NATIONAL INSTITUTE OF TECHNOLOGY , RAIPUR

ARTIFICIAL INTELLIGENCE LAB REPORT

NAME : DISHA JAIN

ROLL NO. : 19118902

ROLL NO. IN WORDS : ONE NINE ONE ONE EIGHT NINE ZERO TWO

ENROLMENT NO. : 190840

SUBJECT : ARTIFICIAL INTELLIGENCE

COURSE : B.TECH

BRANCH : INFORMATION TECHNOLOGY (IT)

SEMESTER : 6th

INDEX

Q1

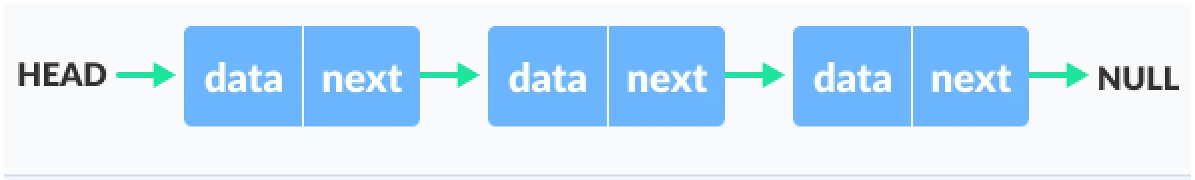
Write a program to find the length of a given list, to find the last element and to delete the first occurrence of a particular element in a given list.

DESCRIPTION:

A linked list is a dynamic data structure.Unlike arrays, We can insert and delete data easily.In linked list, size of data can vary.Linked lists can be of multiple types: singly, doubly, and circular linked list.

A linked List consists of two data items:

- One is data
- Another is a node which points to other nodes in the list



Linked list Data Structure

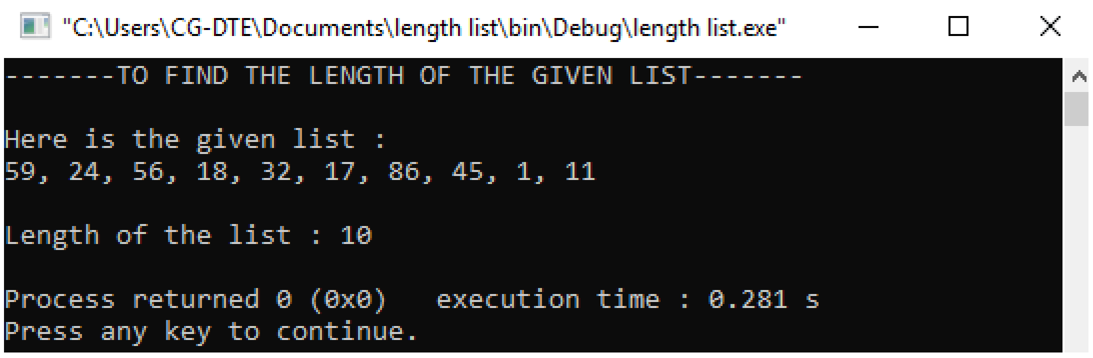
Write a program to find the length of a given list,

CODE:

```
//Write a program to find the length of a given list.

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

//Container class to store all related data and functions
class linkedList
{
            struct Node
            {
                        int data;
                        Node * next;
                        Node(int num);
            };
            Node * head, * last;
        public:
            linkedList();
            bool isEmpty();
        void insertNode(int);
        void printNode();
        int lenOfList();
};

//Node constructor
linkedList::Node::Node(int num)
{
        data = num;
        next = nullptr;
}

//Simple function to check if the list is empty
bool linkedList::isEmpty()
{
      return (head == nullptr);
}

//Constructor for class linkedList
linkedList::linkedList()
{
      last = head = nullptr;
}

//Insert the given number at the end of the list
void linkedList::insertNode(int num)
{
        if (isEmpty())
        {
                head = new Node(num);
                last = head;
        }
        else
        {
                Node * temp = last;
                last = new Node(num);
                temp -> next = last;
        }
}

//Print all the elements
```

```
void linkedList::printNode()
{
        if (isEmpty()) return;
        Node * temp = head;
        while (temp != last)
        {
                cout << temp -> data << ", ";
                temp = temp -> next;
        }
        cout << last -> data << endl;
}

//Return length of list
int linkedList::lenOfList()
{
        Node * temp = head;
        int len = 0;
        while (temp != nullptr)
        {
                len++;
                temp = temp -> next;
        }
        return len;
}

int main()
{
        linkedList L;
        vector < int > tempList = {59,24,56,18,32,17,86,45,1,11};
        //Insert all the elements of the vector into linked list
        for (int num: tempList)
            L.insertNode(num);

        cout<<"-------TO FIND THE LENGTH OF THE GIVEN LIST-------\n\n";

        //Print the just created list;
        cout << "Here is the given list :\n";
        L.printNode();
        //Print the length of the list
        cout << "\nLength of the list : " << L.lenOfList() << endl;
}
```

OUTPUT:


"C:\Users\CG-DTE\Documents\length list\bin\Debug\length list.exe"

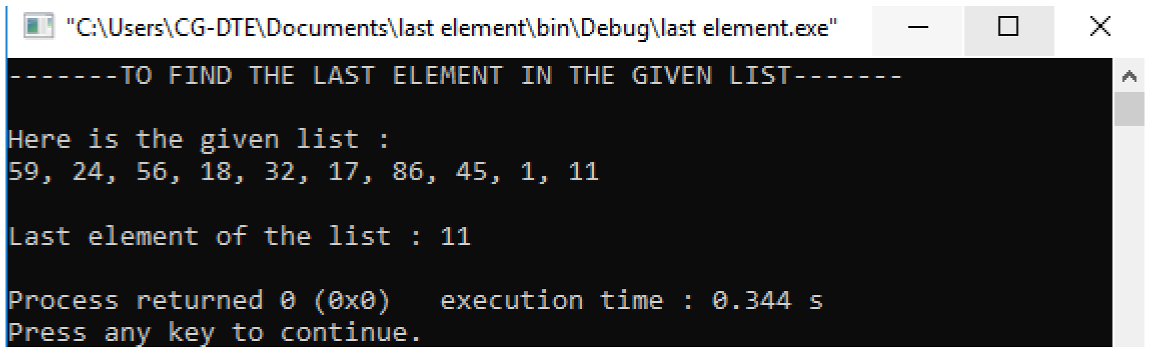
```
-------TO FIND THE LENGTH OF THE GIVEN LIST-------

Here is the given list :
59, 24, 56, 18, 32, 17, 86, 45, 1, 11

Length of the list : 10

Process returned 0 (0x0)   execution time : 0.281 s
Press any key to continue.
```

Write a program to find the last element

CODE:

```
//to find the last element in given list

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

//Container class to store all related data and functions
class linkedList
{
        struct Node
        {
                int data;
                Node * next;
                Node(int num);
        };
        Node * head, * last;
      public:
        linkedList();
        bool isEmpty();
        void insertNode(int);
        void printNode();
        int lastNode();
};

//Node constructor
linkedList::Node::Node(int num)
{
        data = num;
        next = nullptr;
}

//Simple function to check if the list is empty
bool linkedList::isEmpty()
{
      return (head == nullptr);
}

//Constructor for class linkedList
linkedList::linkedList()
{
      last = head = nullptr;
}

//Insert the given number at the end of the list
void linkedList::insertNode(int num)
{
        if (isEmpty())
        {
                head = new Node(num);
                last = head;
        }
        else
        {
                Node * temp = last;
                last = new Node(num);
                temp -> next = last;
        }
}

//Print all the elements
void linkedList::printNode()
{
        if (isEmpty()) return;
        Node * temp = head;
        while (temp != last)
        {
                cout << temp -> data << ", ";
                temp = temp -> next;
        }
        cout << last -> data << endl;
}

//Return last node
int linkedList::lastNode()
{
        Node * temp = head;
        while (temp -> next != nullptr) temp = temp -> next;
        return temp -> data;
}

int main()
{
        linkedList L;
        vector < int > tempList = {59,24,56,18,32,17,86,45,1,11};

        //Insert all the elements of the vector into linked list
        for (int num: tempList)
            L.insertNode(num);

        cout<<"-------TO FIND THE LAST ELEMENT IN THE GIVEN LIST-------\n\n";

        //Print the just created list;
        cout << "Here is the given list :\n";
        L.printNode();

        //Print the last element
        cout << "\nLast element of the list : " << L.lastNode() << endl ;
}
```

OUTPUT:



"C:\Users\CG-DTE\Documents\last element\bin\Debug\last element.exe"

```
-------TO FIND THE LAST ELEMENT IN THE GIVEN LIST-------

Here is the given list :
59, 24, 56, 18, 32, 17, 86, 45, 1, 11

Last element of the list : 11

Process returned 0 (0x0)   execution time : 0.344 s
Press any key to continue.
```

Write a program to delete the first occurrence of a particular element in a given list.

CODE:

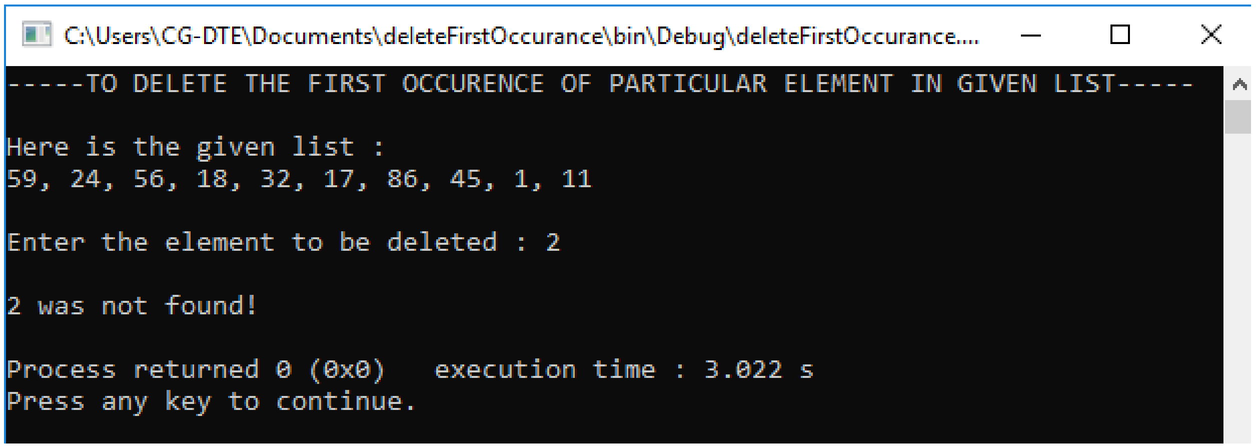
```
// to delete the first  occurrence of a particular element in a given list.
#include<bits/stdc++.h>
using namespace std;

//Container class to store all related data and functions
class linkedList
{
        struct Node
        {
                int data;
                Node * next;
                Node(int num);
        };
        Node * head, * last;
      public:
        linkedList();
        bool isEmpty();
        void insertNode(int);
        bool deleteNode(int);
        void printNode();
};

int main()
{
        linkedList L;
        vector < int > tempList = {59,24,56,18,32,17,86,45,1,11};

        //Insert all the elements of the vector into linked list
        for (int num: tempList) L.insertNode(num);
            cout<<"-----TO DELETE THE FIRST OCCURENCE OF PARTICULAR
      ELEMENT IN GIVEN LIST-----\n\n";

        //Print the just created list;
        cout << "Here is the given list :\n";
        L.printNode();
```

```
            //Get user input to delete first occurance of given element
            int element;
            cout << "\nEnter the element to be deleted : ";
            cin >> element;

            //Delete and print the new list, if possible
            if (L.deleteNode(element))
            {
                        cout << "\nList after deletion of " << element << " : \n";
                        L.printNode();
            }
            else
                  cout << endl << element << " was not found!\n";
            return 0;
}

//Node constructor
linkedList::Node::Node(int num)
{
            data = num;
            next = nullptr;
}

//Simple function to check if the list is empty
bool linkedList::isEmpty()
{
         return (head == nullptr);
}

//Constructor for class linkedList
linkedList::linkedList()
{
         last = head = nullptr;
}

//Insert the given number at the end of the list
void linkedList::insertNode(int num)
{
            if (isEmpty())
            {
                        head = new Node(num);
```

```
                    last = head;
            }
            else
            {
                    Node * temp = last;
                    last = new Node(num);
                    temp -> next = last;
            }
}

//Delete the first occurance of given element and return status
bool linkedList::deleteNode(int num)
{
            if (isEmpty())
                return false;
            Node * garbage;
            if (head -> data == num)
            {
                    garbage = head;
                    head = head -> next;
                    delete garbage;
                    return true;
            }
            if (head -> next != nullptr)
            {
                    Node * currNode = head, * nextNode = head -> next;
                    while (nextNode != nullptr)
                    {
                            if (nextNode -> data == num)
                            {
                                    garbage = nextNode;
                                    currNode -> next = nextNode -> next;
                                    delete garbage;
                                    return true;
                            }
                            currNode = nextNode;
                            nextNode = currNode -> next;
                    }
            }
            return false;
}
```

```
//Print all the elements
void linkedList::printNode()
{
        if (isEmpty()) return;
        Node * temp = head;
        while (temp != last)
        {
                cout << temp -> data << ", ";
                temp = temp -> next;
        }
        cout << last -> data << endl;
}
```

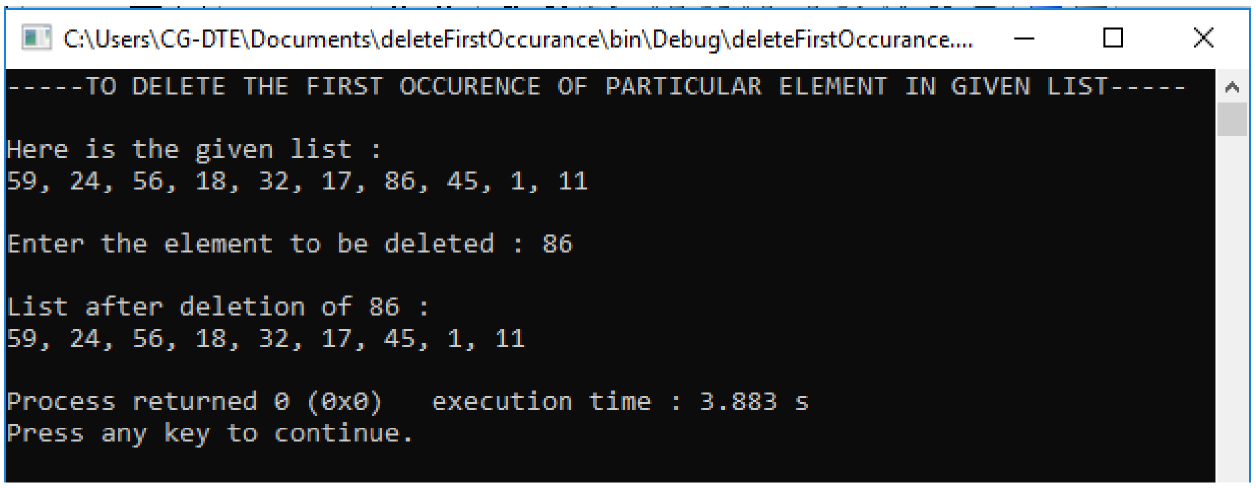
OUTPUT1:



C:\Users\CG-DTE\Documents\deleteFirstOccurance\bin\Debug\deleteFirstOccurance....

```
-----TO DELETE THE FIRST OCCURENCE OF PARTICULAR ELEMENT IN GIVEN LIST-----

Here is the given list :
59, 24, 56, 18, 32, 17, 86, 45, 1, 11

Enter the element to be deleted : 2

2 was not found!

Process returned 0 (0x0)   execution time : 3.022 s
Press any key to continue.
```

OUTPUT 2:



C:\Users\CG-DTE\Documents\deleteFirstOccurance\bin\Debug\deleteFirstOccurance....

```
-----TO DELETE THE FIRST OCCURENCE OF PARTICULAR ELEMENT IN GIVEN LIST-----

Here is the given list :
59, 24, 56, 18, 32, 17, 86, 45, 1, 11

Enter the element to be deleted : 86

List after deletion of 86 :
59, 24, 56, 18, 32, 17, 45, 1, 11

Process returned 0 (0x0)   execution time : 3.883 s
Press any key to continue.
```

Q2.
Implement BFS algorithm for finding the goal node form the tree.

DESCRIPTION:

Breadth first search is a general technique of traversing a graph. Breadth first search may use more memory but will always find the shortest path first. In this type of search the state space is represented in form of a tree. The solution is obtained by traversing through the tree. The nodes of the tree represent the start value or starting state, various intermediate states and the final state. In this search a queue data structure is used and it is level by level traversal. Breadth first search expands nodes in order of their distance from the root. It is a path finding algorithm that is capable of always finding the solution if one exists. The solution which is found is always the optional solution. This task is completed in a very memory intensive manner. Each node in the search tree is expanded in a breadth wise at each level.

CONCEPT :

- Step 1: Traverse the root node
- Step 2: Traverse all neighbours of root node.
- Step 3: Traverse all neighbours of neighbours of the root node.
- Step 4: This process will continue until we are getting the goal node.

ALGORITHM :

- Step 1: Place the root node inside the queue.
- Step 2: If the queue is empty then stops and return failure.
- Step 3: If the FRONT node of the queue is a goal node then stop and return success.
- Step 4: Remove the FRONT node from the queue. Process it and find all its neighbours that are in ready state then place them inside the queue in any order.
- Step 5: Go to Step 3.
- Step 6: Exit.

COMPLEXITY :

- The time complexity of BFS is **O(V + E)**, where V is the number of nodes and E is the number of edges.
- The space complexity of BFS can be expressed as **O(V)**, where V is the number of vertices.

PSEUDOCODE :

```
BFS (G, s)                    //Where G is the graph and s is the source node
```

```
let Q be queue.
Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

mark s as visited.
while ( Q is not empty)
    //Removing that vertex from queue,whose neighbour will be visited now
    v  =  Q.dequeue( )

   //processing all the neighbours of v
   for all neighbours w of v in Graph G
       if w is not visited
              Q.enqueue( w )          //Stores w in Q to further visit its neighbour
              mark w as visited.
```

BFS ALGORITHM APPLICATIONS :

1. To build index by search index
2. For GPS navigation
3. Path finding algorithms
4. In Ford-Fulkerson algorithm to find maximum flow in a network
5. Cycle detection in an undirected graph
6. In minimum spanning tree

ADVANTAGES :

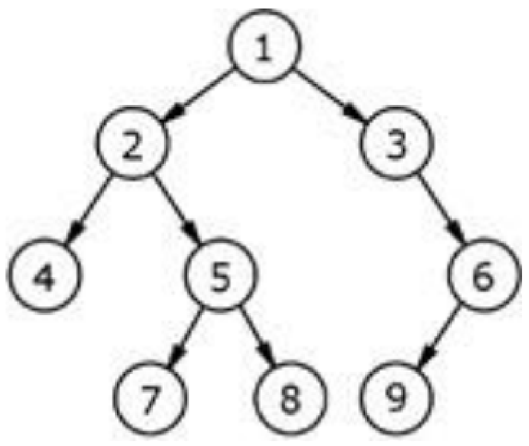
- In this procedure at any way it will find the goal.
- It does not follow a single unfruitful path for a long time.
- It finds the minimal solution in case of multiple paths.

DISADVANTAGES :

- BFS consumes large memory space.
- Its time complexity is more.
- It has long pathways, when all paths to a destination are on approximately the same search depth.

IMPLEMENTATION OF BFS ALGORITHM:

Now, let's see the implementation of BFS algorithm in C++.

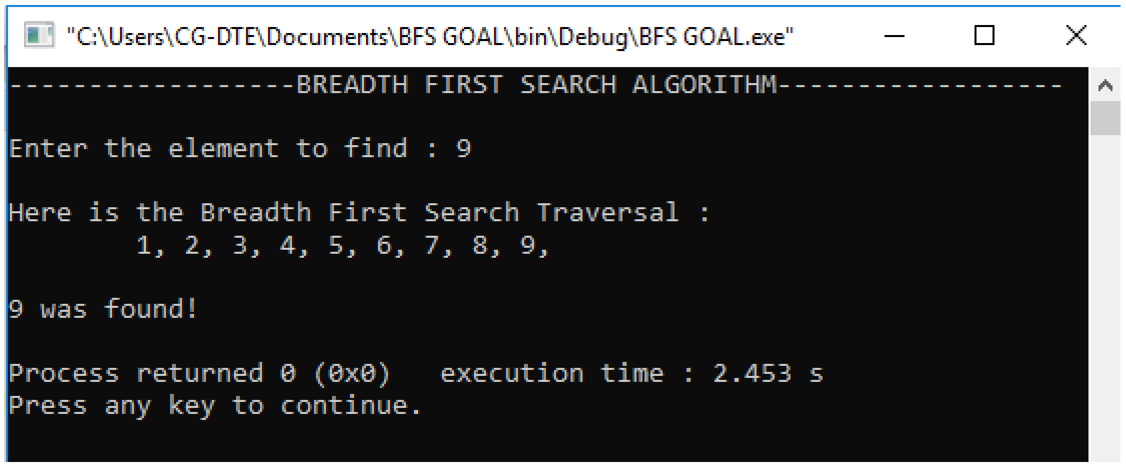


CODE:

```
//Implement BFS algorithm for finding the goal node form the tree.

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

//Defining node structure
struct Node
{
		int data;
		Node * left, * right;
		Node(int num)
		{
				data = num;
				left = right = nullptr;
		}
}* root;

//Creating the tree
void createTree()
{
		root = new Node(1);
		root -> left = new Node(2);
		root -> right = new Node(3);
		root -> left -> left = new Node(4);
		root -> left -> right = new Node(5);
		root -> right -> right = new Node(6);
		root -> left -> right -> left = new Node(7);
		root -> left -> right -> right = new Node(8);
		root -> right -> right -> left = new Node(9);
}

//Doing a breadth first search to find given element
bool BFS(int num)
{
		if (root == nullptr) return false;
		bool flag = false;
		queue < Node * > cache;
		cache.push(root);
		while (not cache.empty())
		{
				Node * currNode = cache.front();
				cout << currNode -> data << ", ";
				cache.pop();
				if (currNode -> data == num) flag = true;
				if (currNode -> left != nullptr) cache.push(currNode -> left);
				if (currNode -> right != nullptr) cache.push(currNode -> right);
		}
		return flag;
}

int main()
{
		int num;
		cout<<"------------------BREADTH FIRST SEARCH ALGORITHM------------------";

		//Get user input to find element
		cout << "\n\nEnter the element to find : ";
		cin >> num;

		//Create the tree
		createTree();

		//Traverse and find the element
		cout << "\nHere is the Breadth First Search Traversal :\n\t";

		if (BFS(num))
		{
		    cout << "\n\n" << num << " was found!\n";
		}
```

```
            else
            {
                  cout << "\n\n" << num << " was not found!\n";
            }

            return 0;
}
```

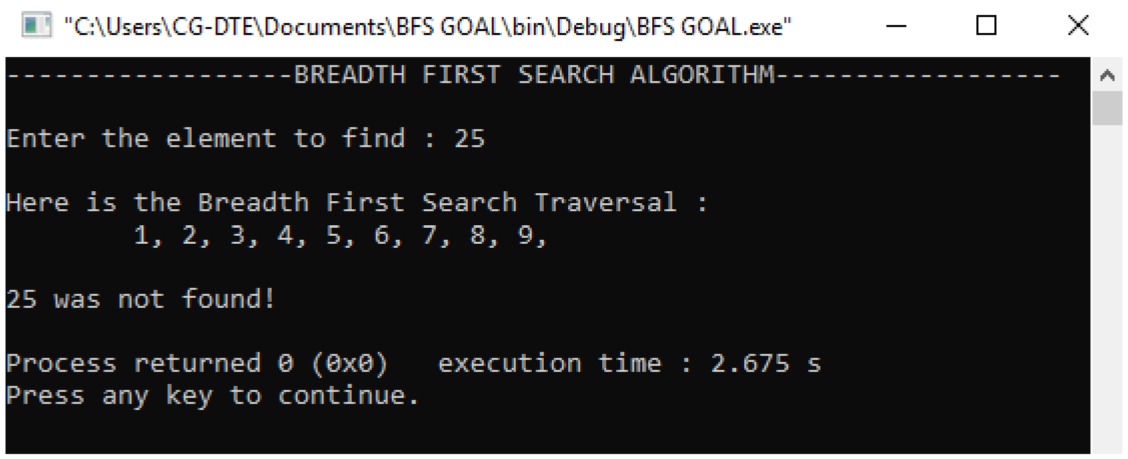
OUTPUT 1:



"C:\Users\CG-DTE\Documents\BFS GOAL\bin\Debug\BFS GOAL.exe"

```
------------------BREADTH FIRST SEARCH ALGORITHM------------------

Enter the element to find : 9

Here is the Breadth First Search Traversal :
        1, 2, 3, 4, 5, 6, 7, 8, 9,

9 was found!

Process returned 0 (0x0)   execution time : 2.453 s
Press any key to continue.
```

OUTPUT 2:



"C:\Users\CG-DTE\Documents\BFS GOAL\bin\Debug\BFS GOAL.exe"

```
------------------BREADTH FIRST SEARCH ALGORITHM------------------

Enter the element to find : 25

Here is the Breadth First Search Traversal :
        1, 2, 3, 4, 5, 6, 7, 8, 9,

25 was not found!

Process returned 0 (0x0)   execution time : 2.675 s
Press any key to continue.
```

Q3.
Implement DFS algorithm for finding the goal node form the tree

DESCRIPTION:
DFS is also an important type of uniform search. DFS visits all the vertices in the graph. This type of algorithm always chooses to go deeper into the graph. After DFS visited all the reachable vertices from a particular sources vertices it chooses one of the remaining undiscovered vertices and continues the search. DFS reminds the space limitation of breath first search by always generating next a child of the deepest unexpanded nodded. The data structure stack or last in first out (LIFO) is used for DFS. One interesting property of DFS is that, the discover and finish time of each vertex from a parenthesis structure. If we use one open parenthesis when a vertex is finished then the result is properly nested set of parenthesis.

CONCEPT :
- Step 1: Traverse the root node.
- Step 2: Traverse any neighbour of the root node.
- Step 3: Traverse any neighbour of neighbour of the root node.
- Step 4: This process will continue until we are getting the goal node.

ALGORITHM :
- Step 1: PUSH the starting node into the stack.
- Step 2: If the stack is empty then stop and return failure.
- Step 3: If the top node of the stack is the goal node, then stop and return success.
- Step 4: Else POP the top node from the stack and process it. Find all its neighbours that are in ready state and PUSH them into the stack in any order.
- Step 5: Go to step 3.
- Step 6: Exit.

COMPLEXITY OF DEPTH-FIRST SEARCH ALGORITHM :
- The time complexity of the DFS algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.
- The space complexity of the DFS algorithm is **O(V)**.

PSEUDOCODE :
```
DFS-iterative (G, s):                          //Where G is graph and s is source vertex
  let S be stack
  S.push( s )            //Inserting s in stack
  mark s as visited.
  while ( S is not empty):
      //Pop a vertex from stack to visit next
      v  =  S.top( )
     S.pop( )
     //Push all the neighbours of v in stack that are not visited
    for all neighbours w of v in Graph G:
        if w is not visited :
                S.push( w )
               mark w as visited

DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```

APPLICATION OF DFS ALGORITHM :

1. For finding the path
2. To test if the graph is bipartite
3. For finding the strongly connected components of a graph
4. For detecting cycles in a graph

ADVANTAGES :

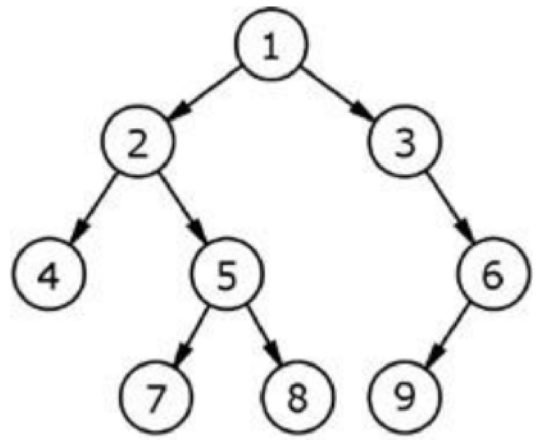
- DFS consumes very less memory space.
- It will reach at the goal node in a less time period than BFS if it traverses in a right path.
- It may find a solution without examining much of search because we may get the desired solution in the very first go.

DISADVANTAGES :

- It is possible that may states keep reoccurring.
- There is no guarantee of finding the goal node.
- Sometimes the states may also enter into infinite loops

IMPLEMENTATION OF DFS ALGORITHM :

Now, let's see the implementation of DFS algorithm in C++.

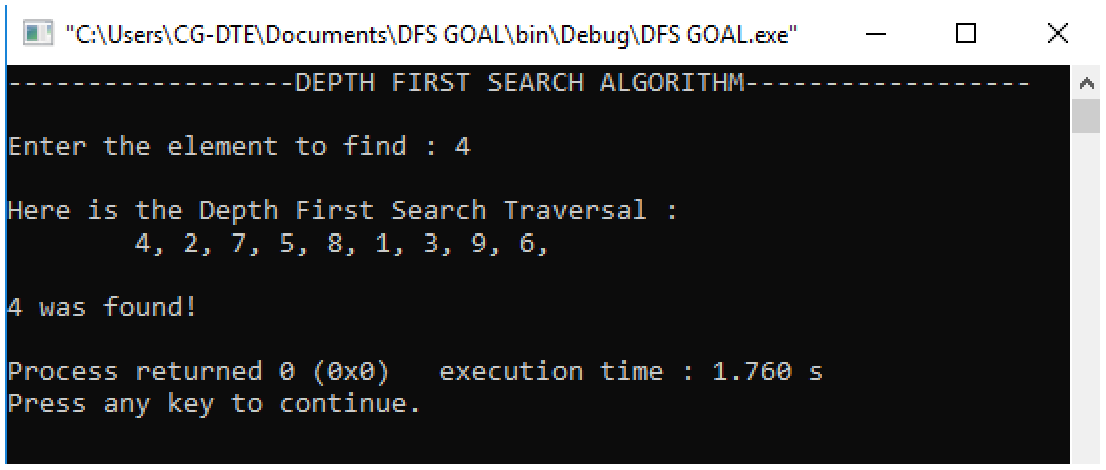


CODE:

```
//Q3 Implement DFS algorithm for finding the goal node form the tree
#include<bits/stdc++.h>
using namespace std;

//Defining node structure
struct Node
{
		int data;
		Node * left, * right;
		Node(int num)
		{
			data = num;
			left = right = nullptr;
		}
} * root;

//Creating the tree
void createTree()
{
		root = new Node(1);
		root->left = new Node(2);
		root->right = new Node(3);
		root->left->left = new Node(4);
		root->left->right = new Node(5);
		root->right->right = new Node(6);
		root->left->right->left = new Node(7);
```

```
        root->left->right->right = new Node(8);
        root->right->right->left = new Node(9);
}
bool flag = false;

//Doing a depth first search to find given element
void DFS(int num, Node * root)
{
        if (root == nullptr) return;
        DFS(num, root->left);
        cout << root->data << ", ";
        if (root->data == num) flag = true;
        DFS(num, root->right);
}

int main()
{
        int num;
        cout<<"------------------DEPTH FIRST SEARCH ALGORITHM------------------";

        //Get user input to find element
        cout << "\n\nEnter the element to find : ";
        cin >> num;

        //Create the tree
        createTree();

        //Traverse and find the element
        cout << "\nHere is the Depth First Search Traversal :\n\t";
        DFS(num, root);

        if (flag) cout << "\n\n" << num << " was found!\n";
        else cout << "\n\n" << num << " was not found!\n";
        return 0;
}
```

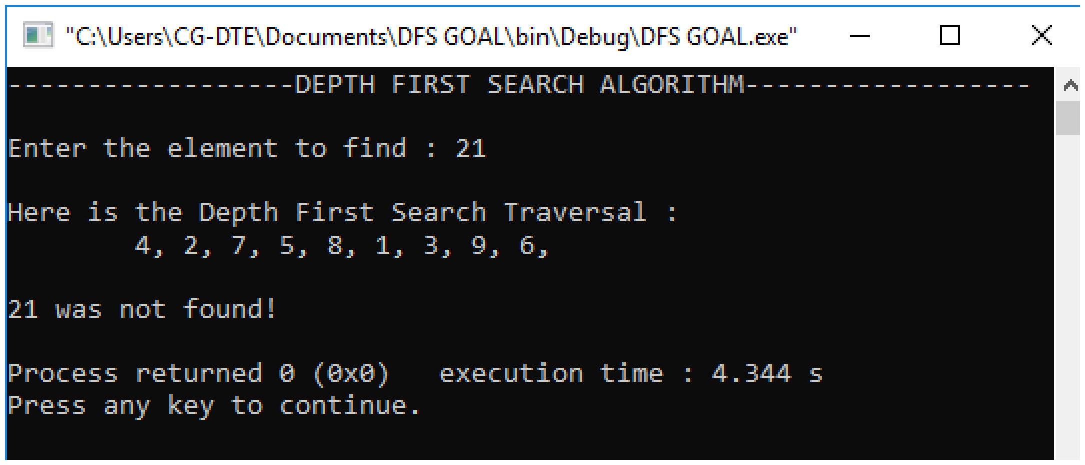
OUTPUT 1:



"C:\Users\CG-DTE\Documents\DFS GOAL\bin\Debug\DFS GOAL.exe"

```
------------------DEPTH FIRST SEARCH ALGORITHM------------------

Enter the element to find : 4

Here is the Depth First Search Traversal :
        4, 2, 7, 5, 8, 1, 3, 9, 6,

4 was found!

Process returned 0 (0x0)   execution time : 1.760 s
Press any key to continue.
```

OUTPUT2 :



"C:\Users\CG-DTE\Documents\DFS GOAL\bin\Debug\DFS GOAL.exe"

```
------------------DEPTH FIRST SEARCH ALGORITHM------------------

Enter the element to find : 21

Here is the Depth First Search Traversal :
        4, 2, 7, 5, 8, 1, 3, 9, 6,

21 was not found!

Process returned 0 (0x0)   execution time : 4.344 s
Press any key to continue.
```

Q4

Solve the 8-puzzle problem by any Search algorithm.

(Start Node)

1	2	3
8	5	6
4	7	

(Goal Node)

1	2	3
4	5	6
7	8	

DESCRIPTION :
“It has set off a 3x3 board having 9 block spaces out of which 8 blocks having tiles bearing number from 1 to 8. One space is left blank. The tile adjacent to blank space can move into it. We have to arrange the tiles in a sequence for getting the goal state”.

PROCEDURE :
The 8-puzzle problem belongs to the category of “sliding block puzzle” type of problem. The 8-puzzle is a square tray in which eight square tiles are placed. The remaining ninth square is uncovered. Each tile in the tray has a number on it. A tile that is adjacent to blank space can be slide into that space. The game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around. The control mechanisms for an 8-puzzle solver must keep track of the order in which operations are performed, so that the operations can be undone one at a time if necessary. The objective of the puzzles is to find a sequence of tile movements that leads from a starting configuration to a goal configuration .

BRANCH AND BOUND :
The search for an answer node can often be speeded by using an “intelligent” ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an answer node. It is similar to the backtracking technique but uses a BFS-like search.
There are basically three types of nodes involved in Branch and Bound

1. **Live node** is a node that has been generated but whose children have not yet been generated.
2. **E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
3. **Dead node** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Cost function:

Each node X in the search tree is associated with a cost. The cost function is useful for determining the next E-node. The next E-node is the one with the least cost. The cost function is defined as

C(X) = g(X) + h(X) where
g(X) = cost of reaching the current node
from the root
h(X) = cost of reaching an answer node from X.

The ideal Cost function for an 8-puzzle Algorithm :

We assume that moving one tile in any direction will have a 1 unit cost. Keeping that in mind, we define a cost function for the 8-puzzle algorithm as below:

c(x) = f(x) + h(x) where
f(x) is the length of the path from root to x
(the number of moves so far) and
h(x) is the number of non-blank tiles not in
their goal position (the number of mis-
-placed tiles). There are at least h(x)
moves to transform state x to a goal state

ALGORITHM :

```
/* Algorithm LCSearch uses c(x) to find an answer node
 * LCSearch uses Least() and Add() to maintain the list of live nodes
 * Least() finds a live node with least c(x), deletes it from the list and returns it
 * Add(x) adds x to the list of live nodes
 * Implement list of live nodes as a min-heap */

struct list_node
{
            list_node *next;

            // Helps in tracing path when answer is found
            list_node *parent;
            float cost;
}

algorithm LCSearch(list_node *t)
{
            // Search t for an answer node
            // Input: Root node of tree t
            // Output: Path from answer node to root
```

```
    if (*t is an answer node)
    {
            print(*t);
            Return;
    }

    E = t; // E-node

    Initialize the list of live nodes to be empty;
    while (true)
    {
            for each child x of E
            {
                    if x is an answer node
                    {
                            print the path from x to t;
                            return;
                    }
                    Add (x); // Add x to list of live nodes;
                    x->parent = E; // Pointer for path to root
            }

            if there are no more live nodes
            {
                    print ("No answer node");
                    return;
            }

            // Find a live node with least estimated cost
            E = Least();

            // The found node is deleted from the list of
            // live nodes
    }
}
```

COMMENTS:

- This problem requires a lot of space for saving the different trays.
- Time complexity is more than that of other problems.
- The user has to be very careful about the shifting of tiles in the trays.
- Very complex puzzle games can be solved by this technique.

IMPLEMENTAION OF 8-PUZZLE PROBLEM:

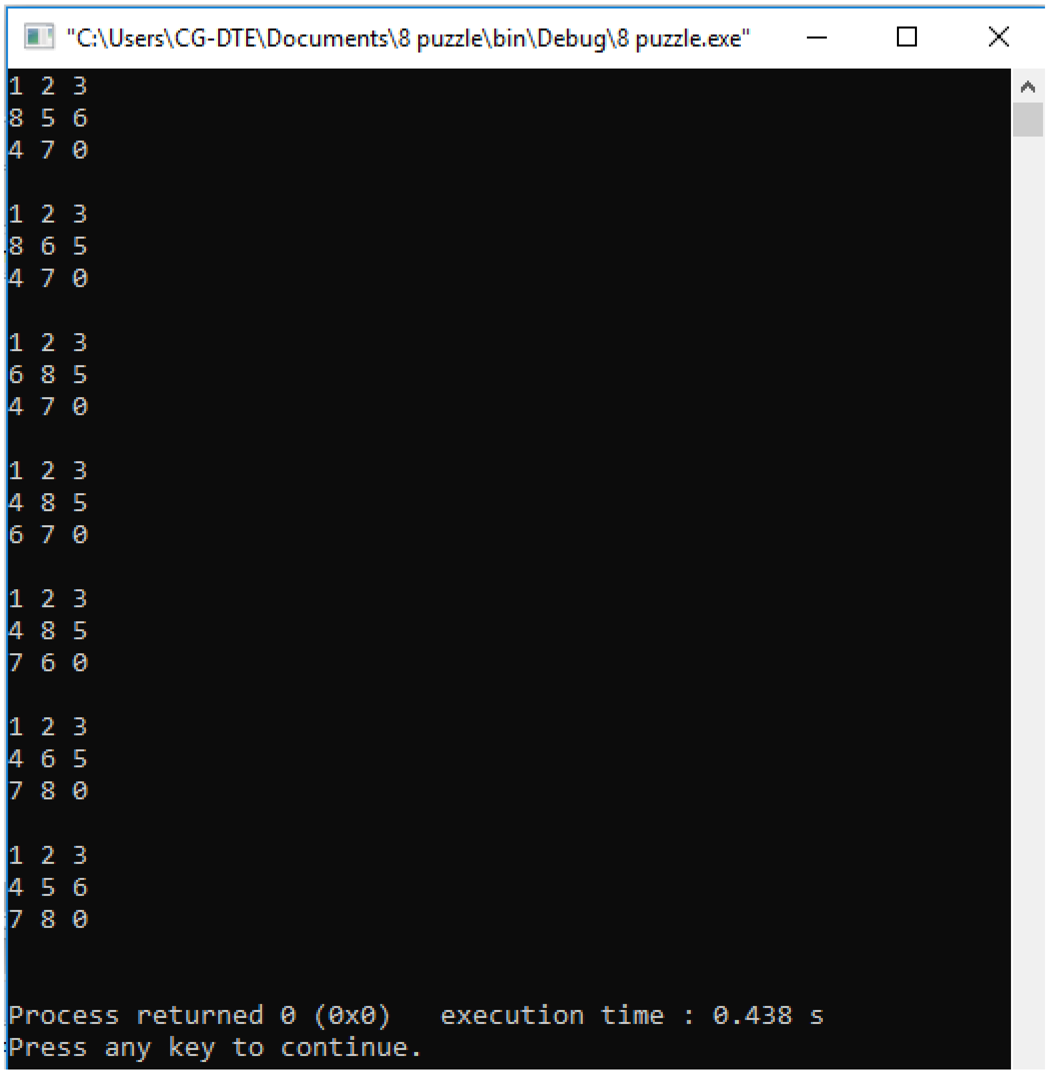
CODE:

```
//4. Solve the 8-puzzle problem by any Search algorithm.
// Program to print path from root node to destination node
// for N*N -1 puzzle algorithm using Branch and Bound
// The solution assumes that instance of puzzle is solvable
#include <bits/stdc++.h>
using namespace std;
#define N 3

// state space tree nodes
struct Node
{
	// stores the parent node of the current node
	// helps in tracing path when the answer is found
	Node* parent;

	// stores matrix
	int mat[N][N];

	// stores blank tile coordinates
	int x, y;

	// stores the number of misplaced tiles
	int cost;

	// stores the number of moves so far
	int level;
};

// Function to print N x N matrix
int printMatrix(int mat[N][N])
{
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < N; j++)
		    printf("%d ", mat[i][j]);
		printf("\n");
	}
}

// Function to allocate a new node
```

```
Node* newNode(int mat[N][N], int x, int y, int newX, int newY, int level, Node* parent)
{
            Node* node = new Node;
            // set pointer for path to root
            node->parent = parent;

            // copy data from parent node to current node
            memcpy(node->mat, mat, sizeof node->mat);

            // move tile by 1 position
            swap(node->mat[x][y], node->mat[newX][newY]);

            // set number of misplaced tiles
            node->cost = INT_MAX;

            // set number of moves so far
            node->level = level;

            // update new blank tile coordinates
            node->x = newX;
            node->y = newY;
            return node;
}

// bottom, left, top, right
int row[] = { 1, 0, -1, 0 };
int col[] = { 0, -1, 0, 1 };

// Function to calculate the number of misplaced tiles
// ie. number of non-blank tiles not in their goal position
int calculateCost(int initial[N][N], int final[N][N])
{
            int count = 0;
            for (int i = 0; i < N; i++) for (int j = 0; j < N; j++)
                  if (initial[i][j] && initial[i][j] != final[i][j]) count++;
            return count;
}

// Function to check if (x, y) is a valid matrix coordinate
int isSafe(int x, int y)
{
         return (x >= 0 && x < N && y >= 0 && y < N);
}
```

```
// print path from root node to destination node
void printPath(Node* root)
{
        if (root == NULL)
          return;
        printPath(root->parent);
        printMatrix(root->mat);
        printf("\n");
}

// Comparison object to be used to order the heap
struct comp
{
        bool operator()(const Node* lhs, const Node* rhs) const
        {
            return (lhs->cost + lhs->level) > (rhs->cost + rhs->level);
        }
};

// Function to solve N*N - 1 puzzle algorithm using Branch and Bound.
// x and y are blank tile coordinates in initial state
void solve(int initial[N][N], int x, int y, int final[N][N])
{
        // Create a priority queue to store live nodes of search tree;
        priority_queue<Node*, std::vector<Node*>, comp> pq;

        // create a root node and calculate its cost
        Node* root = newNode(initial, x, y, x, y, 0, NULL);
        root->cost = calculateCost(initial, final);

        // Add root to list of live nodes;
        pq.push(root);

        // Finds a live node with least cost, add its childrens to list of live nodes and
        // finally deletes it from the list.
        while (!pq.empty())
        {
                // Find a live node with least estimated cost
                Node* min = pq.top();
                // The found node is deleted from the list of  live nodes
                pq.pop();

                // if min is an answer node
                if (min->cost == 0)
                {
                        // print the path from root to destination;
                        printPath(min);
                        return;
                }

                // do for each child of min max 4 children for a node
                for (int i = 0; i < 4; i++)
                {
                        if (isSafe(min->x + row[i], min->y + col[i]))
                        {
                                // create a child node and calculate its cost
                                Node* child = newNode(min->mat, min->x,
                                min->y, min->x + row[i], min->y + col[i],
                                              min->level + 1, min);
                                child->cost = calculateCost(child->mat, final);

                                // Add child to list of live nodes
                                pq.push(child);
                        }
                }
        }
}

int main()
{
        // Initial configuration
        // