



Institute of Management Entrepreneurship and Engineering

Technology (ViMEET)

Computer Science & Engineering (AI & ML) Department

Lab Manual

Analysis of Algorithm Lab

(2163115)

Academic Year

2025-26

S.E. (AIML)

Semester – III

Prepared by

Prof. Sneha Nimbare

CSE(AIML) Department

ViMEET Khalapur

Prerequisites:

- Basic knowledge of programming and data structure.

Lab Objectives:

- To introduce the methods of designing and analyzing algorithms.
- Design and implement efficient algorithms for a specified application.
- Strengthen the ability to identify and apply the suitable algorithm for the given real-world problem.
- Analyze worst-case running time of algorithms and understand fundamental algorithmic problems.

Lab Outcomes:

- Implement the algorithms using different approaches.
- Analyze the complexities of various algorithms.
- Compare the complexity of the algorithms for specific problem.



Vishwaniketan's Institute of Management Entrepreneurship and Engineering
Technology (ViMEET)

List of Experiments

Sr. No.	Name of Experiments
1	Experiment based on common mathematical functions: Insertion Sort
2	Experiment based on divide and conquers approach: Merge sort
3	Experiment based on greedy approach: Single Source Shortest Path- Dijkstra Algorithm
4	Travelling salesperson problem Longest common subsequence
5	Experiment based on graph Algorithms: BFS, DFS
6	Experiment using Backtracking strategy: N-queen problem
7	Experiment using branch and bound strategy: 15 Puzzle Problem
8	Experiment based on string matching/amortized analysis: The Naïve string-matching Algorithms
9	Implementation Min-Max Algorithm
10	To Implement Single Source Shortest Path Bellman Ford.



**Vishwaniketan's Institute of Management Entrepreneurship and Engineering
Technology (ViMEET)**

INDEX

Sr. No.	Name of Experiments	DOP	DOS	Marks	Sign
1	Experiment based on common mathematical functions: Insertion Sort				
2	Experiment based on divide and conquers approach: Merge sort				
3	Experiment based on greedy approach: Single Source Shortest Path- Dijkstra Algorithm				
4	Travelling salesperson problem Longest common subsequence				
5	Experiment based on graph Algorithms: BFS, DFS				
6	Experiment using Backtracking strategy: N-queen problem				
7	Experiment using branch and bound strategy: 15 Puzzle Problem				
8	Experiment based on string matching/amortized analysis: The Naïve string-matching Algorithms				
9	Implementation Min-Max Algorithm				
10	To Implement Single Source Shortest Path Bellman Ford.				

EXPERIMENT NO. 1

AIM: Experiment based on common mathematical functions: **Insertion Sort**

THEORY:

- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.
- Insertion sort works similarly as we sort cards in our hand in a card game.
- We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put at their right place.
- Following are some of the important **characteristics of Insertion Sort**:
 - i. It is efficient for smaller data sets, but very inefficient for larger lists.
 - ii. Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
 - iii. It is better than Selection Sort and Bubble Sort algorithms.
 - iv. Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
 - v. It is a **stable** sorting technique, as it does not change the relative order of elements which are equal.

ALGORITHM:

Step1- Collect Unsorted data as an input.

Step 2 – If it is the first element in the List, then consider it is already sorted. return 1;

Step 3 – Pick next element

Step 4 – Compare with all elements in the sorted sub-list

Step 5 – Shift all the sorted elements as per the comparison.

Step 6 – Insert the value

Step 7 – Repeat until list is sorted

Step 8- Stop.

PROGRAM CODE:

```
#include <stdio.h>
#include<conio.h>
void insertion_sort(); //Function Declaration int a[50],n; //Global variable
void main()
{
    int i;
    printf("\nEnter size of an array: ");
    scanf("%d", &n);
    printf("\nEnter elements of an array:\n"); //Accept Elements in array
    for(i=0; i<n; i++) scanf("%d", &a[i]);
    insertion_sort(); //Function Call
    printf("\n\nAfter sorting:\n");
    for(i=0; i<n; i++)
        printf("\n%d", a[i]);
    getch();
}
void insertion_sort()
{
    int p, i, temp;
    for(p=1; p<n; p++) //p=pass of insertion //sort
    {
        temp = a[p]; i = p-1;
        while (i>=0 && a[i]>temp)
            // i= index of array
        {
            a[i+1] = a[i]; // shifting
            //elements to right i--;
        }
        a[i+1] = temp; //inserting element to new //index position
    }
}
```

OUTPUT:

