

Workbook

Data Structure Algorithm & Implementation

(CT – 157)



Name:	Maria Nasir	(SE-19007)
	Mahnoor	(SE-19013)
	Warda Haider	(SE-19016)
	Abdul Hannan	(SE-19041)
	Muhammad Daniyal Amir	(SE-19049)

Batch: 2019

Year: Second Year (3rd Semester)

Department: Software Engineering

Workbook

Data Structure Algorithm & Implementation

(CT – 157)



Prepared by

Engr. Sana Fatima

Lecturer- SE

Approved by

Chairman

Department of Software Engineering

Table of Contents

S. No.	Object	Page No.	Signature
01	To learn the basic concepts of Data Structure & Algorithms		
02	Array data structures & Operations i. Insertion ii. Deletion iii. Traversing		
03	Searching Algorithms i. Linear Search ii. Binary Search		
04	Sorting Algorithms i. Bubble Sort Algorithm ii. Quick Sort Algorithm		
05	Stack data structure & Operations i. Push ii. Pop		
06	Expression Evaluation through Stack Data Structure i. Infix ii. Postfix iii. Prefix		
07	Queue data structure & Operations i. Enqueue ii. Dequeue		
08	Recursive Algorithms (Recursion) Tower of Hanoi Problem		
09	Tree data structure & Operations i. Insertion ii. Deletion		
10	Tree Traversal Algorithms i. Inorder ii. Preorder iii. Postorder		
11	Graph data structure & Operations i. Adjacency List ii. Adjacency Matrix		
12	Graph Traversal Algorithms i. DFS ii. BFS		
13	Rat in a Maze Path finding problem		
14	N-Queen Problem		

Lab 01 Warda Haider SE-19016

To learn the basic concepts of Data Structure & Algorithms

1. Discuss the characteristics of Algorithms.

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.

Characteristics of an Algorithm

- **Unambiguous** – An algorithm should be clear and unambiguous. That is, an algorithm must lead to one meaning. Its steps and their inputs/outputs should be clear.
- **Input** – An algorithm should have 0 or more well-defined inputs.
- **Output** – An algorithm should have 1 or more well-defined outputs, and it should match or it should be according to the desired output.
- **Finiteness** – Algorithms must end or terminate after a finite number of steps.
- **Feasibility** – Should be feasible with the available resources.
- **Independent** – An algorithm should have step-by-step directions, and they should be independent of any programming code.

2. How do you differentiate linear and non- linear data structures? Also give some examples of both data structures.

Sr. No.	Key	Linear Data Structures	Non-linear Data Structures
1	Data Element Arrangement	In linear data structure, data elements are connected sequentially and each element can be traversed through a single run.	In non-linear data structure, data elements are connected hierarchically and at various levels they are present.

2	Levels	In linear data structure, all data elements are present at a single level.	In non-linear data structure, data elements are present at multiple levels.
3	Implementation complexity	Linear data structures are easier to implement.	Non-linear data structures are difficult to understand and implement as compared to linear data structures.
4	Traversal	Linear data structures can be traversed completely in a single run.	Non-linear data structures are not easy to traverse and needs multiple runs to be traversed completely.
5	Memory utilization	Linear data structures are not very memory friendly and are not utilizing memory efficiently.	Non-linear data structures uses memory very efficiently.
6	Time Complexity	Time complexity of linear data structure often increases with increase in size.	Time complexity of non-linear data structure often remain with increase in size.
7	Examples	Array, List, Queue, Stack.	Graph, Map, Tree.

3. Name some commonly used abstract data types.

Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of values and a set of operations.

Examples:

List ADT: Basic Operations are Traverse, Insert, Update, Delete, Search

Stack ADT: Basic Operations are Push, Pop

Queue ADT: Basic Operations are Enqueue, Dequeue

Binary Tree ADT: Basic Operations are Insert, Search, Delete, Preorder Traversal, Inorder Traversal, Postorder Traversal

Graph ADT: Add Vertex, Add Edge, Display Vertex

4. What do you understand by the term Algorithm Analysis? Discuss.

Algorithm analysis is used to estimate the resources required by an algorithm to solve or to compute a problem. Thus to compute complexity of an algorithm analysis is done. In other words algorithm analysis is used to determine the amount of time and space required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as **time complexity**, or volume of memory, known as **space complexity**.

Usually, the time required by an algorithm falls under three types –

- Best Case – Minimum time required for program execution.
- Average Case – Average time required for program execution.
- Worst Case – Maximum time required for program execution.

Lab 02

Array data structures

i. Insertion ii. Deletion iii. Traversing

Maria Nasir SE-007

1. Let an array named LA consist of 4 elements 2, 4, 6 & 8. Write down the code to traverse (or print) all elements in the array.

CODE:

```
#include <iostream>

using namespace std;

int main() {

    int LA [4] = {2,4,6,8};

    int start = 0, N = 4;

    while(start<N)

    {

        cout<<"The element present at index "<<start<<" is "<<LA[start]<<endl;

        start = start + 1;

    }

    return 0;

}
```

OUTPUT:

```
/tmp/pAX2zE7ZBC.o
The element present at index 0 is 2
The element present at index 1 is 4
The element present at index 2 is 6
The element present at index 3 is 8
```

2. Use insertion algorithm to add an element (Give implementation)

i. 1 at index 0

CODE:

```
#include <iostream>

using namespace std;

int main() {

    int LA [4] = {2,4,6,8}, k, item, j, i, N = 4;

    cout<<"enter the item to insert: ";

    cin>>item;

    cout<<"enter the index no to insert: ";

    cin>>k;

    j=N-1;

    While (j>=k)

    {

        LA[j+1]=LA[j];

        j=j-1;
```



```

    }

    LA[k]=item;

    N=N+1;

    // print the updated array

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

        cout<< LA[i] << " ";

    cout << endl;

    return 0;

}

```

OUTPUT:

```

enter the item to insert: 1
enter the index no to insert: 0
1 2 4 6 8

```

ii. 5 at index 2

CODE:

```

#include <iostream>

using namespace std;

int main() {

    int LA [4] = {2,4,6,8}, k, item, j, i, N = 4;

    cout<<"enter the item to insert: ";

    cin>>item;

    cout<<"enter the index no to insert: ";

    cin>>k;

```

```

    j=N-1;

    While (j>=k)
    {
        LA[j+1]=LA[j];
        j=j-1;
    }

    LA[k]=item;

    N=N+1;

    // print the updated array

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

        cout<< LA[i] << " ";

    cout << endl;

    return 0;

}

```

OUTPUT:

```

enter the item to insert: 5
enter the index no to insert: 2
2 4 5 6 8

```

iii. 3 at index 4

CODE:

```

#include <iostream>

using namespace std;

```

```

int main() {
    int LA [4] = {2,4,6,8}, k, item, j, i, N = 4;

    cout<<"enter the item to insert: ";

    cin>>item;

    cout<<"enter the index no to insert: ";

    cin>>k;

    j=N-1;

    While (j>=k)
    {
        LA[j+1]=LA[j];

        j=j-1;

    }

    LA[k]=item;

    N=N+1;

    // print the updated array

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

        cout << LA[i] << " ";

    cout << endl;

    return 0;
}

```

OUTPUT:

```
/tmp/pAX2zE7ZBC.o
enter the item to insert: 3
enter the index no to insert an item: 4
2 4 6 8 3
```

3. If the size of given array is 4 then calculate the Time complexity of insertion algorithm when ,

1. Insert element at the beginning of array
2. Insert element at the end of array
3. Insert element in the middle of array

Location of insertion	At the beginning of array	At the end of array	At the middle of array
Number of iteration	4	1	2
Big o notation	$O(n)$	$O(1)$	$O(n)$

4. Consider the array in question 2, Use deletion algorithm (Give implementation) to remove an element

i.1 at index 0

CODE:

```
#include <iostream>
```

```
using namespace std;
```

```

int main() {

    int LA [5] = {2,4,6,8,3};

    int k, item, j, i, N = 4;

    cout<<"enter the item to delete: ";

    cin>>item;

    cout<<"enter the position to delete an item: ";

    cin>>k;

    j=k;

    while(j<N)

    {

        LA[j]=LA[j+1];

        j=j+1;

    }

    N=N-1;

    // print the updated array

    for (i = 0; i < N+1; i++)

        cout<< LA[i] << " ";

    cout << endl;

    return 0;

}

```

OUTPUT:

```
/tmp/pAX2zE7ZBC.o
enter the item to delete: 1
enter the position to delete an item: 0
2 4 6 8
```

ii. 5 at index 2

CODE:

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int LA [5] = {2,4,6,8,3};
```

```
    int k, item, j, i, N = 4;
```

```
    cout<<"enter the item to delete: ";
```

```
    cin>>item;
```

```
    cout<<"enter the position to delete an item: ";
```

```
    cin>>k;
```

```
    j=k;
```

```
    while(j<N)
```

```
    {
```

```
        LA[j]=LA[j+1];
```

```
        j=j+1;
```

```
    }
```

```
    N=N-1;
```

```
    // print the updated array
```

```
    for (i = 0; i < N+1; i++)
```

```

        cout << LA[i] << " ";

        cout << endl;

        return 0;

}

```

OUTPUT:

```

enter the item to delete: 5
enter the position to delete an item: 2
2 4 6 8

```

iii. 3 at index 4

CODE:

```

#include <iostream>

using namespace std;

int main() {

    int LA [5] = {2,4,6,8,3};

    int k, item, j, i, N = 4;

    cout<<"enter the item to delete: ";

    cin>>item;

    cout<<"enter the position to delete an item: ";

    cin>>k;

    j=k;

    while(j<N)

    {

        LA[j]=LA[j+1];
    }
}

```

```

j=j+1;

}

N=N-1;

// print the updated array

for (i = 0; i < N+1; i++)

cout<< LA[i] << " ";

cout << endl;

return 0;

}

```

OUTPUT:

```

enter the item to delete: 3
"enter the position to delete an item: 4
2 4 6 8

```

5. If the size of given array is 4 then calculate the Time complexity of Deletion algorithm when ,

1. Delete element at the beginning of array
2. Delete element at the end of array
3. Delete element in the middle of array

Location of deletion	At the beginning of array	At the end of array	At the middle of array
Number of iteration	3	0	1
Big o notation	$O(n)$	$O(1)$	

Lab 03 Abdul Hannan SE-041

Searching Algorithms

i. Linear Search

ii. Binary Search

QUESTION # 01:

Assume array[] = {2,4,6,8,}. Give implementation of Linear search algorithm to ;

i. find element 8 in array

ii. find element 3 in array

Source Code (C++):

```
#include <iostream>
```

```
using namespace std;
```

```
int linearSearch(int DATA[], int N, int ITEM)
```

```
{
```

```
    DATA[N] = ITEM;
```

```
    int LOC = 0;
```

```
    while (DATA[LOC] != ITEM)
```

```
        LOC = LOC + 1;
```

```
    if (LOC == N)
```

```
        LOC = -1;
```

```
    return LOC;
```

```
}
```

```
int main()
```

```
{
```

```

int DATA[4] = { 2, 4, 6, 8, };

cout << "Location of 8 is " << linearSearch(DATA, 4, 8) << endl;

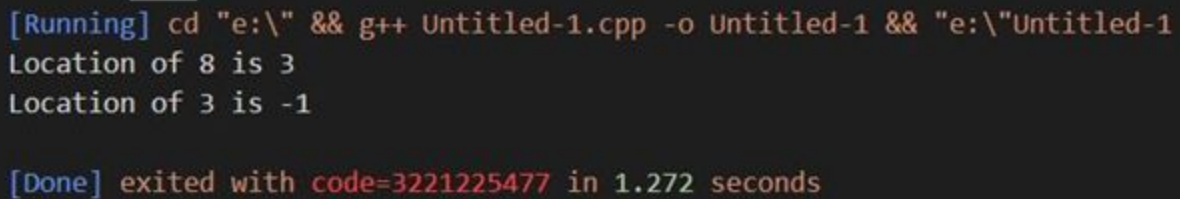
cout << "Location of 3 is " << linearSearch(DATA, 4, 3) << endl;

return 0;

}

```

Output:



```

[Running] cd "e:\\" && g++ Untitled-1.cpp -o Untitled-1 && "e:\\"Untitled-1
Location of 8 is 3
Location of 3 is -1

[Done] exited with code=3221225477 in 1.272 seconds

```

QUESTION # 02:

let assume A[] = {2,5,5,5,6,6,8,9,9,9} sorted array of integers containing duplicates, apply binary search algorithm to Count occurrences of a number provided, if the number is not found in the array report that as well. (Give implementation)

Source Code (C++):

