0, count = 1;
    cout << "Enter no of activities : ";
    cin >> n;
    int arr_start[n];
    int arr_end[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter start and end time of activity " << i + 1 << " : ";
        cin >> arr_start[i];
        cin >> arr_end[i];
    }
    int i = 0;
    for (int j = 1; j < n; j++)
    {
        // ending of first should be less than or equal to the second for count
        if (arr_end[i] <= arr_start[j])
        {
            count++;
            i = j;
        }
    }
}
```

```
    }  
}  
cout << endl << "Max no of activity possible : " << count << endl;  
return 0;  
}
```

Output:**Test Case 1:**

```
Enter no of activities : 9  
Enter start and end time of activity 1 : 1 2  
Enter start and end time of activity 2 : 1 3  
Enter start and end time of activity 3 : 1 4  
Enter start and end time of activity 4 : 2 5  
Enter start and end time of activity 5 : 3 7  
Enter start and end time of activity 6 : 4 9  
Enter start and end time of activity 7 : 5 6  
Enter start and end time of activity 8 : 6 8  
Enter start and end time of activity 9 : 7 9  
  
Max no of activity possible : 4  
PS F:\B.Tech. Sem 5\DAA Practical> █
```

Test Case 2:

```
Enter no of activities : 11  
Enter start and end time of activity 1 : 1 4  
Enter start and end time of activity 2 : 3 5  
Enter start and end time of activity 3 : 0 6  
Enter start and end time of activity 4 : 3 8  
Enter start and end time of activity 5 : 5 7  
Enter start and end time of activity 6 : 5 9  
Enter start and end time of activity 7 : 6 10  
Enter start and end time of activity 8 : 8 12  
Enter start and end time of activity 9 : 8 11  
Enter start and end time of activity 10 : 12 14  
Enter start and end time of activity 11 : 2 13  
  
Max no of activity possible : 4  
PS F:\B.Tech. Sem 5\DAA Practical> █
```

Conclusion:

The problem is based on getting maximum activities completed such that maximum amount of the schedule can be utilized. In the problem definition, we are given to solve this problem for seminar hall. Here, greedy approach is used to solve the problem which allows to complete maximum activities so that maximum schedule time of seminar hall can be used.

Aim 5: Dynamic programming.

5.1 Implement a program which has BNMCOEF() function that takes two parameters n and k and returns the value of Binomial Coefficient $C(n, k)$. Compare the dynamic programming implementation with recursive implementation of BNMCOEF(). (In output, entire table should be displayed.)

Test Case	n	k
1	5	2
2	11	6
3	12	5

Is the output of Test case 2 is optimal? Write your observation.

Code:

```
#include <iostream>
using namespace std;

int BNMCOEFIterative(int n, int k)
{
    int mat[n + 1][k + 1];

    for (int i = 0; i <= n; i++)
    {
        for (int j = 0; j <= k; j++)
        {
            // i = n, j = k
            if (i == j || j == 0)
            {
                mat[i][j] = 1;
            }
            else if (i > j)
            {
                mat[i][j] = mat[i - 1][j - 1] + mat[i - 1][j];
            }
            else // n < k
            {
```

```
        mat[i][j] = 0;
    }
}

cout << "Matrix..." << endl;
for (int i = 0; i <= n; i++)
{
    for (int j = 0; j <= k; j++)
    {
        cout << mat[i][j] << "    ";
    }
    cout << endl;
}

return mat[n][k];
}

int BNMCOEF(int n, int k)
{
    int mat[n][k];
    for (int i = 0; i <= n; i++)
    {
        for (int j = 0; j <= k; j++)
        {
            if (n == k || k == 0)
            {
                mat[i][j] = 1;
                return 1;
            }
            else if (n > k)
            {
                return (BNMCOEF(n - 1, k - 1) + BNMCOEF(n - 1, k));
            }
            else // n < k
            {
                mat[i][j] = 0;
            }
        }
    }
}
```

```
    }  
  }  
}  
  
int main()  
{  
    int n, k;  
    cout << "Enter n and k : ";  
    cin >> n >> k;  
    cout << endl  
        << "Recursive..." << endl;  
    cout << "C(" << n << ", " << k << ") = " << BNMCOEF(n, k) << endl;  
  
    cout << endl  
        << "Iterative..." << endl;  
  
    int res = BNMCOEFIterative(n, k);  
    cout << "C(" << n << ", " << k << ") = " << res << endl  
        << endl;  
  
    return 0;  
}
```

Output:**Test Case 1:**

```
Enter n and k : 5 2
```

```
Recursive...
```

```
C(5,2) = 10
```

```
Iterative...
```

```
Matrix...
```

```
1 0 0
```

```
1 1 0
```

```
1 2 1
```

```
1 3 3
```

```
1 4 6
```

```
1 5 10
```

```
C(5,2) = 10
```

Test Case 2:

```

Enter n and k : 11 6

Recursive...
C(11,6) = 462

Iterative...
Matrix...
1  0  0  0  0  0  0
1  1  0  0  0  0  0
1  2  1  0  0  0  0
1  3  3  1  0  0  0
1  4  6  4  1  0  0
1  5 10 10  5  1  0
1  6 15 20 15  6  1
1  7 21 35 35 21  7
1  8 28 56 70 56 28
1  9 36 84 126 126 84
1 10 45 120 210 252 210
1 11 55 165 330 462 462
C(11,6) = 462

```

Test Case 3:

```

Enter n and k : 12 5

Recursive...
C(12,5) = 792

Iterative...
Matrix...
1  0  0  0  0  0
1  1  0  0  0  0
1  2  1  0  0  0
1  3  3  1  0  0
1  4  6  4  1  0
1  5 10 10  5  1
1  6 15 20 15  6
1  7 21 35 35 21
1  8 28 56 70 56
1  9 36 84 126 126
1 10 45 120 210 252
1 11 55 165 330 462
1 12 66 220 495 792
C(12,5) = 792

```

Conclusion:

In this practical, a program to compute Binomial Coefficient $C(n,k)$ is written in two different methods which includes iterative method and recursive method. The concept of dynamic programming is used and created a matrix from which one can easily get the answer of binomial co-efficient.

5.2 Implement the program 4.2 using Dynamic Programing. Compare Greedy and Dynamic approach. (0 -1 knap sack problem)

Test Case	S	Profit Weight Values	W
1	{A,B,C}	Profit:(1,2,5) Weight: (2,3,4)	5
2	{A,B,C,D,E,F,G}	Profit:(10,5,15,7,6,18,3) Weight: (2,3,5,7,1,4,1)	15
3	{A,B,C,D,E,F,G}	A:(12,4), B:(10,6), C:(8,5), D:(11,7), E:(14,3), F:(7,1), G:(9,6)	18

Code:

```
#include <iostream>
#include <stdio.h>
#include <vector>

using namespace std;

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

int knapSack(int W, int weights[], int values[], int n)
{
    int i, w;
    vector<vector<int>> K(n + 1, vector<int>(W + 1));

    for (i = 0; i <= n; i++)
    {
        for (w = 0; w <= W; w++)
        {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (weights[i - 1] <= w)
                K[i][w] = max(values[i - 1] +
                               K[i - 1][w - weights[i - 1]],
```



```
                K[i - 1][w]));
            else
                K[i][w] = K[i - 1][w];
        }
    }
    return K[n][W];
}

int main()
{
    int sack;
    cout << "Enter number of sack : ";
    cin >> sack;

    int values[sack];
    int weights[sack];

    cout << "Enter values and corresponding weights..." << endl;
    for (int i = 0; i < sack; i++)
    {
        cin >> values[i] >> weights[i];
    }

    int w;
    cout << "Enter total weight : ";
    cin >> w;

    int n = sizeof(values) / sizeof(values[0]);
    cout << knapSack(w, weights, values, n);
    return 0;
}
```

Output:**Test Case 1:**

```
Enter number of sack : 3
Enter values and corresponding weights...
1 2
2 3
5 4
Enter total weight : 5
Maximum profit : 5
```

Test Case 2:

```
Enter number of sack : 7
Enter values and corresponding weights...
10 2
5 3
15 5
7 7
6 1
18 4
3 1
Enter total weight : 15
Maximum profit : 54
```

Test Case 3:

```
Enter number of sack : 7
Enter values and corresponding weights...
12 4
10 6
8 5
11 7
14 3
7 1
9 6
Enter total weight : 18
Maximum profit : 44
```

Conclusion:

From this practical, I learned to implement 0 - 1 knap sack problem which is based on getting maximum profit items in the knapsack having fixed capacity by utilizing the dynamic programming approach.

5.3 Given a chain $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices, where for $i=1,2,\dots,n$ matrix A_i with dimensions. Implement the program to fully parenthesize the product A_1, A_2, \dots, A_n in a way that minimizes the number of scalar multiplications. Also calculate the number of scalar multiplications for all possible combinations of matrices.

Test Case	N	Matrix Dimension
1	3	A1: 3*5, A2: 5*6, A3: 6*4
2	6	A1: 30*35, A2: 35*15, A3: 15*5, A4: 5*10, A5: 10*20, A6: 20*25

Code:

```
#include <stdio.h>
#include <iostream>
#include <set>
#include <cstring>

#define F first
#define S second
#define pb push_back

using namespace std;

int dp[100][100];

// Function for matrix chain multiplication
int matrixChainMemoised(int *p, int i, int j)
{
    if (i == j)
    {
        return 0;
    }
    if (dp[i][j] != -1)
    {
        return dp[i][j];
    }

    dp[i][j] = 2147483647;
```

```
        for (int k = i; k < j; k++)
        {
            dp[i][j] = min(
                dp[i][j], matrixChainMemoised(p, i, k) +
                matrixChainMemoised(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 matrixChainMemoised(p, i, j);
}

int main()
{
    int arr_size;
    cin >> arr_size;
    int arr[arr_size];

    for (int i = 0; i < arr_size; i++)
    {
        cin >> arr[i];
    }

    int n = sizeof(arr) / sizeof(arr[0]);
    memset(dp, -1, sizeof dp);
    cout << "Minimum number of multiplications is: " <<
    MatrixChainOrder(arr, n) << endl;
    return 0;
}
```

Output:**Test Case 1:**

```
4
3 5 6 4
Minimum number of multiplications is: 162
```

Test Case 2:

```
7
30 35 15 5 10 20 25
Minimum number of multiplications is: 15125
```

Conclusion:

In this practical I learnt about deriving the formula for matrix chain multiplication. We find the proper range of i to k and $k+1$ to j to properly apply dp on it to reach our result.

5.4 Aim: Implement a program to print the longest common subsequence for the following strings:

Test Case	String 1	String 2
1	ABCDAB	BDCABA
2	EXPONENTIAL	POLYNOMIAL
3	LOGARITHM	ALGORITHM

Code:

```
#include <cstdlib>
#include <cstring>
#include <stdio.h>
#include <iostream>
using namespace std;

void lcs(char *X, char *Y, int m, int n)
{
    int L[m + 1][n + 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]);
        }
    }

    int index = L[m][n];

    char lcs[index + 1];
    lcs[index] = '\0';

    int i = m, j = n;
```

```
while (i > 0 && j > 0)
{
    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
    }

    else if (L[i - 1][j] > L[i][j - 1])
        i--;
    else
        j--;
}

cout << "LCS of " << X << " and " << Y << " : " << lcs;
}

int main()
{
    char X[] = "LOGARITHM";
    char Y[] = "ALGORITHM";
    int m = strlen(X);
    int n = strlen(Y);
    cout << endl;
    lcs(X, Y, m, n);
    cout << endl << endl;
    return 0;
}
```

Output:**Test Case 1:**

```
LCS of ABCDAB and BDCABA : BDAB
```

Test Case 2:

```
LCS of EXPONENTIAL and POLYNOMIAL : PONIAL
```

Test Case 3:

```
LCS of LOGARITHM and ALGORITHM : LGRITHM
```

Conclusion:

In this practical I learnt about creating a dynamic programming program about finding the longest common subsequence from two strings.

Aim 6: Graph

6.1 Aim: Write a program to detect cycles in a directed graph.

Code:

```
#include <stdio.h>
#include <iostream>
#include <list>

#define F first
#define S second
#define pb push_back

using namespace std;

class Graph
{
    int V;
    list<int> *adj;
    bool isCyclicUtil(int v, bool visited[], bool *rs);

public:
    Graph(int V);
    void addEdge(int v, int w);
    bool isCyclic();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}
```

```
bool Graph::isCyclicUtil(int v, bool visited[], bool *recStack)
{
    if (visited[v] == false)
    {
        visited[v] = true;
        recStack[v] = true;

        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            if (!visited[*i] && isCyclicUtil(*i, visited, recStack))
                return true;
            else if (recStack[*i])
                return true;
        }
    }
    recStack[v] = false;
    return false;
}

bool Graph::isCyclic()
{
    bool *visited = new bool[V];
    bool *recStack = new bool[V];
    for (int i = 0; i < V; i++)
    {
        visited[i] = false;
        recStack[i] = false;
    }

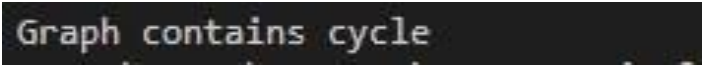
    for (int i = 0; i < V; i++)
        if (!visited[i] && isCyclicUtil(i, visited, recStack))
            return true;

    return false;
}
```

```
int main()
{
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    if (g.isCyclic())
        cout << "Graph contains cycle.";
    else
        cout << "Graph doesn't contain cycle.";

    return 0;
}
```

Output:A screenshot of a terminal window with a dark background. The text "Graph contains cycle" is displayed in a light-colored, monospaced font.**Conclusion:**

In this practical I learnt to detect a cycle in a directed graph. The algorithm is similar to a variation of DFS. The time complexity is same as that of DFS which is $O(V+E)$.

6.2 Aim: From a given vertex in a weighted graph, implement a program to find shortest paths to other vertices using Dijkstra's algorithm.

Test Case	Adjacency Matrix of graph	Start Vertex																																																																																	
1	<table><tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>0</td><td></td><td></td><td></td><td>2</td><td></td><td></td><td></td><td></td></tr><tr><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td>7</td><td></td></tr><tr><td>2</td><td></td><td></td><td></td><td></td><td>3</td><td></td><td></td><td></td></tr><tr><td>3</td><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>4</td><td></td><td></td><td>3</td><td></td><td></td><td></td><td>1</td><td>7</td></tr><tr><td>5</td><td></td><td></td><td></td><td></td><td></td><td></td><td>9</td><td></td></tr><tr><td>6</td><td></td><td>7</td><td></td><td></td><td>1</td><td>9</td><td></td><td></td></tr><tr><td>7</td><td></td><td></td><td></td><td></td><td>7</td><td></td><td></td><td></td></tr></table>		0	1	2	3	4	5	6	7	0				2					1							7		2					3				3	2								4			3				1	7	5							9		6		7			1	9			7					7				1
	0	1	2	3	4	5	6	7																																																																											
0				2																																																																															
1							7																																																																												
2					3																																																																														
3	2																																																																																		
4			3				1	7																																																																											
5							9																																																																												
6		7			1	9																																																																													
7					7																																																																														
2	<table><tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>0</td><td></td><td></td><td>2</td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>1</td><td>6</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>2</td><td>3</td><td>8</td><td></td><td></td><td>5</td><td></td><td></td><td></td></tr><tr><td>3</td><td></td><td>9</td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>4</td><td></td><td></td><td></td><td></td><td></td><td></td><td>1</td><td></td></tr><tr><td>5</td><td></td><td></td><td></td><td>7</td><td></td><td></td><td></td><td></td></tr><tr><td>6</td><td></td><td></td><td>9</td><td></td><td>4</td><td></td><td></td><td>3</td></tr><tr><td>7</td><td></td><td></td><td></td><td></td><td></td><td>1</td><td>6</td><td></td></tr></table>		0	1	2	3	4	5	6	7	0			2						1	6								2	3	8			5				3		9							4							1		5				7					6			9		4			3	7						1	6		3
	0	1	2	3	4	5	6	7																																																																											
0			2																																																																																
1	6																																																																																		
2	3	8			5																																																																														
3		9																																																																																	
4							1																																																																												
5				7																																																																															
6			9		4			3																																																																											
7						1	6																																																																												

Code:

```
#include <stdio.h>
#include <iostream>
#include <limits.h>
#define F first
#define S second
#define pb push_back
#define V 8
using namespace std;

int minDistance(int dist[], bool sptSet[])
{
    int min = INT_MAX, min_index;
```

```
        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min)
                min = dist[v], min_index = v;
        return min_index;
    }

void printSolution(int dist[])
{
    cout << "Vertex \t Distance from Source" << endl;
    for (int i = 0; i < V; i++)
        cout << i << " \t\t" << dist[i] << endl;
}

void dijkstra(int graph[V][V], int src)
{
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    dist[src] = 0;
    for (int count = 0; count < V - 1; count++)
    {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;

        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u]
+ graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    printSolution(dist);
}

int main()
{
    int graph[V][V];
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
```

```

        {
            cin >> graph[i][j];
        }
    }
    dijkstra(graph, 1);
    return 0;
}

```

Output:**Test Case 1:**

```

0 0 0 2 0 0 0 0
0 0 0 0 0 0 7 0
0 0 0 0 3 0 0 0
2 0 0 0 0 0 0 0
0 0 3 0 0 0 1 7
0 0 0 0 0 0 9 0
0 7 0 0 1 9 0 0
0 0 0 0 7 0 0 0
Vertex   Distance from Source
0                2147483647
1                0
2                11
3                2147483647
4                8
5                16
6                7
7                15

```

Test Case 1:

```

0 0 2 0 0 0 0 0
6 0 0 0 0 0 0 0
3 8 0 0 5 0 0 0
0 9 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 7 0 0 0 0
0 0 9 0 4 0 0 3
0 0 0 0 0 1 6 0
Vertex   Distance from Source
0                6
1                0
2                8
3                25
4                13
5                18
6                14
7                17

```

Conclusion:

In this practical I implemented Dijkstra algorithm for finding the shortest path from a given source. The algorithm's time complexity is $O(V^2)$ where V is the dimension of the adjacency matrix.

Aim 7: Backtracking

7.1 Aim: Implement a program to print all permutations of a given string.

Test Case	String
1	ACT
2	NOTE

Code:

```
#include <stdio.h>
#include <iostream>

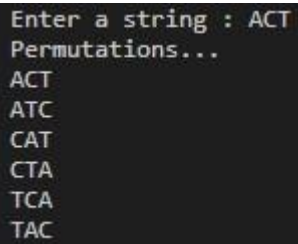
#define F first
#define S second
#define pb push_back

using namespace std;

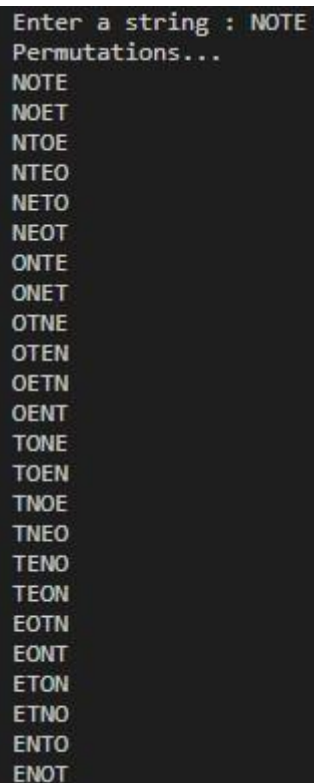
void permute(string a, int l, int r)
{
    if (l == r)
        cout << a << endl;
    else
    {
        for (int i = l; i <= r; i++)
        {
            swap(a[l], a[i]);
            permute(a, l + 1, r);
            swap(a[l], a[i]);
        }
    }
}

int main()
{
    string str;
    cout << "Enter a string : ";
    cin >> str;
    int n = str.size();
```

```
    cout << "Permutations..." << endl;
    permute(str, 0, n - 1);
    cout << endl;
    return 0;
}
```

Output:**Test Case 1:**A terminal window showing the execution of a program. The user enters 'ACT' when prompted 'Enter a string :'. The program outputs 'Permutations...' followed by a list of all permutations of 'ACT': ACT, ATC, CAT, CTA, TCA, and TAC.

```
Enter a string : ACT
Permutations...
ACT
ATC
CAT
CTA
TCA
TAC
```

Test Case 2:A terminal window showing the execution of a program. The user enters 'NOTE' when prompted 'Enter a string :'. The program outputs 'Permutations...' followed by a list of all 24 permutations of 'NOTE', starting with NOTE and ending with ENOT.

```
Enter a string : NOTE
Permutations...
NOTE
NOET
NTOE
NTEO
NETO
NEOT
ONTE
ONET
OTNE
OTEN
OETN
OENT
TONE
TOEN
TNOE
TNEO
TENQ
TEON
EOTN
EONT
ETON
ETNO
ENTO
ENOT
```

Conclusion:

In this practical I used backtracking principle to find all the permutations of a given string. The time complexity of the algorithm is $O(n!)$ since we are finding total permutations of a string of length n .

Aim 8: String Matching Algorithm

8.1 Aim: Suppose you are given a source string $S[0..n-1]$ of length n , consisting of symbols a and b . Suppose that you are given a pattern string $P[0..m-1]$ of length $m < n$, consisting of symbols a , b , and $*$, representing a pattern to be found in string S . The symbol $*$ is a “wild card” symbol, which matches a single symbol, either a or b . The other symbols must match exactly. The problem is to output a sorted list M of valid “match positions”, which are positions j in S such that pattern P matches the substring $S[j..j+|P|-1]$. For example, if $S = ababbab$ and $P = ab*$, then the output M should be $[0, 2]$. Implement a straightforward, naive algorithm to solve the problem.

Code:

```
#include <iostream>
#include <stdio.h>
#include <utility>
#include <vector>
#define F first
#define S second
#define pb push_back
using namespace std;

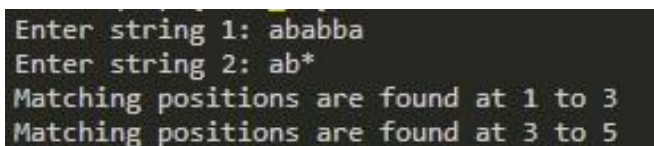
int main()
{
    cout << "Enter string 1: ";
    string S;
    cin >> S;
    int n = S.length();

    cout << "Enter string 2: ";
    string P;
    cin >> P;
    int m = P.length();

    vector<pair<int, int>> M;

    if (m < n)
    {
```

```
    for (int i = 0; i < n - m; i++)
    {
        int j;
        for (j = 0; j < m; j++)
        {
            if (S[i + j] != P[j] && P[j] != '*')
            {
                break;
            }
        }
        if (j == m)
        {
            M.pb(make_pair(i, i + j - 1));
        }
    }
}
else
{
    cout << "Length of pattern string is greater, not possible to
match the string to Source" << endl;
}
for (auto x : M)
{
    cout << "Matching positions are found at " << x.first + 1 << " to
" << x.second + 1 << endl;
}
return 0;
}
```

Output:

```
Enter string 1: ababba
Enter string 2: ab*
Matching positions are found at 1 to 3
Matching positions are found at 3 to 5
```

Conclusion:

In this practical I learnt about creating a pattern matching algorithm. The time complexity of this algorithm is $O(n*m)$ where n is the length of source string and m is the length of pattern string that is to be matched.

8.2 Aim: Implement Rabin karp algorithm and test it on the following test cases:

Test Case	String	Pattern
1	2359023141526739921	31415 q=13
2	ABAAABCDBBABCDDDEBCABC	ABC q=101

Code:

```

#include <iostream>
#include <stdio.h>
#include <string.h>

#define F first
#define S second
#define pb push_back

using namespace std;
#define d 19

void rabinKarp(char pattern[], char text[], int q)
{
    int m = strlen(pattern);
    int n = strlen(text);
    int i, j;
    int p = 0;
    int t = 0;
    int h = 1;

    for (i = 0; i < m - 1; i++)
        h = (h * d) % q;

    // Calculate hash value for pattern and text
    for (i = 0; i < m; i++)
    {
        p = (d * p + pattern[i]) % q;

```

```
        t = (d * t + text[i]) % q;
    }

    for (i = 0; i <= n - m; i++)
    {
        if (p == t)
        {
            for (j = 0; j < m; j++)
            {
                if (text[i + j] != pattern[j])
                    break;
            }

            if (j == m)
                cout << "Pattern is found at position: " << i + 1 << endl;
        }
        if (i < n - m)
        {
            t = (d * (t - text[i] * h) + text[i + m]) % q;

            if (t < 0)
                t = (t + q);
        }
    }
}

int main()
{
    char text[] = "ABAAABCDBBABCDDEBCABC";
    char pattern[] = "ABC";
    int q = 101;

    rabinKarp(pattern, text, q);

    return 0;
}
```

Test Case 1:**Input:**

```
char text[] = "2359023141526739921";  
char pattern[] = "31415";  
int q = 13;
```

Output:

```
Pattern is found at position: 7
```

Test Case 2:**Input:**

```
char text[] = "ABAAABCDBBABCDDDEBCABC";  
char pattern[] = "ABC";  
int q = 101;
```

Output:

```
Pattern is found at position: 5  
Pattern is found at position: 11  
Pattern is found at position: 19
```

Conclusion:

In this practical I learnt to implement Rabin-Karp algorithm and using the concept of hashes I ran the program for two test cases. The average and best case complexity of this algorithm is $O(m+n)$ and for worst case it is $O(m*n)$.