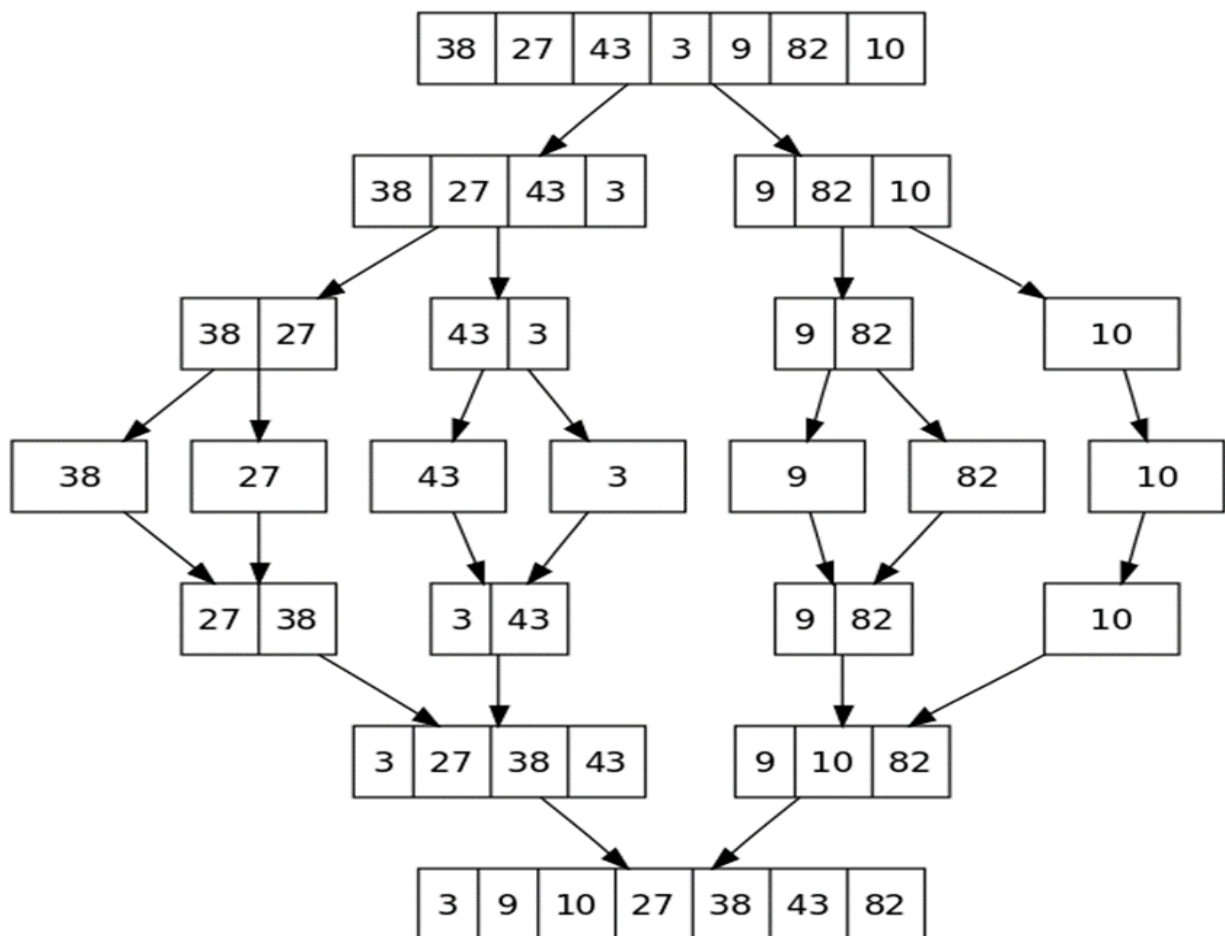
```
Enter size of an array: 4  
  
Enter elements of an array:  
40 60 10 30  
  
After sorting:  
  
10  
30  
40  
60
```

EXPERIMENT NO. 2

AIM: Experiment based on divide and conquers approach: **Merge sort.**

THEORY:

- Merge sort runs in $O(n \log n)$ running time.
- It is very efficient sorting algorithm with near optimal number of comparison.
- Recursive algorithm used for merge sort comes under the category of divide and conquer technique.
- An array of n elements is split around its centre producing two smaller arrays. After these two arrays are sorted independently, they can be merged to produce the final sorted array.
- The process of splitting and merging can be carried recursively till there is only one element in the array. An array with 1 element is always sorted.



ALGORITHM:

1. Find the middle index of the array to divide it in two halves: $m = (l+r)/2$.
2. Call **MergeSort** for first half: **mergeSort**(array, l, m)
3. Call **mergeSort** for second half: **mergeSort**(array, m+1, r)
4. Recursively, **merge** the two halves in a **sorted** manner, so that only one **sorted** array is left:

PROGRAM CODE

```
#include<stdio.h>

void mergesort(int a[],int i,int j);

void merge(int a[],int i1,int j1,int i2,int j2);

int main()
{
    int a[30],n,i;

    printf("Enter no of elements:");

    scanf("%d",&n);

    printf("Enter array elements:");

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

        scanf("%d",&a[i]);

    mergesort(a,0,n-1);

    printf("\nSorted array is :");

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

        printf("%d ",a[i]);

    return 0;
}

void mergesort(int a[],int i,int j)
```

```

{
int mid;
if(i<j)
{
mid=(i+j)/2;
mergesort(a,i,mid); //left recursion
mergesort(a,mid+1,j); //right recursion
merge(a,i,mid,mid+1,j); //merging of two sorted sub-arrays
}
}