```

#include <iostream>

using namespace std;

int binarySearch(int DATA[], int LB, int UB, int ITEM)
{
    int BEG = LB, END = UB;

    int MID = (BEG + END) / 2;

    int LOC;

    while (BEG <= END && DATA[MID] != ITEM)
    {

```

```

    if (ITEM < DATA[MID])

        END = MID - 1;

    else

        BEG = MID + 1;

        MID = (BEG + END) / 2;

    }

    if (DATA[MID] == ITEM)

        LOC = MID;

    else

        LOC = -1;

    return LOC;

}

int countOccurrences(int DATA[], int LB, int UB, int ITEM)

{

    int count = 0;

    int LOC = binarySearch(DATA, LB, UB, ITEM);

    if (LOC != -1)

    {

        count = 1;

        int LEFT = LOC - 1, RIGHT = LOC + 1;

        while (DATA[LEFT--] == ITEM)

```

```

        count++;

while (DATA[RIGHT++] == ITEM)

    count++;

return count;

}

}

int main()

{

    int DATA[] = { 2, 5, 5, 5, 6, 6, 8, 9, 9, 9, };

    int LB = 0, UB = 9;

    int ITEM = 5;

    int count = countOccurrences(DATA, LB, UB, ITEM);

    cout << "Count of " << ITEM << " is " << count << endl;

    return 0;

}

```

Output:

```

[Running] cd "e:\\" && g++ Untitled-1.cpp -o Untitled-1 && "e:\\"Untitled-1
Count of 5 is 3

[Done] exited with code=0 in 1.184 seconds

```

QUESTION # 03:

Compare linear & binary Search algorithms on the basis of time complexity , which one is

better in your opinion? Discuss.

“COMPARATIVE ANALYSIS OF LINEAR SEARCH AND BINARY SEARCH ON THE BASIS OF TIME COMPLEXITY”

The given chart compares the linear search and binary search algorithms on the basis of their time complexities:

	Linear Search	Binary Search
<i>General Time Complexity</i>	O(n) The linear search algorithm finds an item by comparing it with the items in an array in a linear fashion (from left to right) until an item matches.	O(log n) The binary search algorithm finds an item by keeping on dividing the array into half repeatedly (or recursively) and comparing the middle element with the item to be searched until it matches.
<i>Best Case Time Complexity</i>	O(1) If the first item is the item to be searched, then only one comparison is made and the algorithm takes constant time.	O(1) If the left middle item is the item to be searched, then only one comparison is made and the algorithm takes constant time.
<i>Worst Case Time Complexity</i>	O(n) If the last item is the item to be searched, then “n” comparisons are made	O(log n) If either the leftmost or the rightmost item is the item to be searched then “ $\lfloor \log n \rfloor$ ” comparisons are made.

Which one is better?

Pros of Linear Search:

1. It works well in case of both sorted and unsorted arrays.
2. It is time efficient in case of small arrays.
3. If the element to be searched is very likely to be at the beginning, then it is the recommended algorithm to be used.
4. Search operations can be done on multidimensional arrays as well.

Pros of Binary Search:

1. It is more time efficient.
2. It is a recommended choice if the data set is large. So for a million elements, linear search would take an average of 500,000 comparisons, whereas binary search would take 20.
3. Better to use when the item to be searched is likely to be at the middle or the end of the array

Conclusion:

Since under different conditions, the algorithms have their own benefits therefore, it cannot be decided which one of them is better to use. Hence, we consider both of the algorithms as the best of their kind.

Lab 04 Mahnoor SE-013

Sorting Operation in an Array

i. Bubble Sort Algorithm ii. Quick Sort Algorithm

1. Give implementation of bubble sort algorithm let's assume Array "A" is consisting of 6 elements which are stored in descending order.

CODE:

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

using namespace std;

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

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)
        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
```

```

        cout << arr[i] << " ";
        cout << endl;
    }
// Test code
int main()
{
    int arr[] = {90,80,70,60,50,40};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    cout<<"Sorted array: \n";
    printArray(arr, n);
    return 0;
}

```

OUTPUT:

```

D:\project\hello\bin\Debug\hello.exe
Sorted array:
40 50 60 70 80 90

Process returned 0 (0x0)   execution time : 0.501 s
Press any key to continue.

```

2. Re-write bubble sort algorithm for recursive function call. Also analyze the time complexity of the recursive bubble sort algorithm.

RECURSIVE BUBBLE SORT ALGORITHM:

DATA: Linear Array

N: Number of elements in array

RecursiveBubbleSort(DATA, N)

1. If $N = 1$ then

Return

2. Repeat for i = 0 to N-1
 - (a) If DATA[i] > DATA[i+1]
 - Interchange DATA[i] and DATA[i+1]
 [End of loop]
3. RecursiveBubbleSort(DATA, N-1)
4. End

Time Complexity of Recursive Bubble Sort Algorithm:

We are calling the same function recursively for each element of the array and inside the function, we are looping till the given length of the array,

So Time complexity is $O(n \wedge n) = O(n \wedge 2)$.

3. Implement the Quick sort Algorithm on Array A[]={ 9,7,5,11,12,2,14,3,10,6}

CODE:

```
#include <iostream>

using namespace std;

void swap(int* a, int* b)    // Swap two elements - Utility function
{
    int t = *a;
    *a = *b;
    *b = t;}

// partition the array using last element as pivot
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1);
    for (int j = low; j <= high- 1; j++)
    {
        //if current element is smaller than pivot, increment the low element
        //swap elements at i and j
        if (arr[j] <= pivot)
        {
            i++; // increment index of smaller element
```

```

        swap(&arr[i], &arr[j]);
    }
}
swap(&arr[i + 1], &arr[high]);
return (i + 1);
}

//quicksort algorithm
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    { //partition the array
        int pivot = partition(arr, low, high);
        //sort the sub arrays independently
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}

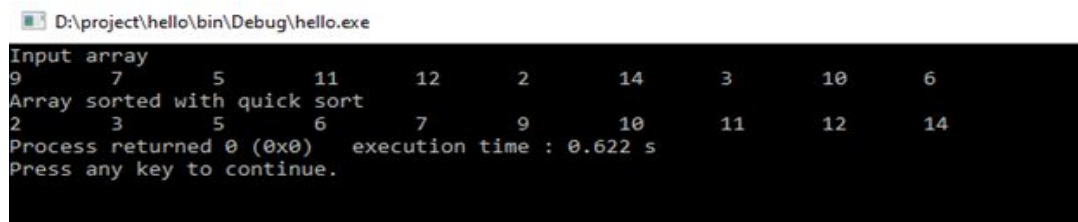
void displayArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout<<arr[i]<<"\t";
}

int main()
{
    int arr[] = {9,7,5,11,12,2,14,3,10,6};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout<<"Input array"<<endl;
    displayArray(arr,n);
}

```

```
cout<<endl;
quickSort(arr, 0, n-1);
cout<<"Array sorted with quick sort"<<endl;
displayArray(arr,n);
return 0;}
```

OUTPUT:



```
D:\project\hello\bin\Debug\hello.exe
Input array
9 7 5 11 12 2 14 3 10 6
Array sorted with quick sort
2 3 5 6 7 9 10 11 12 14
Process returned 0 (0x0) execution time : 0.622 s
Press any key to continue.
```

4. Compare the two sorting algorithms (discuss in this session) and discuss their advantages and disadvantages.

Comparison Parameter	Bubble Sort	Quick Sort
Method	Swaps of two adjacent elements in order to put them in the right place.	The array is partitioned around a pivot element and is not necessarily always partitioned into two halves. It can be partitioned in any ratio
Best case and worst case time complexity	$O(n)$ when the array is already sorted (best case). $O(n^2)$, when array is sorted in reverse order (worst case)	$O(n \cdot \log n)$ when the array is partitioned into equal subarrays (best case). $O(n^2)$, if partitioning leads to unbalanced subarrays.
Performance	It is slower and iterative.	It is faster and recursive.
Time Consumption	Less Time Consumption to run an algorithm	More Time Consumption to run an algorithm
Coding	Complex	Simple

BUBBLE SORT ALGORITHM

Advantages:

- It is popular and easy to implement.
- Elements are swapped in place without using additional temporary storage, so the space requirement is at a minimum.
- Performs greatly when the array is almost sorted.

Disadvantages:

- It does not deal well with a list containing a huge number of items.
- The bubble sort requires n -squared processing steps for every n number of elements to be sorted.
- The bubble sort is mostly suitable for academic teaching but not for real-life applications.

QUICK SORT ALGORITHM

Advantages:

- It is able to deal well with a huge list of items.
- It requires only $n \log n$ time to sort n items.
- The quick sort produces the most effective and widely used method of sorting a list of any item size.

Disadvantages:

- It is recursive. Especially, if recursion is not available, the implementation is extremely complicated.
- Its worst-case performance is similar to average performances of the bubble, insertion or selections sorts and requires quadratic (i.e., n^2) time in the worst-case.
- It is fragile, i.e. a simple mistake in the implementation can go unnoticed and cause it to perform badly.

Lab 05 Muhammad Daniyal Aamir SE-049

Stack data structure & Operations

i. Push ii. Pop

QUESTION # 01:

Give implementations of Push & Pop Algorithms with underflow & Overflow exceptions.

Source Code (C++):

```
# include <iostream>
```

```
using namespace std;
```

```
struct stack
```

```
{
```

```
    int MAXSIZE;
```

```
    int TOP;
```

```
    int *DATA;
```

```
};
```

```
struct stack* createStack(int MAXSIZE)
```

```
{
```

```
    struct stack *newStack = (struct stack *)malloc(sizeof(struct stack));
```

```
    newStack->TOP = -1;
```

```
    newStack->MAXSIZE = MAXSIZE;
```

```
    newStack->DATA = (int *)malloc(sizeof(int) * MAXSIZE);
```

```
    return newStack;
```

```
}
```

```
void push(struct stack *stack, int ITEM)
{
    if (stack->TOP == stack->MAXSIZE - 1)
        cout << "Stack Overflow!" << endl;
    else
        stack->DATA[++stack->TOP] = ITEM;
}
```

```
int pop(struct stack *stack)
{
    if (stack->TOP == -1)
        cout << "Stack Underflow!" << endl;
    else
    {
        int ITEM = stack->DATA[stack->TOP--];
        return ITEM;
    }
}
```

```
int traverse(struct stack *stack)
{
    for (int i = 0; i <= stack->TOP; i++)
        cout << stack->DATA[i] << '\t';
    cout << endl;
}
```

```
int main()
{
    int MAXSIZE = 5;
    struct stack *stack = createStack(MAXSIZE);

    cout << "\nPush " << 6 << ":\t";
    push(stack, 6);
    traverse(stack);

    cout << "\nPush " << -1 << ":\t";
    push(stack, -1);
    traverse(stack);

    cout << "\nPop " << pop(stack) << ":\t";
    traverse(stack);

    cout << "\nPush " << 4 << ":\t";
    push(stack, 4);
    traverse(stack);

    cout << "\nPush " << 0 << ":\t";
    push(stack, 0);
    traverse(stack);
```



```
cout << "\nPop " << pop(stack) << ":\t";  
traverse(stack);
```

```
cout << "\nPush " << 10 << ":\t";  
push(stack, 10);  
traverse(stack);
```

```
cout << "\nPush " << 8 << ":\t";  
push(stack, 8);  
traverse(stack);
```

```
cout << "\nPush " << -6 << ":\t";  
push(stack, -6);  
traverse(stack);
```

```
cout << "\nPush " << -9 << ":\t";  
push(stack, -9);  
traverse(stack);
```

```
cout << "\nPop " << pop(stack) << ":\t";  
traverse(stack);
```

```
cout << "\nPop " << pop(stack) << ":\t";  
traverse(stack);
```

```

cout << "\nPop " << pop(stack) << ":\t";

traverse(stack);

return 0;
}

```

Output:

```

[Running] cd "e:\\" && g++ Untitled-1.cpp -o Untitled-1 && "e:\\"Untitled-1

Push 6:  6

Push -1:  6  -1

Pop -1:  6

Push 4:  6  4

Push 0:  6  4  0

Pop 0:  6  4

Push 10:  6  4  10

Push 8:  6  4  10  8

Push -6:  6  4  10  8  -6

Push -9:  Stack Overflow!
6  4  10  8  -6

Pop -6:  6  4  10  8

Pop 8:  6  4  10

Pop 10:  6  4

[Done] exited with code=0 in 1.21 seconds

```

QUESTION # 02:

What should be the time complexity of algorithms (discuss in this session) in your opinion? Discuss.

Time Complexity of Push and Pop Operations on Stack:

Since both of the push and pop operations involve simple array operations and conditionals, and do not involve any use of looping structures or recursion, we conclude that these algorithms will take constant time to execute, irrespective of the input. Therefore, the time complexity of push and pop operations is $O(1)$.

QUESTION # 03:

Write algorithm to reverse a string using stack data structure also gives implementation.

Algorithm for String Reversal Using Stack:

- 1. Create an empty stack.**
- 2. Scan the string from left to right and push all its characters into the stack.**
- 3. Replace the respective characters of the string by repeatedly popping the stack.**

Implementation (C++):

```
#include <iostream>
```

```
using namespace std;
```

```
struct stack
```

```
{
```

```
    int MAXSIZE;
```

```
    int TOP;
```

```
    char *DATA;
```

```
};
```

```
struct stack *createStack(int MAXSIZE)
```

```
{
```

```
    struct stack *newStack = (struct stack *)malloc(sizeof(struct stack));
```

```
    newStack->TOP = -1;
```

```
    newStack->MAXSIZE = MAXSIZE;
```

```
newStack->DATA = (char *)malloc(sizeof(char) * MAXSIZE);  
  
return newStack;  
  
}
```

```
void push(struct stack *stack, char ITEM)  
  
{  
  
    if (stack->TOP == stack->MAXSIZE - 1)  
  
        cout << "Stack Overflow!" << endl;  
  
    else  
  
        stack->DATA[++stack->TOP] = ITEM;  
  
}
```

```
char pop(struct stack *stack)  
  
{  
  
    if (stack->TOP == -1)  
  
        cout << "Stack Underflow!" << endl;  
  
    else  
  
    {  
  
        char ITEM = stack->DATA[stack->TOP--];  
  
        return ITEM;  
  
    }  
  
}
```

```

void reverseString(char string[])
{
    int MAXSIZE;

    for (MAXSIZE = 0; string[MAXSIZE] != 0; MAXSIZE++);

    struct stack *stack = createStack(MAXSIZE);

    for (int i = 0; i < MAXSIZE; i++)
        push(stack, string[i]);

    for (int i = 0; i < MAXSIZE; i++)
        string[i] = pop(stack);
}

int main()
{
    char string[] = "daniyal";

    cout << "Original String:\t" << string << endl;

    reverseString(string);

    cout << "Reversed String:\t" << string << endl;

    return 0;
}

```

Output:

```
[Running] cd "e:\\" && g++ Untitled-1.cpp -o Untitled-1 && "e:\\"Untitled-1
Original String: daniyal
Reversed String:   layinad

[Done] exited with code=0 in 1.142 seconds
```

QUESTION # 04:

Write an algorithm to implement two stacks in a single array.

Source Code (C++):

```
# include <iostream>

using namespace std;

# define SIZE 10

int ARRAY[SIZE];

int TOP1 = -1, TOP2 = SIZE;

void pushInStack1(int ITEM)
{
    if (TOP1 < TOP2)
    {
        ARRAY[++TOP1] = ITEM;

        cout << "Pushed " << ITEM << " in stack 1" << endl;
    }

    else
```

```
        cout << "Cannot push " << ITEM << " in stack 1 (Stack Overflow)" << endl;
    }
```

```
void pushInStack2(int ITEM)
```

```
{
    if (TOP1 < TOP2 - 1)
    {
        ARRAY[--TOP2] = ITEM;
        cout << "Pushed " << ITEM << " in stack 2" << endl;
    }
    else
        cout << "Cannot push " << ITEM << " in stack 2 (Stack Overflow)" << endl;
}
```

```
void popStack1()
```

```
{
    if (TOP1 > -1)
        TOP1--;
    else
        cout << "Cannot pop from stack 1 (Stack Underflow)" << endl;
}
```

```
void popStack2()
```

```
{  
    if (TOP1 < SIZE)  
        TOP2++;  
    else  
        cout << "Cannot pop from stack 2 (Stack Underflow)" << endl;  
}
```

```
void traverse()
```

```
{  
    cout << "Stack 1:\t";  
    for (int i = 0; i <= TOP1; i++)  
        cout << ARRAY[i] << " ";  
    cout << "\nStack 2:\t";  
    for (int i = SIZE - 1; i > TOP2; i--)  
        cout << ARRAY[i] << " ";  
    cout << endl;  
}
```

```
int main()
```

```
{  
    pushInStack1(3);  
    pushInStack1(8);  
    pushInStack1(0);
```



```
pushInStack1(1);  
pushInStack1(-5);  
pushInStack2(2);  
pushInStack2(5);  
popStack1();  
popStack2();  
pushInStack2(2);  
traverse();  
return 0;  
}
```

Output:

```
[Running] cd "e:\\" && g++ Untitled-1.cpp -o Untitled-1 && "e:\\"Untitled-1  
Pushed 3 in stack 1  
Pushed 8 in stack 1  
Pushed 0 in stack 1  
Pushed 1 in stack 1  
Pushed -5 in stack 1  
Pushed 2 in stack 2  
Pushed 5 in stack 2  
Pushed 2 in stack 2  
Stack 1:    3, 8, 0, 1,  
Stack 2:    2,  
  
[Done] exited with code=0 in 0.79 seconds
```

Lab 06 Warda Haider SE-19016

Expression Evaluation through Stack Data Structure

i. Infix ii. Postfix iii. Prefix

Q1. Solve the following postfix expressions via algorithm

P: 5, 6, 2, +, *, 12, 4, /, -	P: 2, 3, ^, 5, 2, 2, ^, *, 12, 6, /, -, +																																																		
<table border="1"> <thead> <tr> <th>Symbol Scan</th><th>Stack</th></tr> </thead> <tbody> <tr><td>5</td><td>5</td></tr> <tr><td>6</td><td>5,6</td></tr> <tr><td>2</td><td>5,6,2</td></tr> <tr><td>+</td><td>5,8</td></tr> <tr><td>*</td><td>40</td></tr> <tr><td>12</td><td>40,12</td></tr> <tr><td>4</td><td>40,12,4</td></tr> <tr><td>/</td><td>40,3</td></tr> <tr><td>-</td><td>37</td></tr> <tr><td>)</td><td></td></tr> </tbody> </table>	Symbol Scan	Stack	5	5	6	5,6	2	5,6,2	+	5,8	*	40	12	40,12	4	40,12,4	/	40,3	-	37)		<table border="1"> <thead> <tr> <th>Symbol Scan</th><th>Stack</th></tr> </thead> <tbody> <tr><td>2</td><td>2</td></tr> <tr><td>3</td><td>2,3</td></tr> <tr><td>^</td><td>8</td></tr> <tr><td>5</td><td>8,5</td></tr> <tr><td>2</td><td>8,5,2</td></tr> <tr><td>2</td><td>8,5,2,2</td></tr> <tr><td>^</td><td>8,5,4</td></tr> <tr><td>*</td><td>8,20</td></tr> <tr><td>12</td><td>8,20,12</td></tr> <tr><td>6</td><td>8,20,12,6</td></tr> <tr><td>/</td><td>8,20,2</td></tr> <tr><td>-</td><td>8,18</td></tr> <tr><td>)</td><td>26</td></tr> </tbody> </table>	Symbol Scan	Stack	2	2	3	2,3	^	8	5	8,5	2	8,5,2	2	8,5,2,2	^	8,5,4	*	8,20	12	8,20,12	6	8,20,12,6	/	8,20,2	-	8,18)	26
Symbol Scan	Stack																																																		
5	5																																																		
6	5,6																																																		
2	5,6,2																																																		
+	5,8																																																		
*	40																																																		
12	40,12																																																		
4	40,12,4																																																		
/	40,3																																																		
-	37																																																		
)																																																			
Symbol Scan	Stack																																																		
2	2																																																		
3	2,3																																																		
^	8																																																		
5	8,5																																																		
2	8,5,2																																																		
2	8,5,2,2																																																		
^	8,5,4																																																		
*	8,20																																																		
12	8,20,12																																																		
6	8,20,12,6																																																		
/	8,20,2																																																		
-	8,18																																																		
)	26																																																		

ALGORITHM

1. Add right parenthesis “)” at end it will act as sentinel
2. Scan P from left to right and repeat 3 & 4 for each element of P until the sentinel “)” is encountered.
3. If an operand is encountered, put it on Stack
4. If an operator X is encountered, Then :
 - i. Remove the two top elements of STACK, where A is the top element and B is the next to top

element

ii Evaluate BXA

iii. Place the result of (b) back on STACK

[End of if structure]

[End of step2 loop]

5. Set value equal to the top element on STACK

6. Exit

Q2. Implement the algorithm to convert expression to equivalent postfix form.

Q: $(A + (B * C - (D / E * F) * G) * H)$ infix

CODE:

```
#include<iostream>
```

```
#include<stack>
```

```
using namespace std;
```

```
bool isOperator(char c)
```

```
{
    if(c=='+'||c=='-'||c=='*'||c=='/'||c=='^')
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

```
int precedence(char c)
```

```
{
```

```

        if(c == '^')
            return 3;
        else if(c == '*' || c == '/')
            return 2;
        else if(c == '+' || c == '-')
            return 1;
        else
            return -1;
    }

```

```

string InfixToPostfix(stack<char> s, string infix)
{
    string postfix;
    for(int i=0;i<infix.length();i++)
    {
        if((infix[i] >= 'a' && infix[i] <= 'z') || (infix[i] >= 'A' && infix[i] <= 'Z'))
        {
            postfix+=infix[i];
        }
        else if(infix[i] == '(')
        {
            s.push(infix[i]);
        }
        else if(infix[i] == ')')
        {
            while((s.top()!='(') && (!s.empty()))
            {
                char temp=s.top();
                postfix+=temp;
                s.pop();
            }
            if(s.top()=='(')
            {

```

```

        s.pop();
    }
}
else if(isOperator(infix[i]))
{
    if(s.empty())
    {
        s.push(infix[i]);
    }

else
{
    if(precedence(infix[i])>precedence(s.top()))
    {
        s.push(infix[i]);
    }
else if((precedence(infix[i])==precedence(s.top()))&&(infix[i]!='^'))
{
s.push(infix[i]);
}
else
{
while((!s.empty())&&( precedence(infix[i])<=precedence(s.top()))
{
postfix+=s.top();
s.pop();
}
s.push(infix[i]);
}
}
}
}

```

```

while(!s.empty())
{
    postfix+=s.top();
    s.pop();
}

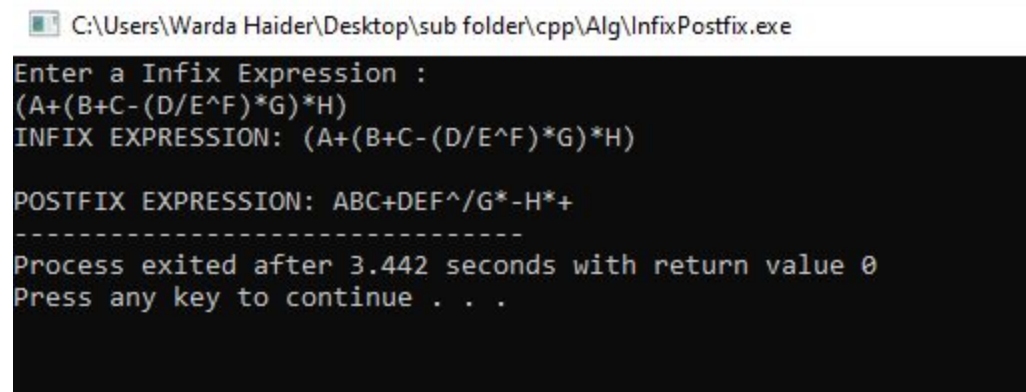
return postfix;
}

int main()
{
    string infix_exp, postfix_exp;
    cout<<"Enter a Infix Expression : "<<endl;
    cin>>infix_exp;
    stack <char> stack;
    cout<<"INFIX EXPRESSION: "<<infix_exp<<endl;
    postfix_exp = InfixToPostfix(stack, infix_exp);
    cout<<endl<<"POSTFIX EXPRESSION: "<<postfix_exp;

    return 0;
}

```

OUTPUT



```

C:\Users\Warda Haider\Desktop\sub folder\cpp\Alg\InfixPostfix.exe
Enter a Infix Expression :
(A+(B+C-(D/E^F)*G)*H)
INFIX EXPRESSION: (A+(B+C-(D/E^F)*G)*H)

POSTFIX EXPRESSION: ABC+DEF^/G*-H*+
-----
Process exited after 3.442 seconds with return value 0
Press any key to continue . . .

```

EXPLANATION

Input String	Expression	Stack
A+(B+C-(D/E^F)*G)*H	A	
A+(B+C-(D/E^F)*G)*H	A	+
A+(B+C-(D/E^F)*G)*H	A	+(
A+(B+C-(D/E^F)*G)*H	AB	+(
A+(B+C-(D/E^F)*G)*H	AB	+(+
A+(B+C-(D/E^F)*G)*H	ABC	+(+
A+(B+C-(D/E^F)*G)*H	ABC+	+(-
A+(B+C-(D/E^F)*G)*H	ABC+	+(-(
A+(B+C-(D/E^F)*G)*H	ABC+D	+(-(

$A+(B+C-(D/E^F)^*G)^*H$	ABC+D	$+(-(/$
$A+(B+C-(D/E^F)^*G)^*H$	ABC+DE	$+(-(/$
$A+(B+C-(D/E^F)^*G)^*H$	ABC+DE	$+(-(/^$
$A+(B+C-(D/E^F)^*G)^*H$	ABC+DEF	$+(-(/^$
$A+(B+C-(D/E^F)^*G)^*H$	ABC+DEF^/	$+(-$
$A+(B+C-(D/E^F)^*G)^*H$	ABC+DEF^/	$+(-^*$
$A+(B+C-(D/E^F)^*G)^*H$	ABC+DEF^/G	$+(-^*$
$A+(B+C-(D/E^F)^*G)^*H$	ABC+DEF^/G*-	+
$A+(B+C-(D/E^F)^*G)^*H$	ABC+DEF^/G*-	+*
$A+(B+C-(D/E^F)^*G)^*H$	ABC+DEF^/G*- H	+*
$A+(B+C-(D/E^F)^*G)^*H$	ABC+DEF^/G*- H*+	

Q3. Write an algorithm to solve prefix expressions and also give implementation.

ALGORITHM

Reverse the given expression and Iterate through it, one character at a time

1. If the character is an operand, push it to the stack.
2. If the character is an operator,
 - a. pop the operand from the stack, say it's s1.
 - b. pop another operand from the stack, say it's s2.
 - c. perform (**s1 operator s2**) and push it to stack.
3. Once the expression iteration is completed, The stack will have the final result. pop from the stack and return the result.

Explanation

Expression: +9*26

Reverse: 62*9+

6	6	6 is an operand,
		push to Stack

2	6 2	2 is an operand,
		push to Stack

*	12 (6*2)	* is an operator,
		pop 6 and 2, multiply
		them and push result

to Stack

9 12 9 9 is an operand, push

to Stack

+ 21 (12+9) + is an operator, pop
12 and 9 add them and
push result to Stack

Result: 21

CODE

```
#include <bits/stdc++.h>
using namespace std;
bool isOperand(char c)
{
    // If the character is a digit then it must
    // be an operand
    return isdigit(c);
}

double evaluatePrefix(string exprsn)
{
    stack<double> Stack;

    for (int j = exprsn.size() - 1; j >= 0; j--) {

        // Push operand to Stack
        // To convert exprsn[j] to digit subtract
```

```

// '0' from exprsn[j].
if (isOperand(exprsn[j]))
    Stack.push(exprsn[j] - '0');

else {

    // Operator encountered
    // Pop two elements from Stack
    double o1 = Stack.top();
    Stack.pop();
    double o2 = Stack.top();
    Stack.pop();

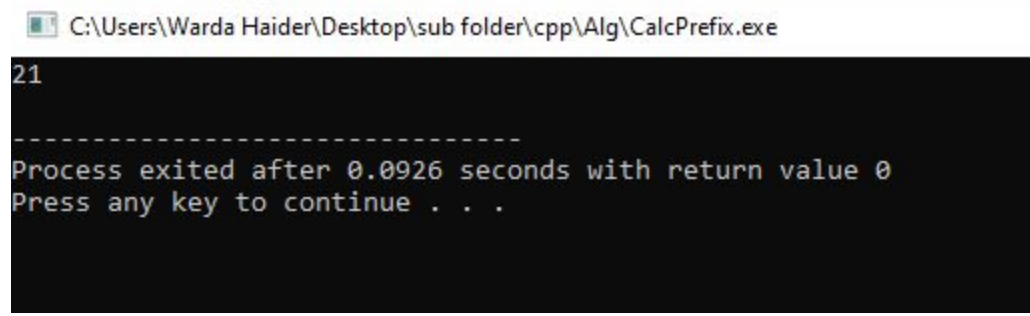
    // Use switch case to operate on o1
    // and o2 and perform o1 O o2.
    switch (exprsn[j]) {
    case '+':
        Stack.push(o1 + o2);
        break;
    case '-':
        Stack.push(o1 - o2);
        break;
    case '*':
        Stack.push(o1 * o2);
        break;
    case '/':
        Stack.push(o1 / o2);
        break;
    }
}

return Stack.top();
}

```

```
// Driver code
int main()
{
    string exprsn = "+9*26";
    cout << evaluatePrefix(exprsn) << endl;
    return 0;
}
```

OUTPUT



```
C:\Users\Warda Haider\Desktop\sub folder\cpp\Alg\CalcPrefix.exe
21
-----
Process exited after 0.0926 seconds with return value 0
Press any key to continue . . .
```

Lab 07 Muhammad Daniyal Aamir SE-049

Queue data structure & Operations

i. Enqueue ii. Dequeue

QUESTION # 01:

Write Algorithms for enqueue and dequeue operations in a circular queue.

Source Code (C++):

```
# include <iostream>
```

```
using namespace std;
```

```
# define MAXSIZE 10
```

```
int DATA[MAXSIZE];
```

```
int REAR = -1, FRONT = 0;
```

```
int COUNT = 0;
```

```
void enqueue(int ITEM)
```

```
{
```

```

if (COUNT != MAXSIZE)

{
    DATA[++REAR] = ITEM;

    COUNT++;

    if (REAR == MAXSIZE - 1)

        REAR = -1;

    cout << "Enqueued " << ITEM << " in queue\tFRONT = " << FRONT << ",\tREAR = " <<
REAR << "\t" << endl;

}

else

    cout << "Cannot enqueue " << ITEM << " in queue (Overflow)" << endl;

}

void dequeue()

{
    if (COUNT != 0)

    {
        int ITEM = DATA[FRONT++];

        COUNT--;

        if (FRONT == MAXSIZE)

            FRONT = 0;
    }
}

```

```
    cout << "Dequeued " << ITEM << " from queue\tFRONT = " << FRONT << ",\tREAR = "
<< REAR << "\t" << endl;
```

```
}
```

```
else
```

```
{
```

```
    cout << "Cannot dequeue from queue (Underflow)" << endl;
```

```
}
```

```
}
```

```
void traverse()
```

```
{
```

```
    cout << "Queue:\t(R) ";
```

```
    for (int i = REAR; i != FRONT - 1; i = (i - 1) % MAXSIZE)
```

```
        cout << DATA[i] << " ";
```

```
    cout << "(F)" << endl;
```

```
}
```

```
int main()
```

```
{
```

```
    enqueue(5);
```

```
    enqueue(3);
```

```
    enqueue(7);
```

```
    enqueue(1);
```

```
    enqueue(0);
```

```

    dequeue();

    dequeue();

    traverse();

    return 0;

}

```

Output:

```

[Running] cd "e:\\" && g++ Untitled-1.cpp -o Untitled-1 && "e:\\"Untitled-1
Enqueued 5 in queue FRONT = 0, REAR = 0
Enqueued 3 in queue FRONT = 0, REAR = 1
Enqueued 7 in queue FRONT = 0, REAR = 2
Enqueued 1 in queue FRONT = 0, REAR = 3
Enqueued 0 in queue FRONT = 0, REAR = 4
Dequeued 5 from queue FRONT = 1, REAR = 4
Dequeued 3 from queue FRONT = 2, REAR = 4
Queue: (R) 0 1 7 (F)

[Done] exited with code=0 in 0.794 seconds

```

QUESTION # 02:

Give implementation of queue data structure using two stacks (let S1 & S2 be the two stacks for push and pop operations respectively).

Source Code (C++):

```

#include <iostream>

using namespace std;

struct stack
{
    int MAXSIZE;

```



```

    int TOP;

    int *DATA;
};

struct stack* createStack(int MAXSIZE)
{
    struct stack *newStack = (struct stack *)malloc(sizeof(struct stack));

    newStack->TOP = -1;

    newStack->MAXSIZE = MAXSIZE;

    newStack->DATA = (int *)malloc(sizeof(int) * MAXSIZE);

    return newStack;
}

void push(struct stack *stack, int ITEM)
{
    if (stack->TOP == stack->MAXSIZE - 1)

        cout << "Stack Overflow!" << endl;

    else

        stack->DATA[++stack->TOP] = ITEM;
}

int pop(struct stack *stack)
{

```

```
if (stack->TOP == -1)

    cout << "Stack Underflow!" << endl;

else

{

    int ITEM = stack->DATA[stack->TOP--];

    return ITEM;

}

}
```

struct queue

```
{

    struct stack *s1;

    struct stack *s2;

};
```

struct queue *createQueue(int MAXSIZE)

```
{

    struct queue *q;

    q->s1 = createStack(MAXSIZE);

    q->s2 = createStack(MAXSIZE);

    return q;

}
```

```

void enqueue(struct queue *q, int ITEM)
{
    if (q->s1->TOP < q->s1->MAXSIZE - 1)
    {
        push(q->s1, ITEM);
        cout << "Enqueued " << ITEM << " in queue" << endl;
    }
    else
        cout << "Queue overflow" << endl;
}

void dequeue(struct queue *q)
{
    if (q->s1->TOP > -1)
    {
        while (q->s1->TOP != -1)
            push(q->s2, pop(q->s1));
        int ITEM = pop(q->s2);
        while (q->s2->TOP != -1)
            push(q->s1, pop(q->s2));
        cout << "Dequeued " << ITEM << " from queue" << endl;
    }
    else

```

```
        cout << "Queue Underflow" << endl;
    }
```

```
void traverse(struct queue *q)
{
    for (int i = 0; i <= q->s1->TOP; i++)
        cout << q->s1->DATA[i] << "t";
    cout << endl;
}
```

```
int main()
{
    struct queue *q = createQueue(5);
    enqueue(q, 5);
    enqueue(q, 3);
    enqueue(q, 7);
    enqueue(q, 1);
    enqueue(q, 0);
    dequeue(q);
    dequeue(q);
    traverse(q);
    return 0;
}
```

Output:

```
[Running] cd "e:\\" && g++ tempCodeRunnerFile.cpp -o tempCodeRunnerFile && "e:\\"tempCodeRunnerFile
Enqueued 5 in queue
Enqueued 3 in queue
Enqueued 7 in queue
Enqueued 1 in queue
Enqueued 0 in queue
Dequeued 5 from queue
Dequeued 3 from queue
7    1    0
[Done] exited with code=0 in 0.846 seconds
```

Lab 08

Recursive Algorithms (Recursion)

Maria Nasir SE-007

1. Calculate the Time complexity of Fibonacci and factorial recursive algorithms; also give the upper bound of both algorithms.

TIME COMPLEXITY OF FACTORIAL ALGORITHM:

$$T(n) = T(n-1) + 1 \quad // \text{if } n > 0$$

$$T(0) = 1 \quad // \text{if } n = 0$$

$$T(n) = T(n-1) + 3$$

$$T(n) = (T(n-2) + 3) + 3 = T(n-2) + 6$$

$$T(n) = ((T(n-3) + 3) + 3) + 3 = T(n-3) + 9$$

Generally

$$T(n) = T(n-k) + 3K$$

$$\text{For } k=n \Rightarrow n-k=0$$

$$= T(0) + 3n$$

$$T(n) = 1 + 3n$$

Worst time complexity: $O(n)$

TIME COMPLEXITY OF FIBONACCI ALGORITHM:

$$T(n) = T(n-1) + T(n-2) + 4; \text{ if } n > 1$$

$$T(0) = 1; \text{ if } n \leq 1$$

Since $T(n-1)$ takes more time so we assume here $T(n-2)$ almost takes same time

So we rewrite above equation 1 as:

$$T(n) = T(n-1) + T(n-1) + 4$$

$$T(n) = 2T(n-1) + 4 \text{ or}$$

$$T(n) = 2T(n-1) + C$$

$$T(n) = 2T(n-1) + C$$

$$T(n) = 2(2(T(n-2) + C) + C) = 4T(n-2) + 3C$$

$$T(n) = 4T(n-2) + 3C$$

$$T(n) = 4(2T(n-3) + C) + 3C$$

$$T(n) = 8T(n-3) + 7C$$

Similarly:

$$T(n) = 8(2T(n-4) + C) + 7C$$

$$T(n) = 16T(n-4) + 15C$$

$$T(n) = 16T(n-4) + 15C$$

Generally,

$$T(n) = 2^k T(n-k) + (2^k - 1)C$$

At $k=n$ we have $n-k=0$

$$T(n) = 2^n T(0) + (2^n - 1)C$$

$$T(n) = 2^n (1) + (2^n - 1)C$$

$$T(n) = 2^n + 2^n C - C \Rightarrow 2^n (1+C) - C$$

Worst time complexity: $O(n) = 2^n$

2. Write iterative factorial and Fibonacci algorithms, also analyze the time complexity.

ALGORITHM FOR CALCULATING FACTORIAL:

Factorial (FACT, N)

1. If $N=0$, then set $FACT=1$ and return
2. Set $FACT=1$
3. Repeat for $k=1$ to N
 - 3a. Set $FACT=k*FACT$
4. Return

TIME COMPLEXITY OF FACTORIAL ALGORITHM:

Statement	Operation	Iteration	Sub-total
1	2	1	1

2	1	1	1
3	1	n-1	n-1
3a	2	n	n

$$f(n) = 3n+4$$

$$f(n) = 3n+c$$

Upper bond = O (n)

ALGORITHM FOR CALCULATING FIBONACCI:

Step 1: Start

Step 2: Declare variable a, b, c, n, i

Step 3: Initialize variable a=1, b=1, i=3

Step 4: Read n from user

Step 5: Print a and b

Step 6: Repeat until $i \leq n$

- a) $c=a + b$
- b) print c
- c) $a=b, b=c$
- d) $i=i+1$

Step 7: Stop

TIME COMPLEXITY OF FIBONACCI ALGORITHM:

Statement	Operation	Iteration	Sub-total
-----------	-----------	-----------	-----------

3	3	1	1
4	1	1	1
5	1	1	1
6	1	n-1	n-1
6a	2	n-2	n-2
6b	1	n-2	n-2
6c	2	n-2	n-2
6d	2	n-2	n-2

$$f(n) = 8n-10$$

Upper bound = $O(n)$

3. Write a recursive algorithm for Tower of Hanoi Puzzle and calculate its upper bound (also give implementation)?

ALGORITHM:

1. Hanoi(disk, source, destination, aux)
2. Start
3. If (disk == 1) then

Move disk from source to destination.
4. Else

[Move n-1 disk from peg source to aux]

- a. Call Hanoi (disk-1, source, destination, aux)
- b. Move disk from source to destination.
[Move n-1 disk from peg aux to destination]
- c. Call Hanoi (disk-1, aux, source, destination)

5. Stop

TIME COMPLEXITY:

$$T(n) = 2T(n-1) + 1 \dots \dots \dots (1)$$

Solving it by back substitution:

$$T(n-1) = 2T(n-2) + 1 \dots \dots \dots (2)$$

$$T(n-2) = 2T(n-3) + 1 \dots \dots \dots (3)$$

Substitute equation (2) in (1)

$$T(n) = 4T(n-2) + 3 \dots \dots \dots (4)$$

Substitute equation (3) in (4)

$$T(n) = 8T(n-3) + 7 \dots \dots \dots (4)$$

After Generalization:

$$T(n) = 2^k T(n-k) + 2^k - 1$$

Base condition $T(1) = 1$

$$n - k = 1$$

put, $k = n-1$

$$T(n) = 2^{(n-1)} T(n-(n-1)) + 2^{(n-1)} - 1$$

$$T(n) = 2^n - 1$$

$$O(n) = O(2^n)$$

IMPLEMENTATION:

```
#include<iostream>

using namespace std;

//tower of HANOI function implementation

void TOH(int n, char Sour, char Aux, char Des)

{
    if(n==1)
    {
        cout<<"Move Disk "<<n<<" from "<<Sour<<" to "<<Des<<endl;
        return;
    }

    TOH (n-1, Sour, Des, Aux);

    cout<<"Move Disk "<<n<<" from "<<Sour<<" to "<<Des<<endl;

    TOH(n-1,Aux,Sour,Des);
}


//main program

int main()

{
    int n;

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

    cin>>n;
```

```
//calling the TOH  
TOH(n,'A','B','C');  
  
return 0;  
  
}
```

OUTPUT:

```
Enter no. of disks:3  
Move Disk 1 from A to C  
Move Disk 2 from A to B  
Move Disk 1 from C to B  
Move Disk 3 from A to C  
Move Disk 1 from B to A  
Move Disk 2 from B to C  
Move Disk 1 from A to C
```

Lab 09 Abdul Hannan SE-041

Tree data structure & Operations

i. Insertion ii. Deletion

QUESTION # 01 & 02:

1. Construct a binary tree (initially empty). Insert nodes 15,10,20,25,8,12 with the help of following definition of Node in BST:

```
struct BSTNode{  
    int data;  
    BSTNode* left;
```

```
    BSTNode* right;  
} BSTNode* rootPtr;
```

2.Delete one left-child-node and one right-child- node from above tree. Analyze the change occurring in the tree after deletion.

Source Code (C++):

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

```
struct BSTNode {  
    int data;  
    BSTNode *left;  
    BSTNode *right;  
};
```

```
BSTNode *getNewNode(int data)  
{  
    BSTNode *newNode = new BSTNode;  
    newNode->data = data;  
    newNode->left = newNode->right = NULL;  
    return newNode;  
}
```

```
BSTNode *insert(BSTNode *root, int data)  
{
```

```
if (root == NULL)
    root = getNewNode(data);
else if (data <= root->data)
    root->left = insert(root->left, data);
else
    root->right = insert(root->right, data);
return root;
}
```

```
bool search(BSTNode *root, int data)
{
    if (root == NULL)
        return false;
    else if (root->data == data)
        return true;
    else if (data <= root->data)
        return search(root->left, data);
    else
        return search(root->right, data);
}
```

```
void traverseInorder(BSTNode *root)
{
    if (root != NULL)
    {
```

```
        traverseInorder(root->left);  
        cout << root->data << '\t';  
        traverseInorder(root->right);  
    }  
}
```

```
void traversePreorder(BSTNode *root)  
{  
    if (root != NULL)  
    {  
        cout << root->data << '\t';  
        traverseInorder(root->left);  
        traverseInorder(root->right);  
    }  
}
```

```
void traversePostorder(BSTNode *root)  
{  
    if (root != NULL)  
    {  
        traverseInorder(root->left);  
        traverseInorder(root->right);  
        cout << root->data << '\t';  
    }  
}
```

```

BSTNode *getLeftMostChildNode(BSTNode *root)
{
    BSTNode *currentNode = root;
    while (currentNode != NULL && currentNode->left != NULL)
        currentNode = currentNode->left;
    return currentNode;
}

```

```

BSTNode *deleteNode(BSTNode *root, int data)
{
    if (root == NULL)
        return root;

    if (data < root->data)
        root->left = deleteNode(root->left, data);
    else if (data > root->data)
        root->right = deleteNode(root->right, data);
    else
    {
        if (root->left == NULL)
        {
            BSTNode *temp = root->right;
            free(root);
            return temp;
        }
    }
}

```



```

else if (root->right == NULL)
{
    BSTNode *temp = root->left;

    free(root);

    return temp;
}
else
{
    BSTNode *temp = getLeftMostChildNode(root->right);

    root->data = temp->data;

    root->right = deleteNode(root->right, temp->data);
}
}

return root;
}

int main()
{
    BSTNode *root = NULL;

    root = insert(root, 15);
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 25);
    root = insert(root, 8);
    root = insert(root, 12);

```

```

cout << "1) INSERTION:\nInsert 15, 10, 20, 25, 8, 12:" << endl;

traverseInorder(root);

root = deleteNode(root, 8);

cout << "\n2) DELETION:\ni) Delete left child node (8):" << endl;

traverseInorder(root);

root = deleteNode(root, 25);

cout << "\nii) Deleted right child node (25):" << endl;

traverseInorder(root);

return 0;
}

```

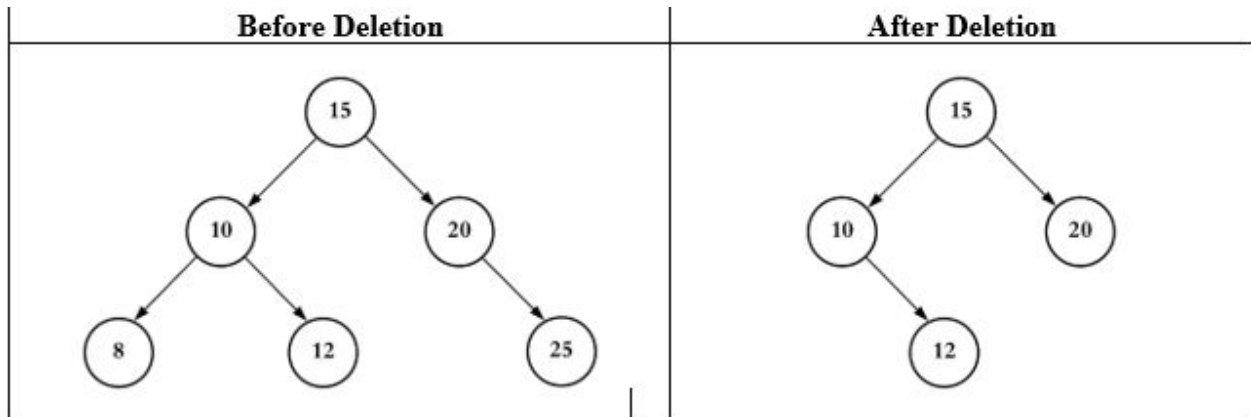
Output:

```

[Running] cd "e:\\" && g++ Untitled-1.cpp -o Untitled-1 && "e:\\"Untitled-1
1) INSERTION:
Insert 15, 10, 20, 25, 8, 12:
8  10  12  15  20  25
2) DELETION:
i) Delete left child node (8):
10  12  15  20  25
ii) Deleted right child node (25):
10  12  15  20
[Done] exited with code=0 in 2.526 seconds

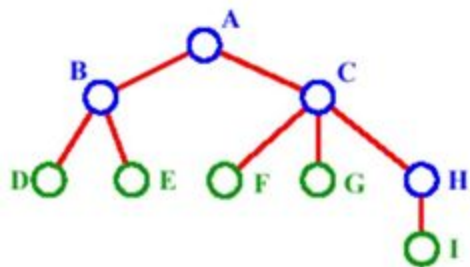
```

Pictorial Representation:



QUESTION # 03:

From the figure below, identify i. root ii. height of tree iii. Degree iv. Size v. leaf node(s)



1.	Root	A
2.	Height of Tree	3
3.	Degree of	D, E, F, G, I
		H
		A, B

		C	3
4.	Size	9	
5.	Leaf nodes	D, E, F, G, I	

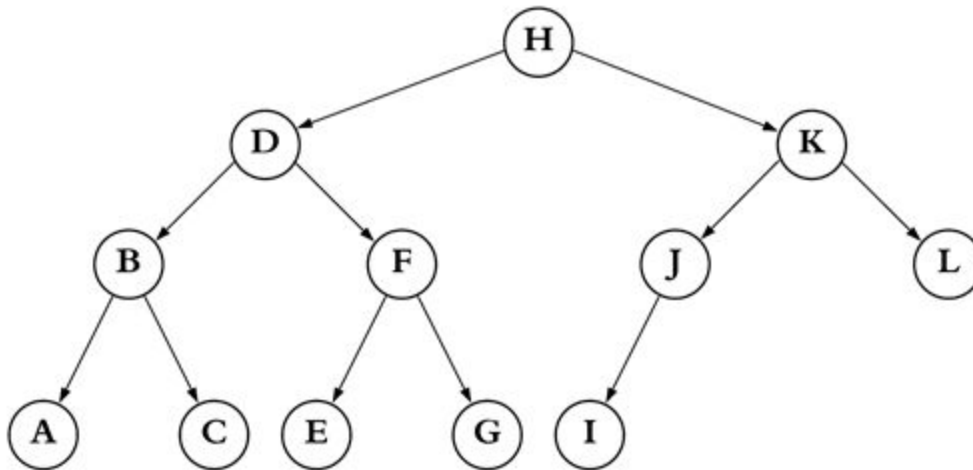
QUESTION # 04:

From the figure below, identify i. root ii. Height of tree iii. Degree iv. Size v. leaf node(s)

Draw a tree from the following representation: and identify i. Child of H ii. Height of Tree
iii. Leaf node(s) iv. Parent of A

$D \rightarrow F, D \rightarrow B, K \rightarrow J, K \rightarrow L, B \rightarrow A, B \rightarrow C, H \rightarrow D, H \rightarrow K, F \rightarrow E, F \rightarrow G, J \rightarrow I$

Pictorial Representation:



1.	Childs of H	D, K
2.	Height of Tree	3
3.	Leaf Node(s)	A, C, E, G, I

4.	Parent of A	B
----	-------------	---

QUESTION # 05:

Write down the applications of Tree data structure.

Applications of Tree:

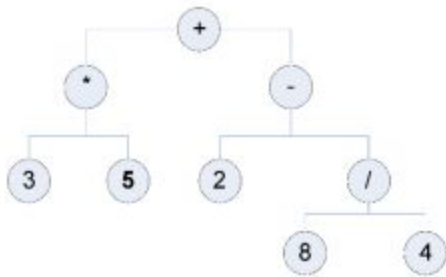
- 1.Tree data structure can be used when you want to store data naturally in a hierarchical manner. For example, a computer's file system.
- 2.If we organize data in the form of a tree, with some ordering (e.g. BST), it takes moderate time to search an item (which is relatively less than other data structures).
- 3.Insertion and deletion operations take moderate time.
- 4.The operations do not depend on the number of nodes due to pointer implementation.
- 5.Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
- 6.B-Tree and B+ Tree: They are used to implement indexing in databases.
- 7.Syntax Tree: Used in Compilers.
- 8.K-D Tree: A space partitioning tree used to organize points in K dimensional space.
- 9.Trie: Used to implement dictionaries with prefix lookup.
- 10.Suffix Tree: For quick pattern searching in a fixed text.

LAB 10 Mahnoor SE-013

Tree Traversal Algorithms

i. Inorder ii. Preorder iii. Postorder

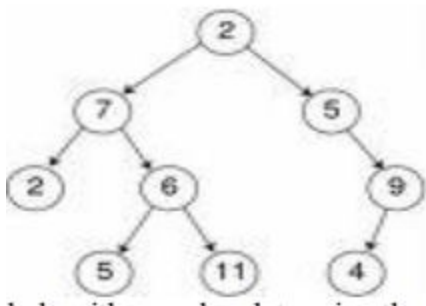
- 1. Traverse the following binary trees using the pre, in, and post order traversal methods.**



PRE ORDER TRAVERSAL: + * 3 5 - 2 / 8 4

POST ORDER TRAVERSAL: 3 5 * 2 8 4 / - +

IN ORDER TRAVERSAL: 3 * 5 + 2 - 8 / 4



PRE ORDER TRAVERSAL: 2 7 2 6 5 11 5 9 4

POST ORDER TRAVERSAL: 2 5 11 6 7 4 9 5 2

IN ORDER TRAVERSAL: 2 7 5 6 11 2 5 4 9

2. Implement all three tree traversal algorithms, also determine the worst time complexity

CODE:

```
#include<iostream>
```

```
using namespace std;
```

```
struct Node {
```

```
    char data;
```

```
    struct Node *left;
```

```
    struct Node *right;
```

```
};
```

PRE ORDER TRAVERSAL:

```
//Function to visit nodes in Preorder
```

```
void Preorder(struct Node *root) {  
    // base condition for recursion  
    // if tree/sub-tree is empty, return and exit  
    if(root == NULL) return;  
    printf ("%c ",root->data); // Print data  
    Preorder(root->left); // Visit left subtree  
    Preorder(root->right); // Visit right subtree  
}
```

IN ORDER TRAVERSAL

```
//Function to visit nodes in Inorder
```

```
void Inorder(Node *root) {  
    if(root == NULL) return;  
    Inorder(root->left); //Visit left subtree  
    printf ("%c ",root->data); //Print data  
    Inorder(root->right); // Visit right subtree  
}
```

POST ORDER TRAVERSAL

```
//Function to visit nodes in Postorder
```

```
void Postorder(Node *root) {  
    if(root == NULL) return;  
    Postorder(root->left); // Visit left subtree  
    Postorder(root->right); // Visit right subtree  
    printf ("%c ",root->data); // Print data  
}
```

```
// Function to Insert Node in a Binary Search Tree
```

```

Node* Insert(Node *root,char data) {
    if(root == NULL) {
        root = new Node();
        root->data = data;
        root->left = root->right = NULL;
    }
    else if(data <= root->data)
        root->left = Insert(root->left,data);
    else
        root->right = Insert(root->right,data);
    return root;
}

```

```

int main() {
    /*Code To Test the logic creating an example tree

```

```

        M
      /\
     B  Q
    /\  \
   A  C  Z

```

```

    */

    Node* root = NULL;
    root = Insert(root,'M'); root = Insert(root,'B');
    root = Insert(root,'Q'); root = Insert(root,'Z');
    root = Insert(root,'A'); root = Insert(root,'C');
    //Print Nodes in Preorder.
    cout<<"Preorder: ";
    Preorder(root);
    cout<<"\n";

```



```

//Print Nodes in Inorder
cout<<"Inorder: ";
Inorder(root);
cout<<"\n";

//Print Nodes in Postorder
cout<<"Postorder: ";
Postorder(root);
cout<<"\n";

}

```

OUTPUT:

```

d:\project\hello\bin\Debug\hello.exe
Preorder: M B A C Q Z
Inorder: A B C M Q Z
Postorder: A C B Z Q M

Process returned 0 (0x0)   execution time : 0.026 s
Press any key to continue.

```

WORST CASE TIME COMPLEXITY:

Assuming that you use recursion,

$T(n) = 2 * T(n/2) + 1$ -----> (1) Maria ke bhee tum kro ge..?

$T(n/2)$ for left subtree and $T(n/2)$ for right subtree and '1' for verifying the base case.

$T(n) = 4 * T(n/4) + 3$ -----> (2)

$T(n) = 8 * T(n/8) + 7$ -----> (3)

..

..

..

$T(n) = 2^k * T(n/2^k) + 2^{(k-1)}$

$$n/2^k=1$$

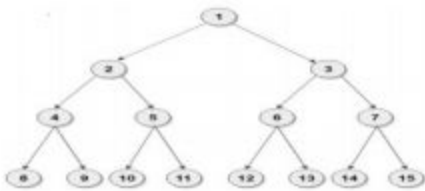
$$n=2^k \text{ and } k=\log(n)$$

$$T(n) = 2^k * T(1) + 2^{(\log(n)-1)}$$

So, the time complexity will be $O(n)$

The traversal(either inorder or preorder or post order) is of order $O(n)$.

3. Gives the implementation of Print all nodes of a perfect binary tree in Top-to-Down order.



CODE:

```

#include <iostream>
#include <queue>
using namespace std;
/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data;
    Node *left;
    Node *right;
};
/* Helper function that allocates a new node with the given data and NULL left and right
pointers. */
Node *newNode(int data)
{
    Node *node = new Node;

```

```

    node->data = data;
    node->right = node->left = NULL;
    return node;
}
/* Given a perfect binary tree, print its nodes in specific level order */
void printSpecificLevelOrder(Node *root)
{
    if (root == NULL)
        return;
    // Let us print root and next level first
    cout << root->data;
    // Since it is perfect Binary Tree, right is not checked
    if (root->left != NULL)
        cout << " " << root->left->data << " " << root->right->data;
    // Do anything more if there are nodes at next level in
    // given perfect Binary Tree
    if (root->left->left == NULL)
        return;
    // Create a queue and enqueue left and right children of root
    queue <Node *> q;
    q.push(root->left);
    q.push(root->right);
    // We process two nodes at a time, so we need two variables
    // to store two front items of queue
    Node *first = NULL, *second = NULL;
    // traversal loop
    while (!q.empty())
    {
        // Pop two items from queue
        first = q.front();

```

```

    q.pop();
    second = q.front();
    q.pop();
    // Print children of first and second in reverse order
    cout << " " << first->left->data << " " << second->right->data;
    cout << " " << first->right->data << " " << second->left->data;
    // If first and second have grandchildren, enqueue them in reverse order
    if (first->left->left != NULL)
    {
        q.push(first->left);
        q.push(second->right);
        q.push(first->right);
        q.push(second->left);
    }
}
}

```

```

int main()
{
    //Perfect Binary Tree of Height 3
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
}

```

```

root->left->right->left = newNode(10);
root->left->right->right = newNode(11);
root->right->left->left = newNode(12);
root->right->left->right = newNode(13);
root->right->right->left = newNode(14);
root->right->right->right = newNode(15);
cout << "Top down Level Order traversal of binary tree is \n";
printSpecificLevelOrder(root);
return 0;}

```

OUTPUT:

```

Top down Level Order traversal of binary tree is
1 2 3 4 7 5 6 8 15 9 14 10 13 11 12
Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.

```

Lab 11 Muhammad Daniyal Aamir SE-049

Graph data structure & Operations

i. Adjacency List ii. Adjacency Matrix

QUESTION # 01:

Give the algorithms for adjacency list and adjacency matrix methods .also determine their time complexities.

Algorithm for Adjacency Matrix:

1. Create a matrix A of size $V \times V$ and initialize it with zero.
2. Iterate over each given edge of the form (u, v) and assign 1 to $A[u][v]$. Also, If graph is undirected then assign 1 to $A[v][u]$.

Time Complexity: Adding or removing an edge takes $O(1)$ of time due to simple array operation.

Algorithm for Adjacency List:

1. Create an array A of size V and type of array must be list of vertices. Initially each list is empty so each array element is initialized with empty list.
2. Iterate each given edge of the form (u, v) and append v to the u^{th} list of array A. Also, if graph is undirected append u to the v^{th} list of array A.

Time Complexity: Adding or removing an edge takes $O(1)$ of time due to linked list operations.

QUESTION # 02:

Assume an undirected graph having 4 vertices 0, 1, 2, 3. The vertices are connected in the following manner $0 \rightarrow 1$, $0 \rightarrow 2$, $1 \rightarrow 0$, $1 \rightarrow 2$, $2 \rightarrow 3$, $3 \rightarrow 2$. Implement graph DS by using Adjacency list and Adjacency Matrix method.

Adjacency Matrix Implementation (C++):

```
#include <iostream>

using namespace std;

int main()
{
    int V, E, u, v;

    cout << "Enter no. of vertices: "; cin >> V;

    cout << "Enter no. of edges: "; cin >> E;

    int A[V][V];

    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
```

```

        A[i][j] = 0;

    }

    cout << "Created a " << V << " x " << V << " adjacency matrix" << endl;

    for (int i = 1; i <= E; i++)

    {

        cout << "\nEDGE " << i << endl;

        cout << "Start: "; cin >> u;

        cout << "End: "; cin >> v;

        A[u][v] = 1;

    }


    cout << "\nADJACENCY MATRIX:" << endl;

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

    {

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

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

        cout << "\n";

    }

    return 0;

}

```

Adjacency List Implementation (C++):

```

#include <bits/stdc++.h>

using namespace std;

int main()
{
    int V, E, u, v;

    cout << "Enter no. of vertices: "; cin >> V;

    cout << "Enter no. of edges: "; cin >> E;

    vector<vector<int>> A(V);

    cout << "Created a " << V << " vertices adjacency list" << endl;

    for (int i = 0; i < E; i++)
    {
        cout << "\nEDGE " << i << endl;

        cout << "Start: "; cin >> u;

        cout << "End: "; cin >> v;

        A[u].push_back(v);
    }

    cout << "\nADJACENCY LIST:" << endl;

    for (int i = 0; i < V; i++)
    {
        cout << i << " ----> ";
    }
}

```



```

        for (int j = 0; j < A[i].size(); j++)

            cout << A[i][j] << " -->";

        cout << "\n";

    }

    return 0;

}

```

Output:

Adjacency Matrix

```

Enter no. of vertices: 4
Enter no. of edges: 6
Created a 4 x 4 adjacency matrix

EDGE 1
Start: 0
End: 1

EDGE 2
Start: 0
End: 2

EDGE 3
Start: 1
End: 0

EDGE 4
Start: 1
End: 2

EDGE 5
Start: 2
End: 3

EDGE 6
Start: 3
End: 2

ADJACENCY MATRIX:
0 1 1 0
1 0 1 0
0 0 0 1
0 0 1 0

```

Adjacency List

```

Enter no. of vertices: 4
Enter no. of edges: 6
Created a 4 vertices adjacency list

EDGE 0
Start: 0
End: 1

EDGE 1
Start: 0
End: 2

EDGE 2
Start: 1
End: 0

EDGE 3
Start: 1
End: 2

EDGE 4
Start: 2
End: 3

EDGE 5
Start: 3
End: 2

ADJACENCY LIST:
0 ----> 1 -->2 -->
1 ----> 0 -->2 -->
2 ----> 3 -->
3 ----> 2 -->

```

QUESTION # 03: Differentiate between graph and Tree data structure.

	Tree	Graph
1.	It stores data in hierarchical manner with parent and child nodes.	It consists of a number of vertices and edges which represent connection (network manner).
2.	Has a root node	Doesn't have a root node
3.	Has no loop	Can have loops
4.	They are less complex	They are more complex

Lab 12 Warda Haider SE-19016

Graph Traversal Algorithms

i. DFS ii. BFS

1. Give Implementation of depth & breadth first search algorithms on graph (lab 11 Q2)

BREADTH FIRST SEARCH CODE:

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

// a directed graph class
class DiGraph
{
    int V; // No. of vertices
```

```

        // Pointer to an array containing adjacency lists
        list<int>*adjList;

public:
    DiGraph(int V); // Constructor

    // add an edge from vertex v to w
    void addEdge(int v, int w);

    // BFS traversal sequence starting with s -> starting node
    void BFS(int s);
};

```

```

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

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

```

```

void DiGraph::BFS(int s)
{
    // initially none of the vertices is visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // queue to hold BFS traversal sequence
    list<int>queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
}

```

```

queue.push_back(s);

// iterator 'i' to get all adjacent vertices
list<int>::iterator i;

while(!queue.empty())
{
// dequeue the vertex
s = queue.front();
cout << s << " ";
queue.pop_front();

// get all adjacent vertices of popped vertex and process each if not already visited
for (i = adjList[s].begin(); i != adjList[s].end(); ++i)
{
if (!visited[*i])
{
visited[*i] = true;
queue.push_back(*i);
}
}
}
}

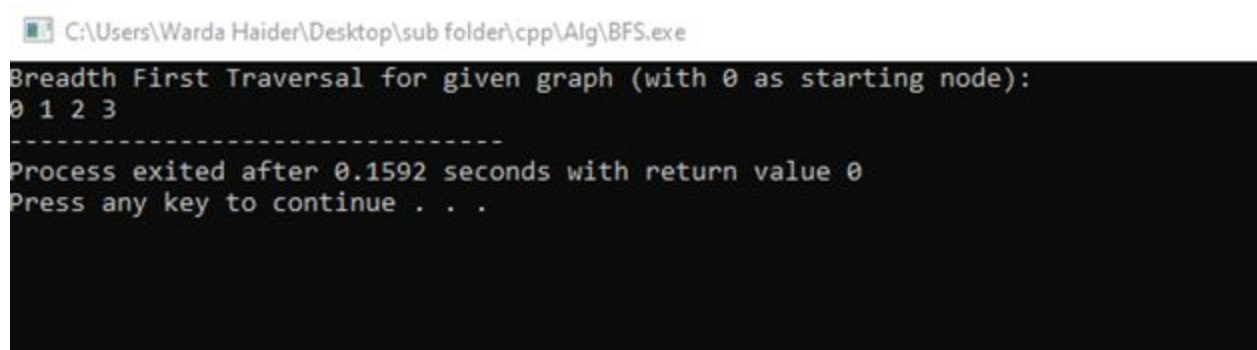
int main()
{
// create a graph
DiGraph dg(4);
dg.addEdge(0, 1);
dg.addEdge(0, 2);
dg.addEdge(1, 0);
dg.addEdge(1, 2);
dg.addEdge(2, 3);
dg.addEdge(3, 2);

```

```
    cout << "Breadth First Traversal for given graph (with 0 as starting node): "<<endl;
    dg.BFS(0);

    return 0;
}
```

OUTPUT:



The screenshot shows a Windows command prompt window with the title bar "C:\Users\Warda Haider\Desktop\sub folder\cpp\Alg\BFS.exe". The output of the program is displayed in the console:

```
Breadth First Traversal for given graph (with 0 as starting node):
0 1 2 3
-----
Process exited after 0.1592 seconds with return value 0
Press any key to continue . . .
```

DEPTH FIRST SEARCH CODE:

```
#include <iostream>
#include <list>
using namespace std;
//graph class for DFS traversal
```

```

class DFSGraph
{
    int V; // No. of vertices
    list<int> *adjList; // adjacency list
    void DFS_util(int v, bool visited[]); // A function used by DFS
public:
    // class Constructor
    DFSGraph(int V){
        this->V = V;
        adjList = new list<int>[V];
    }
    // function to add an edge to graph
    void addEdge(int v, int w){
        adjList[v].push_back(w); // Add w to v's list.
    }

    void DFS(); // DFS traversal function
};

void DFSGraph::DFS_util(int v, bool visited[])
{
    // current node v is visited
    visited[v] = true;
    cout << v << " ";

    // recursively process all the adjacent vertices of the node
    list<int>::iterator i;
    for(i = adjList[v].begin(); i != adjList[v].end(); ++i)
        if(!visited[*i])
            DFS_util(*i, visited);
}

// DFS traversal
void DFSGraph::DFS()
{

```

```

        // initially none of the vertices are visited
        bool *visited = new bool[V];
        for (int i = 0; i < V; i++)
            visited[i] = false;

        // explore the vertices one by one by recursively calling DFS_util
        for (int i = 0; i < V; i++)
            if (visited[i] == false)
                DFS_util(i, visited);
    }

int main()
{
    // Create a graph
    DFSGraph dg(4);
    dg.addEdge(0, 1);
    dg.addEdge(0, 2);
    dg.addEdge(1, 0);
    dg.addEdge(1, 2);
    dg.addEdge(2, 3);
    dg.addEdge(3, 2);

    cout << "Depth-first traversal for the given graph:"<<endl;
    dg.DFS();

    return 0;
}

```

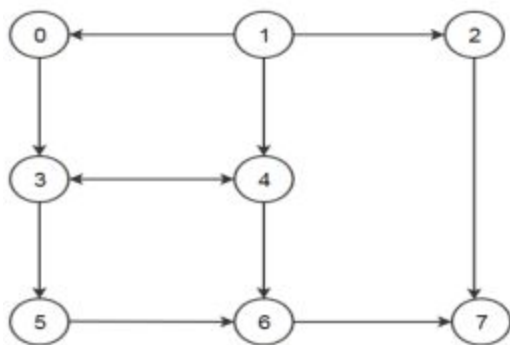
OUTPUT:

```

Depth-first traversal for the given graph:
0 1 2 3
-----
Process exited after 0.1159 seconds with return value 0
Press any key to continue . . .

```

2. Use BFS (diagram below) to Find the path between given vertices in a directed graph



CODE:

```

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

int countpaths=0;
// utility function for printing
// the found path in graph
void printpath(vector<int>& path)
{
    countpaths++;
    cout<<"Path "<<countpaths<<" :";
    int size = path.size();
    for (int i = 0; i < size; i++)
        cout << path[i] << " ";

    cout << endl;
}

```



```

// utility function to check if current
// vertex is already present in path
int isNotVisited(int x, vector<int>& path)
{
    int size = path.size();
    for (int i = 0; i < size; i++)
        if (path[i] == x)
            return 0;
    return 1;
}

// utility function for finding paths in graph
// from source to destination
void findpaths(vector<vector<int> >&g, int src,
               int dst, int v)
{
    // create a queue which stores
    // the paths
    queue<vector<int> > q;

    // path vector to store the current path
    vector<int> path;
    path.push_back(src);
    q.push(path);
    while (!q.empty()) {
        path = q.front();
        q.pop();
        int last = path[path.size() - 1];

        // if last vertex is the desired destination
        // then print the path
        if (last == dst)
            printpath(path);

        // traverse to all the nodes connected to
        // current vertex and push new path to queue
        for (int i = 0; i < g[last].size(); i++) {
            if (isNotVisited(g[last][i], path)) {
                vector<int> newpath(path);
            }
        }
    }
}

```

```

        newpath.push_back(g[last][i]);
        q.push(newpath);
    }
}
}

vector<vector<int> > g;
int v = 8;

void addEdge(int i, int j){
    g[i].push_back(j);
}

// driver program
int main()
{
    g.resize(8);
    // construct a graph
    addEdge(0, 3);
    addEdge(1, 2);
    addEdge(1, 0);
    addEdge(1, 4);
    addEdge(2, 7);
    addEdge(3, 5);
    addEdge(3, 4);
    addEdge(4, 6);
    addEdge(4, 3);
    addEdge(5, 6);
    addEdge(6, 7);
    int src, dst;
    cout<<"ENTER SOURCE: ";
    cin>>src;
    cout<<endl;

    cout<<"ENTER DEST: ";
    cin>>dst;
    cout<<endl;

    // function for finding the paths
    findpaths(g, src, dst, v);
    if(countpaths==0){

```

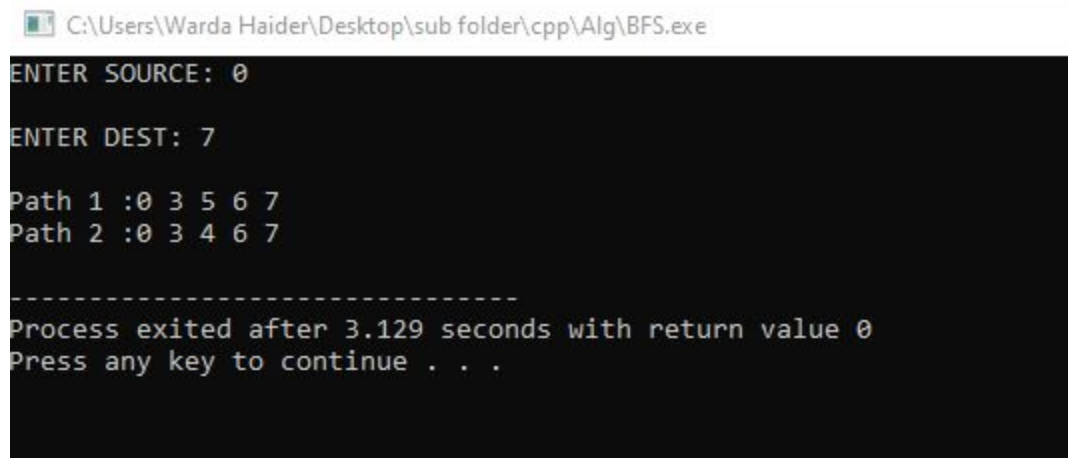
```

        cout << "There is no path from src " << src << " to dst " << dst << " are \n";
    }

    return 0;
}

```

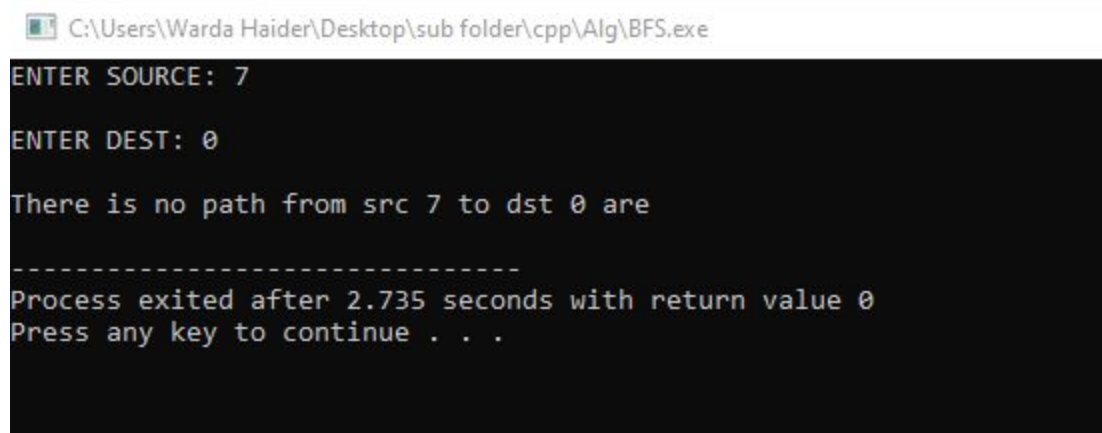
OUTPUT:



```

C:\Users\Warda Haider\Desktop\sub folder\cpp\Alg\BFS.exe
ENTER SOURCE: 0
ENTER DEST: 7
Path 1 :0 3 5 6 7
Path 2 :0 3 4 6 7
-----
Process exited after 3.129 seconds with return value 0
Press any key to continue . . .

```



```

C:\Users\Warda Haider\Desktop\sub folder\cpp\Alg\BFS.exe
ENTER SOURCE: 7
ENTER DEST: 0
There is no path from src 7 to dst 0 are
-----
Process exited after 2.735 seconds with return value 0
Press any key to continue . . .

```

Q3. Write an algorithm to find the minimum no. of throws required to win the snake & Ladder game by using BFS approach , also analyze its time complexity .(Give implementation as well).

ALGORITHM:

1. Each vertex will store 2 information, cell number and number of moves required to reach that cell. (cell, moves)

2. Start from cell (vertex) 1, add it to the queue.
3. For any index = i, Remove vertex 'i' from queue and add all the vertices to which can be reached from vertex 'i' by throwing the dice once and update the moves for each vertex (moves = moves to reach cell 'i' + 1 if no snake or ladder is present else moves = cell to which snake or ladder will leads to)
4. Remove a vertex from the queue and follow the previous step.
5. Maintain visited[] array to avoid going in loops.
6. Once reached to the end(destination vertex), stop.

Time Complexity

Time complexity of the above solution is $O(N)$ as every cell is added and removed only once from the queue. And a typical enqueue or dequeue operation takes $O(1)$ time.

CODE:

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

// An entry in queue used in BFS
struct Vertex
{
    int v;    // Vertex number
    int moves; // Distance of this vertex from source
};

int getMinDiceThrows(int move[], int N)
{
    // The graph has N vertices. Mark all the vertices as
    // not visited
    bool *visited = new bool[N];
    for (int i = 0; i < N; i++)
        visited[i] = false;

    // Create a queue for BFS
    queue<Vertex > q;

    // Mark the node 0 as visited and enqueue it.
    visited[0] = true;
```

```

Vertex s = {0, 0}; // distance of 0't vertex is also 0
q.push(s); // Enqueue 0'th vertex

// Do a BFS starting from vertex at index 0
Vertex qe; // A queue entry (qe)
while (!q.empty())
{
    qe = q.front();
    int v = qe.v; // vertex no. of queue entry

    // If front vertex is the destination vertex,
    // we are done
    if (v == N-1)
        break;

    // Otherwise dequeue the front vertex and enqueue
    // its adjacent vertices
    q.pop();
    for (int j=v+1; j<=(v+6) && j<N; ++j)
    {
        // If this cell is already visited, then ignore
        if (!visited[j]){
            Vertex a;
            a.moves = (qe.moves + 1);
            visited[j] = true;

            // Check if there a snake or ladder at 'j'
            // then tail of snake or top of ladder
            // become the adjacent of 'i'
            if (move[j] != -1)
                a.v = move[j];
            else
                a.v = j;
            q.push(a);
        }
    }
    return qe.moves;
}

int main()
{
    int N = 30;

```

```
int board[N];
for (int i = 0; i<N; i++)
    board[i] = -1;

// Ladders
board[2] = 21;
board[4] = 7;
board[10] = 25;
board[19] = 28;

// Snakes
board[26] = 0;
board[20] = 8;
board[16] = 3;
board[18] = 6;
cout << "Min Dice throws required is " << getMinDiceThrows(board, N);
return 0;
}
```

Lab no 13

Mahnoor SE-19013 & Maria Nasir SE-007

Rat in a Maze Problem

A Maze is given as $N \times N$ binary matrix of blocks where source block is the upper left most block i.e., `maze[0][0]` and destination block is lower rightmost block i.e., `maze[N-1][N-1]`. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down. In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

1. What approach do you use to solve problem and why?

Rat in a MAZE is a problem that can be solved using Backtracking. Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally. Solving one piece at a time, and removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree) is the process of backtracking.

Approach: Form a recursive function, which will follow a path and check if the path reaches the destination or not. If the path does not reach the destination then backtrack and try other paths.

2. Give the algorithm of the above problem. Also analyze the time & space complexity of your algorithm.

ALGORITHM:

X, Y: Position of rat

Sol: 2D array of solution matrix

Maze: 2D array of Initial matrix

Begin

[Create Matrix initially filled with 0's]

SolveMaze (MAZE, SOL, X,Y)

[destination is reached]

1. If ($x = N - 1$ and $y = N - 1$) then

Return True

2. Else

[position of rat is in the matrix and destination is not reached]

i. If ($x \geq 0$ and $x < N$ and $y \geq 0$ and $y < N$ and $\text{maze}[x][y] \neq 0$)

Set $\text{sol}[i][j] = 1$

Repeat step (a) for $i=0$ to N

a) If ($i \leq \text{maze}[x][y]$)

[Move forward in x direction]

1) if ($\text{solveMazeUtil}(\text{maze}, x + i, y, \text{sol}) == \text{True}$):

Return True

[If moving in x direction doesn't give solution

then Move down in y direction]

2) if ($\text{SolveMaze}(\text{maze}, x, y + i, \text{sol}) == \text{True}$):

Return True

[If none of the above movements work then unmark x and y]

Set $\text{sol}[x][y] = 0$

Return False

[End of loop]

Return False

End

Time complexity: $O(2^{(n^2)})$.

For each cell there are two possible choices either can go in x direction or y direction so 2 to the power total cells will the time complexity. The recursion can run upper-bound $2^{(n^2)}$ times.

Space Complexity: $O(N^2)$

Output matrix is required so an extra space of size $n*n$ is needed.

3. Give implementation of Rat in a Maze problem.

CODE:

```
// C++ program to solve Rat in a Maze problem using backtracking //
```

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



```

// Maze size

#define N 4

bool solveMazeUtil(

    int maze[N][N], int x,

    int y, int sol[N][N]

);

// A utility function to print solution matrix sol[N][N]

void printSolution(int sol[N][N])

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

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

            printf(" %d ", sol[i][j]);

        printf("\n");

    }

}

// A utility function to check if x,y is valid index for N*N maze

bool isSafe(int maze[N][N], int x, int y)

{
    // if (x, y outside maze) return false

    if (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1)

        return true;

    return false; }

bool solveMaze(int maze[N][N]){

    int sol[N][N] = { { 0, 0, 0, 0 },

                      { 0, 0, 0, 0 },

                      { 0, 0, 0, 0 },

```

```
{ 0, 0, 0, 0 } };
```

```
    if (solveMazeUtil( maze, 0, 0, sol) == false) {
        printf("Solution doesn't exist");
        return false;}
    printSolution(sol);
    return true;
}

// A recursive utility function to solve Maze problem
bool solveMazeUtil( int maze[N][N], int x, int y, int sol[N][N]){
    // if (x, y is goal) return true//
    if (x == N - 1 && y == N - 1 && maze[x][y] == 1)
    {
        sol[x][y] = 1;
        return true;}
    // Check if maze[x][y] is valid
    if (isSafe(maze, x, y) == true)
    {
        // mark x, y as part of solution path
        sol[x][y] = 1;
        /// Move forward in x direction //
        if (solveMazeUtil( maze, x + 1, y, sol)== true)
            return true;
        if (solveMazeUtil(maze, x, y + 1, sol)== true)
            return true;
        sol[x][y] = 0;
        return false;
    }
    return false;
}


// driver program to test above function
```

```
int main()
{
    int maze[N][N] = { { 1, 0, 0, 0 },
                        { 1, 1, 0, 1 },
                        { 0, 1, 0, 0 },
                        { 1, 1, 1, 1 } };

    solveMaze(maze);

    return 0;
}
```

OUTPUT:

 d:\project\hello\bin\Debug\hello.exe

```
1 0 0 0
1 1 0 0
0 1 0 0
0 1 1 1
```

```
Process returned 0 (0x0)   execution time : 0.020 s
Press any key to continue.
```

Lab no 14 Warda Haider SE-19016

N- Queen Problem

The N Queen is the problem of placing N chess queens on an $N \times N$ chessboard so that no two queens attack each other. For example, the following is a solution for the 4 Queen problem. The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, the following is the output matrix for above 4 queen solution.

{ 0, 1, 0, 0}

{ 0, 0, 0, 1}

{ 1, 0, 0, 0}

{ 0, 0, 1, 0}

Q1. What approach do you use to solve problems and why?

Backtracking Approach using recursion has been used to solve this problem. We have placed the queen one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

Q2. Give the algorithm of the above problem , Also analyze the time & space complexity of your algorithm.

ALGORITHM:

1) Start from the column which is at leftmost

2) If all queens are placed

 return true

3) Try all rows in the current column.

 Do the following for every tried row.

 a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.

b) If placing the queen in [row, column] leads to a solution then return true.

c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.

3) If all rows have been tried and nothing worked, return false to trigger backtracking.

Time Complexity:

$T(N) = N * T(N-1) + c$. Thus after reducing it the dominating term will be $n!$ therefore **$O(n!)$** is the time complexity

Space Complexity:

We are using backtracking that is recursive calls are made and they are held up in stack therefore the space complexity is $n \times n$ as we are dealing with an array i.e. `array[n][n]` we check for each cell that whether the queen can be placed or not. **$O(n \times n)$** is the space complexity.

3. Give implementation of 8 Queen Problem.

CODE:

```
#define N 8
#include <iostream>
using namespace std;

/* To print solution */
void printSol(int plane[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            cout << "\t" << plane[i][j];
        cout << "\n";
    }
}

/* To check if a queen can
   be placed on plane[row][col]. */

bool canPlace(int plane[N][N], int row, int col)
{
    int i, j;
```

```

/* Check left side row*/
for (i = 0; i < col; i++)
    if (plane[row][i])
        return false;

/* Check left side upper diagonal */
for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
    if (plane[i][j])
        return false;

/* Check left side lower diagonal */
for (i = row, j = col; j >= 0 && i < N; i++, j--)
    if (plane[i][j])
        return false;

return true;
}

/* A recursive function */
bool solve8QUtil(int plane[N][N], int col)
{
    /* base case*/
    if (col >= N)
        return true;

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

        if (canPlace(plane, i, col)) {
            /* Place this queen at plane[i][col] */
            plane[i][col] = 1;

            /* recur to place rest of the queens */
            if (solve8QUtil(plane, col + 1))
                return true;

            plane[i][col] = 0; // BACKTRACK
        }
    }
}

/* If the queen cannot be placed in any row in

```

```

        this column col then return false */
    return false;
}

/* Uses Backtracking feature to solve Problem.*/
bool solveNQ()
{
    int plane[N][N] = { { 0, 0, 0, 0 }, { 0, 0, 0, 0 }, { 0, 0, 0, 0 }, { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 }, { 0, 0, 0, 0 }, { 0, 0, 0, 0 }, { 0, 0, 0, 0 } };

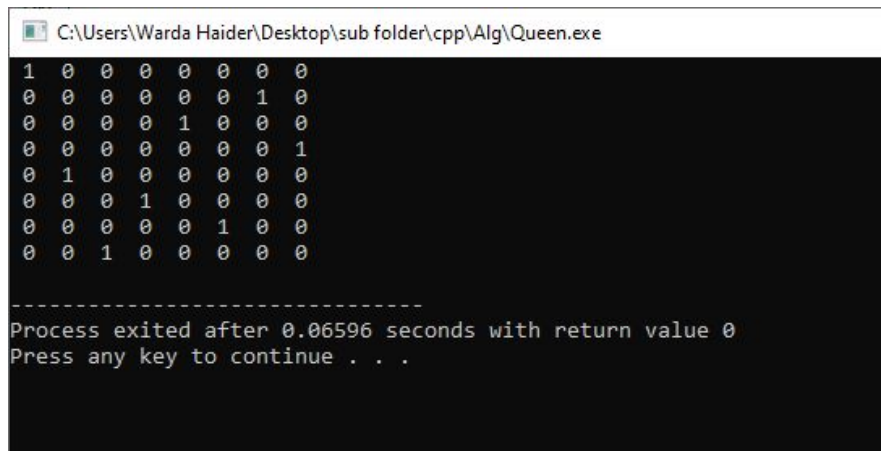
    if (solve8QUtil(plane, 0) == false) {
        cout<<"Solution does not exist";
        return false;
    }

    printSol(plane);
    return true;
}

int main()
{
    solveNQ();
    return 0;
}

```

OUTPUT:



```

C:\Users\Warda Haider\Desktop\sub folder\cpp\Alg\Queen.exe
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0

-----
Process exited after 0.06596 seconds with return value 0
Press any key to continue . . .

```