```

```

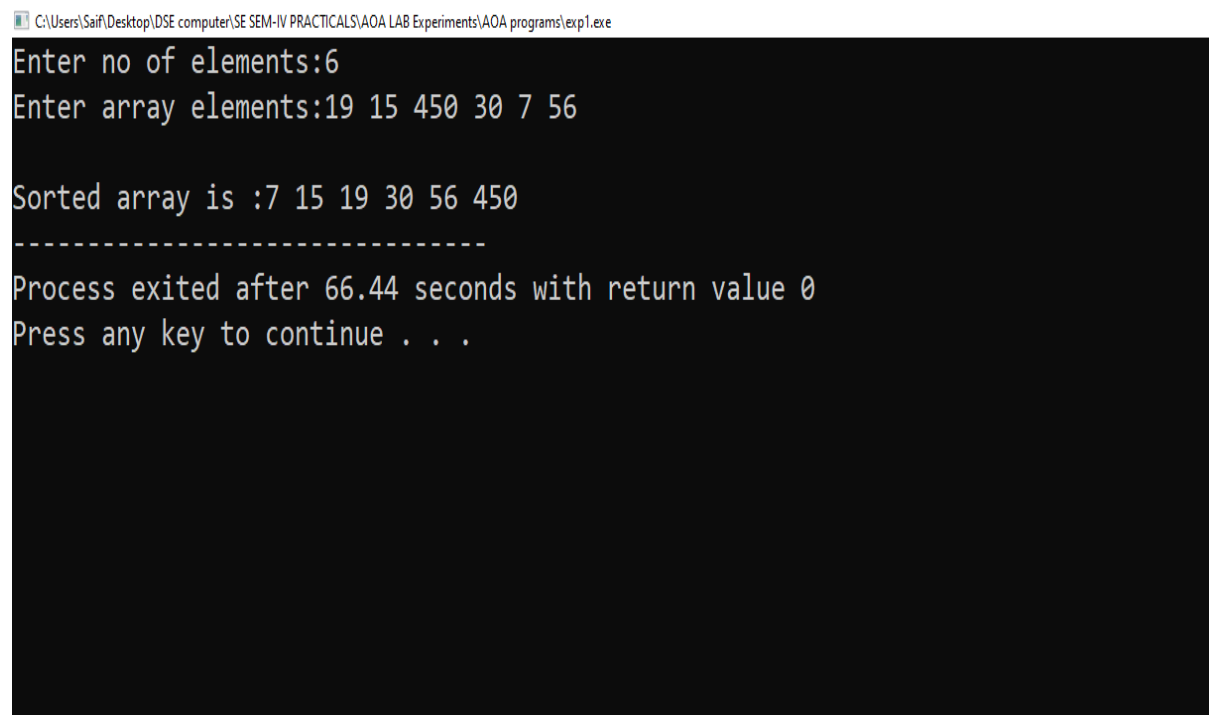
void merge(int a[],int i1,int j1,int i2,int j2)
{
int temp[50];      //array used for merging
int i,j,k;
i=i1;   //beginning of the first list
j=i2;   //beginning of the second list
k=0;
while(i<=j1 && j<=j2) //while elements in both lists
{
if(a[i]<a[j])
temp[k++]=a[i++];
else
temp[k++]=a[j++];
}
while(i<=j1) //copy remaining elements of the first list
temp[k++]=a[i++];

```

```
while(j<=j2) //copy remaining elements of the second list
temp[k++]=a[j++];

//Transfer elements from temp[] back to a[]
for(i=i1,j=0;i<=j2;i++,j++)
a[i]=temp[j];
}
```

OUTPUT:



```
C:\Users\Saif\Desktop\DSE computer\SE SEM-IV PRACTICALS\AOA LAB Experiments\AOA programs\exp1.exe
Enter no of elements:6
Enter array elements:19 15 450 30 7 56

Sorted array is :7 15 19 30 56 450
-----
Process exited after 66.44 seconds with return value 0
Press any key to continue . . .
```

EXPERIMENT NO. 3

AIM: Experiment based on greedy approach: **Single Source Shortest Path-Dijkstra Algorithm.**

THEORY:

- Dijkstra's Algorithm is a **greedy algorithm** used to find the **shortest path** from a **single source** vertex to **all other vertices** in a **graph with non-negative edge weights**.
- At every step, the algorithm **greedily selects the nearest unvisited vertex** (i.e. the vertex with the smallest known distance) and updates the shortest paths to its neighbors.
- It never reconsiders already visited vertices because it assumes the shortest path to them is already found.
- Works only with **non-negative weights**.

ALGORITHM:

Step 1: Initialize

- Set distance of all vertices to **infinity**, except the source (set to 0):
 $\text{dist}[\text{source}] = 0$
- Mark all vertices as **unvisited**.

Step 2: Repeat until all vertices are visited

- Pick the **unvisited vertex u with the smallest distance** ($\min(\text{dist}[v])$)
- Mark u as **visited**
- For each **unvisited neighbor v** of u:
 - If the total distance to v through u is **less than dist[v]**, then update it:
 $\text{dist}[v] = \text{dist}[u] + \text{weight}(u, v)$

Step 3: Stop when all vertices have been visited or when the shortest distances are finalized.

PROGRAM CODE:

```
#include<stdio.h>

void quicksort(int number[25],int first,int last){

    int i, j, pivot, temp;

    if(first<last){

        pivot=first;

        i=first;

        j=last;

        while(i<j){

            while(number[i]<=number[pivot]&& i<last)

                i++;

            while(number[j]>number[pivot])

                j--;

            if(i<j){

                temp=number[i];

                number[i]=number[j];

                number[j]=temp;

            }

        }

        temp=number[pivot];

        number[pivot]=number[j];

        number[j]=temp;

        quicksort(number,first,j-1);

        quicksort(number,j+1,last);

    }

}

int main(){
```

```

    int i, count, number[25];

    printf("How many elements are u going to enter?: ");
    scanf("%d",&count);

    printf("Enter %d elements: ", count);

    for(i=0;i<count;i++)
    scanf("%d",&number[i]);

    quicksort(number,0,count-1);


    printf("Order of Sorted elements: ");

    for(i=0;i<count;i++)
    printf(" %d",number[i]);

    return 0;
}

```

OUTPUT:

 C:\Users\Saif\Desktop\DSE computer\SE SEM-IV PRACTICALS\AOA LAB Experiments\AOA programs\exp3.exe

```

How many elements are u going to enter?: 5
Enter 5 elements: 19 45 -3 86 15
Order of Sorted elements:  -3 15 19 45 86

```

```

Process exited after 35.7 seconds with return value 0
Press any key to continue . . .

```

EXPERIMENT NO. 4

AIM: Experiment based on **Longest common subsequence**

THEORY:

- The **Longest Common Subsequence (LCS)** of two strings is the **longest sequence** that appears in **both strings in the same order**, but **not necessarily contiguously**.
- The longest common subsequence problem is finding the longest sequence which exists in both the given strings.

ALGORITHM:

Step 1: Construct an empty adjacency table with the size, $n \times m$, where n = size of sequence **X** and m = size of sequence **Y**. The rows in the table represent the elements in sequence X and columns represent the elements in sequence Y.

Step 2: The zeroth rows and columns must be filled with zeroes. And the remaining values are filled in based on different cases, by maintaining a counter value.

- If the counter encounters common element in both X and Y sequences, increment the counter by 1.
- If the counter does not encounter common elements in X and Y sequences at $T[i, j]$, find the maximum value between $T[i-1, j]$ and $T[i, j-1]$ to fill it in $T[i, j]$.

Step 3: Once the table is filled, backtrack from the last value in the table. Backtracking here is done by tracing the path where the counter incremented first.

Step 4: The longest common subsequence obtained by noting the elements in the traced path.

PROGRAM CODE:

```
#include<stdio.h>
#include<string.h>
int i,j,m,n,a,c[20][20];
char x[15],y[15],b[20][20];
```

```

void print_lcs(int i,int j);

void lcs_length(void);

void main()
{
printf("Enter 1st sequence : ");
scanf("%s",x);
printf("Enter 2nd sequence : ");
scanf("%s",y);
lcs_length();
printf("\n");
}

void print_lcs(int i,int j)
{
if(i==0 || j==0)
return;
if(b[i][j]=='c')
{
print_lcs(i-1,j-1);
printf(" %c",x[i-1]);
}
else if(b[i][j]=='u')
print_lcs(i-1,j);
else
print_lcs(i,j-1);
}

void lcs_length(void)
{
m=strlen(x);
n=strlen(y);

```



```

for(i=0;i <= m;i++)
c[i][0]=0;
for(i=0;i <= n;i++)
{
printf("0 \t");
c[0][i]=0;
}
printf("\n");
for(i=1;i <= m;i++)
{
printf("0 \t");
for(j=1;j <= n;j++)
{
if(x[i-1]==y[j-1])
{
c[i][j]=c[i-1][j-1]+1;
b[i][j]='c';
printf("%d D\t",c[i][j]);
}
else if(c[i-1][j] >= c[i][j-1])
{
c[i][j]=c[i-1][j];
b[i][j]='u';
printf("%d U\t",c[i][j]);
}
else
{
c[i][j]=c[i][j-1];
b[i][j]='l';

```

```

printf("%d L\t",c[i][j]);

}

}

printf(" \n");

}

printf("\nlongest common subsequence is :");

print_lcs(m,n);

printf("\n");

printf("The length of Subsequence is: %d",c[m][n]);

}

```

OUTPUT:

```

C:\Users\Saif\Desktop\DSE computer\SE SEM-IV PRACTICALS\AOA LAB Experiments\AOA programs\exp7.exe
Enter 1st sequence : 1000101011
Enter 2nd sequence : 00011101
0      0      0      0      0      0      0      0      0
0      0 U     0 U     0 U     1 D     1 D     1 D     1 L     1 D
0      1 D     1 D     1 D     1 U     1 U     1 U     2 D     2 L
0      1 D     2 D     2 D     2 L     2 L     2 L     2 D     2 U
0      1 D     2 D     3 D     3 L     3 L     3 L     3 D     3 L
0      1 U     2 U     3 U     4 D     4 D     4 D     4 L     4 D
0      1 D     2 D     3 D     4 U     4 U     4 U     5 D     5 L
0      1 U     2 U     3 U     4 D     5 D     5 D     5 U     6 D
0      1 D     2 D     3 D     4 U     5 U     5 U     6 D     6 U
0      1 U     2 U     3 U     4 D     5 D     6 D     6 U     7 D
0      1 U     2 U     3 U     4 D     5 D     6 D     6 U     7 D

longest common subsequence is : 0 0 0 1 1 0 1
The length of Subsequence is: 7

-----
Process exited after 28.78 seconds with return value 13
Press any key to continue . . .

```

EXPERIMENT NO. 5

AIM: Experiment based on graph Algorithms: **BFS, DFS**

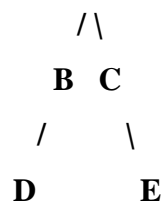
THEORY:

- BFS (Breadth-First Search) and DFS (Depth-First Search) are fundamental graph traversal algorithms.
- BFS explores a graph level by level, while DFS explores as far as possible along each branch before backtracking. They are used for various tasks like finding the shortest path, cycle detection, and topological sorting.
- Graph traversal means visiting **all the nodes (vertices)** in a graph in a **systematic way**.
- There are two main types:

1. Breadth-First Search (BFS)

- **Breadth-First Search (BFS)** is a graph traversal technique that explores all the neighbouring vertices at the current depth before moving on to the next level of vertices.
- **BFS Algorithm:**
 - Step1:** Start from the **source node**.
 - Step2:** Mark the node as **visited** and **enqueue it**
 - Step3:** While the queue is not empty:
 - Dequeue a vertex v
 - Visit all **unvisited neighbors** of v
 - Mark them as visited and enqueue them

- Exapmle: **A**



- BFS starting from A: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$

2. Depth-First Search (DFS)

- **Depth-First Search (DFS)** is a graph traversal technique that explores as far as possible along each branch before backtracking.

- **DFS Algorithm:**

Step1: Start from the **source node**

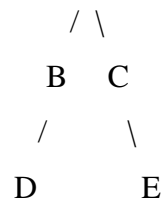
Step2: Mark it as **visited**

Step3: Visit any **unvisited neighbour**

Step 4: Repeat the process **recursively**

Step5: Backtrack when no unvisited neighbours remain.

- **Example:** A



- DFS starting from A: $A \rightarrow B \rightarrow D \rightarrow C \rightarrow E$

PROGRAM CODE:

```
#include<stdio.h>

int q[20],top=-1,front=-1,rear=-1,a[20][20],vis[20],stack[20];

int delete();

void add(int item);

void bfs(int s,int n);

void dfs(int s,int n);

void push(int item);

int pop();

void main()

{

int n,i,s,ch,j;

char c,dummy;

printf("ENTER THE NUMBER VERTICES ");

scanf("%d",&n);

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

```

{
for(j=1;j<=n;j++)
{
printf("ENTER 1 IF %d HAS A NODE WITH %d ELSE 0 ",i,j);
scanf("%d",&a[i][j]);
}
}
printf("THE ADJACENCY MATRIX IS\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
printf(" %d",a[i][j]);
}
printf("\n");
}
do
{
for(i=1;i<=n;i++)
vis[i]=0;
printf("\nMENU");
printf("\n1.B.F.S");
printf("\n2.D.F.S");
printf("\nENTER YOUR CHOICE");
scanf("%d",&ch);
printf("ENTER THE SOURCE VERTEX :");
scanf("%d",&s);
switch(ch)
{
case 1: bfs(s,n);
break;
case 2:
dfs(s,n);
break;

```

```

}
printf("DO U WANT TO CONTINUE(Y/N) ? ");
scanf("%c",&dummy);
scanf("%c",&c);
}while((c=='y')||(c=='Y'));
}

//*****BFS(breadth-first search) code*****//

void bfs(int s,int n)
{
int p,i;
add(s);
vis[s]=1;
p=delete();
if(p!=0)
printf(" %d",p);
while(p!=0)
{
for(i=1;i<=n;i++)
if((a[p][i]!=0)&&(vis[i]==0))
{
add(i);
vis[i]=1;
}
p=delete();
if(p!=0)
printf(" %d ",p);
}
for(i=1;i<=n;i++)
if(vis[i]==0)
bfs(i,n);
}

void add(int item)
{
if(rear==19)

```

```

printf("QUEUE FULL");
else
{
if(rear==-1)
{
q[++rear]=item;
front++;
}
else
q[++rear]=item;
}
}
int delete()
{
int k;
if((front>rear)|| (front==-1))
return(0);
else
{
k=q[front++];
return(k);
}
}
//*****DFS(depth-first search) code*****//
void dfs(int s,int n)
{
int i,k;
push(s);
vis[s]=1;
k=pop();
if(k!=0)
printf(" %d ",k);
while(k!=0)
{

```

```

for(i=1;i<=n;i++)
if((a[k][i]!=0)&&(vis[i]==0))
{
push(i);
vis[i]=1;
}
k=pop();
if(k!=0)
printf(" %d ",k);
}
for(i=1;i<=n;i++)
if(vis[i]==0)
dfs(i,n);
}
void push(int item)
{
if(top==19)
printf("Stack overflow ");
else
stack[++top]=item;
}
int pop()
{
int k;
if(top==-1)
return(0);
else
{
k=stack[top--];
return(k);
}
}

```


OUTPUT:

```
ENTER THE NUMBER VERTICES 3
ENTER 1 IF 1 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 1 HAS A NODE WITH 2 ELSE 0 1
ENTER 1 IF 1 HAS A NODE WITH 3 ELSE 0 0
ENTER 1 IF 2 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 2 HAS A NODE WITH 2 ELSE 0 0
ENTER 1 IF 2 HAS A NODE WITH 3 ELSE 0 1
ENTER 1 IF 3 HAS A NODE WITH 1 ELSE 0 0
ENTER 1 IF 3 HAS A NODE WITH 2 ELSE 0 1
ENTER 1 IF 3 HAS A NODE WITH 3 ELSE 0 1
THE ADJACENCY MATRIX IS
 1 1 0
 1 0 1
 0 1 1
```

```
MENU
1.B.F.S
2.D.F.S
ENTER YOUR CHOICE 1
ENTER THE SOURCE VERTEX : 2
 2 1 3 DO U WANT TO CONTINUE(Y/N) ? y
```

```
MENU
1.B.F.S
2.D.F.S
ENTER YOUR CHOICE2
ENTER THE SOURCE VERTEX :2
 2 3 1 DO U WANT TO CONTINUE(Y/N) ?
```

EXPERIMENT NO. 6

AIM: Experiment using Backtracking strategy: **N-queen problem**

THEORY:

- The **N-Queen problem** is a classic combinatorial problem in computer science and artificial intelligence.
- The goal is to place **N queens** on an **N×N chessboard** such that **no two queens threaten each other**:
 1. No two queens share the same **row**
 2. No two queens share the same **column**
 3. No two queens share the same **diagonal**
- The problem is typically solved using a backtracking algorithm.
- **N-Queen Algorithm:**
 - Step 1:** Place the first queen in the top-left cell of the chessboard.
 - Step 2:** After placing a queen in the first cell, mark the position as a part of the solution and then recursively check if this will lead to a solution.
 - Step 3:** Now, if placing the queen doesn't lead to a solution. Then go to the first step and place queens in other cells. Repeat until all cells are tried.
 - Step 4:** If placing queen returns a lead to solution return TRUE.
 - Step 5:** If all queens are placed return TRUE.
 - Step 6:** If all rows are tried and no solution is found, return FALSE.

PROGRAM CODE:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 20
int board[MAX];

bool isSafe(int row, int col, int n) {
    for (int i = 0; i < row; i++) {
```

```

        if (board[i] == col ||
            board[i] - i == col - row ||
            board[i] + i == col + row) {
            return false;
        }
    }
    return true;
}

```

```

void printSolution(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (board[i] == j)
                printf(" Q ");
            else
                printf(" . ");
        }
        printf("\n");
    }
    printf("\n");
}

```

```

void solveNQueen(int row, int n) {
    if (row == n) {
        printSolution(n);
        return;
    }
    for (int col = 0; col < n; col++) {
        if (isSafe(row, col, n)) {
            board[row] = col;
            solveNQueen(row + 1, n);
            // Backtracking happens here
        }
    }
}

```

```

}

int main() {
    int n;
    printf("Enter the number of queens: ");
    scanf("%d", &n);
    if (n > MAX) {
        printf("Maximum supported size is %d\n", MAX);
        return 1;
    }
    printf("Solutions to %d-Queens Problem:\n\n", n);
    solveNQueen(0, n);
    return 0;
}

```

OUTPUT:

```

Enter the number of queens: 4
Solutions to 4-Queens Problem:

```

```

.  Q  .  .
.  .  .  Q
Q  .  .  .
.  .  Q  .

.  .  Q  .
Q  .  .  .
.  .  .  Q
.  Q  .  .

```

EXPERIMENT NO. 7

AIM: Experiment using branch and bound strategy: **15 Puzzle Problem**

THEORY:

- The 15-puzzle problem can be solved using a branch and bound algorithm.
- This algorithm explores possible moves (branches) from the initial state, using a heuristic to estimate the distance to the goal and pruning paths that are unlikely to lead to an optimal solution.
- The [15-Puzzle](#) is a simple puzzle. It consists of a 4 x 4 grid with tiles numbered 1 through 15, the last tile omitted (call this the blank tile).
- The goal of the puzzle is to go from a scrambled position to the solved position, which is to arrange the numbers in ascending order from left to right, top to bottom. You can only move tiles adjacent to the blank tile, and you do this by “swapping” positions with the blank tile.

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Initial State

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal State

Algorithm:

Step1: It's a 4x4 board containing 15 numbered tiles (1 to 15) and one empty space.

Step2: The tiles are arranged randomly.

Step3: The goal is to move the tiles (by sliding them into the empty space) to reach the **goal configuration**

PROGRAM CODE:

```
#include <stdio.h>

#define SIZE 4

// Function to input a board from the user
void inputBoard(int board[SIZE][SIZE], const char* name) {
    printf("Enter %s state (use 0 for blank):\n", name);
    for (int i = 0; i < SIZE; i++)
        for (int j = 0; j < SIZE; j++)
            scanf("%d", &board[i][j]);
}

// Function to print a board
void printBoard(int board[SIZE][SIZE], const char* name)
{
    printf("\n%s state:\n", name);
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (board[i][j] == 0)
                printf(" ");
            else
                printf("%2d ", board[i][j]);
        }
        printf("\n");
    }
}

int main()
{
    int initial[SIZE][SIZE];
    int goal[SIZE][SIZE];

    // Input boards
    inputBoard(initial, "initial");
    inputBoard(goal, "goal");

    // Display boards
    printBoard(initial, "Initial");
```

```
    printBoard(goal, "Goal");  
    return 0;  
}
```

OUTPUT:

```
Enter initial state (use 0 for blank):
```

```
1  
2  
4  
3  
5  
7  
6  
8  
0  
9  
10  
11  
12  
14  
13  
15
```

```
Enter goal state (use 0 for blank):
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
0
```

Initial state:

```
1  2  4  3
5  7  6  8
   9 10 11
12 14 13 15
```

Goal state:

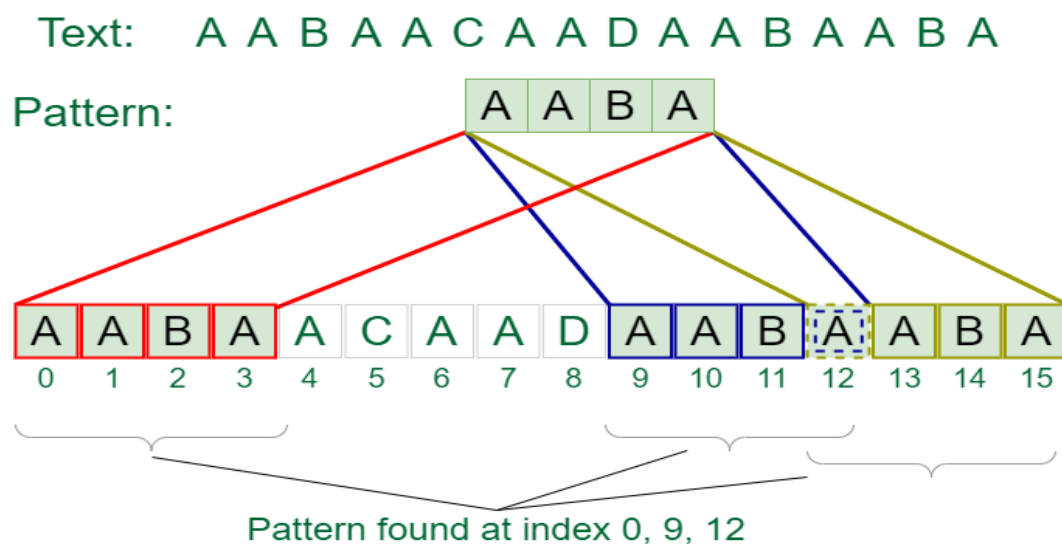
```
1  2  3  4
5  6  7  8
  9 10 11 12
13 14 15
```


EXPERIMENT NO. 8

AIM: Experiment based on string matching/amortized analysis: The Naïve String-matching Algorithms

THEORY:

- The naive string-matching algorithm is a brute-force approach to find a pattern within a larger text. It works by sliding the pattern one character at a time across the text and comparing characters at each position.
- **Example:**



- **Algorithm:**

Step 1: Set n = length of text T

Step 2: Set m = length of pattern P

Step 3: For shift s from 0 to $(n - m)$: // Slide pattern over text

Step 4: Initialize $match = true$

Step 5: For i from 0 to $(m - 1)$: // Compare pattern with text

Step 6: If $T[s + i] \neq P[i]$: // Mismatch found

Step 7: Set $match = false$

Step 8: Break the inner loop.

Step 9: If $match == true$:

Step 10: Report occurrence of pattern at shift s

PROGRAM CODE:

```
#include <stdio.h>

#include <string.h>

void search(char* pat, char* txt) {

    int M = strlen(pat);

    int N = strlen(txt);

    // A loop to slide pat[] one by one

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

        int j;

        // For current index i, check for pattern match

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

            if (txt[i + j] != pat[j]) {

                break;

            }

        }

        // If pattern matches at index i

        if (j == M) {

            printf("Pattern found at index %d\n", i);

        }

    }

}

int main() {

    // Example 1

    char txt1[] = "AABAACAADAABAABA";

    char pat1[] = "AABA";

    printf("Example 1:\n");

    search(pat1, txt1);

}
```

```
// Example 2

char txt2[] = "sneha";

char pat2[] = "ha";

printf("\nExample 2:\n");

search(pat2, txt2);

return 0;

}
```

OUTPUT:

```
Example 1:
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12

Example 2:
Pattern found at index 3

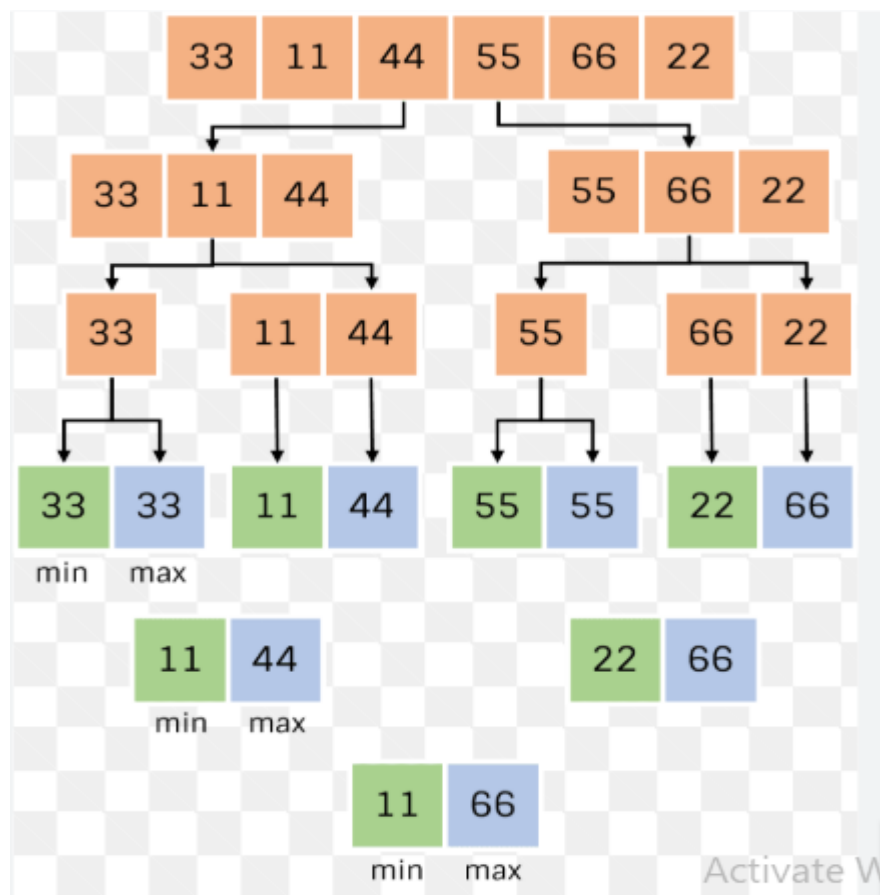
=== Code Execution Successful ===
```

EXPERIMENT NO. 9

AIM: Implementation Min-Max Algorithm

THEORY:

- The **Min-Max Algorithm** (or **Minimax**) is a decision-making algorithm commonly used in **two-player turn-based games** like Tic-Tac-Toe, Chess, or Checkers.
- The **Minimax algorithm** is a recursive algorithm used to **find the optimal move** in **two-player zero-sum games**, where one player's gain is another player's loss.
- **Algorithm:**
 - Step1:** Recursive divide-and-conquer splits array
 - Step2:** Base cases: 1 or 2 elements (direct comparison)
 - Step3:** Recursively compute min/max of subarrays
 - Step4:** Combine sub results to get global min and max
- **Example:**



PROGRAM CODE:

```
#include <stdio.h>

typedef struct {
    int min;
    int max;
} MinMax;

MinMax getMinMax(int arr[], int low, int high) {
    MinMax result, left, right;

    // Only one element
    if (low == high) {
        result.min = arr[low];
        result.max = arr[low];
        return result;
    }

    // Two elements
    if (high == low + 1) {
        if (arr[low] > arr[high]) {
            result.max = arr[low];
            result.min = arr[high];
        } else {
            result.max = arr[high];
            result.min = arr[low];
        }
        return result;
    }

    // More than 2 elements
    int mid = (low + high) / 2;
```

```

    left = getMinMax(arr, low, mid);

    right = getMinMax(arr, mid + 1, high);

    result.min = (left.min < right.min) ? left.min : right.min;

    result.max = (left.max > right.max) ? left.max : right.max;

    return result;
}

int main()
{
    int arr[] = {3, 5, 1, 9, 2, 8, -4, 7};

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

    MinMax result = getMinMax(arr, 0, n - 1);

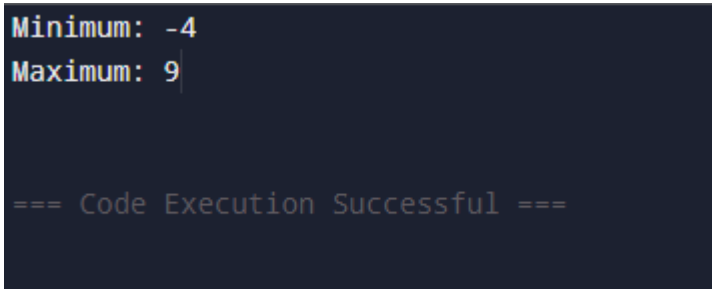
    printf("Minimum: %d\n", result.min);

    printf("Maximum: %d\n", result.max);

    return 0;
}

```

OUTPUT:



```

Minimum: -4
Maximum: 9

=== Code Execution Successful ===

```

EXPERIMENT NO. 10

AIM: To Implement Single Source Shortest Path Bellman Ford.

THEORY:

- The **Bellman-Ford algorithm** is used to compute the **shortest paths from a single source vertex** to all other vertices in a weighted graph.
- Unlike Dijkstra's algorithm, **Bellman-Ford can handle graphs with negative weight edges**.
- The Bellman-Ford algorithm finds the shortest paths from a single source vertex to all other vertices in a weighted graph, even with negative edge weights.
- It works by iteratively relaxing edges, meaning it checks if going through a particular edge provides a shorter path to a destination vertex and updates the shortest path if needed.
- The algorithm repeats this process $V-1$ times, where V is the number of vertices, to ensure all possible shortest paths are considered. Finally, it performs one more relaxation step to detect negative cycles, which would cause infinite loops in shortest path calculations.

PROGRAM CODE:

```
#include <stdio.h>
#include <stdlib.h>
#define INF 99999 // Representation of infinity
// Structure to represent an edge
typedef struct {
    int src, dest, weight;
} Edge;
// Function to implement Bellman-Ford
void bellmanFord(int vertices, int edges, Edge edgeList[], int source) {
    int dist[vertices];
    // Step 1: Initialize distances
    for (int i = 0; i < vertices; i++)
```

```

    dist[i] = INF;
dist[source] = 0;

// Step 2: Relax all edges V-1 times
for (int i = 1; i <= vertices - 1; i++) {
    for (int j = 0; j < edges; j++) {
        int u = edgeList[j].src;
        int v = edgeList[j].dest;
        int w = edgeList[j].weight;
        if (dist[u] != INF && dist[u] + w < dist[v])
            dist[v] = dist[u] + w;
    }
}

// Step 3: Check for negative-weight cycles
for (int j = 0; j < edges; j++) {
    int u = edgeList[j].src;
    int v = edgeList[j].dest;
    int w = edgeList[j].weight;
    if (dist[u] != INF && dist[u] + w < dist[v]) {
        printf("Graph contains a negative weight cycle\n");
        return;
    }
}

// Print the result
printf("Vertex\tDistance from Source %d\n", source);
for (int i = 0; i < vertices; i++)
    printf("%d\t%d\n", i, dist[i]);
}

// Main function
int main() {
    int V = 5; // Number of vertices
    int E = 8; // Number of edges
    // Define all edges: {src, dest, weight}

```



```

Edge edgeList[] = {
    {0, 1, -1},
    {0, 2, 4},
    {1, 2, 3},
    {1, 3, 2},
    {1, 4, 2},
    {3, 2, 5},
    {3, 1, 1},
    {4, 3, -3}
};
int source = 0; // Start from vertex 0
bellmanFord(V, E, edgeList, source);
return 0;
}

```

OUTPUT:

```

Vertex Distance from Source 0
0      0
1      -1
2      2
3      -2
4      1

=== Code Execution Successful ===

```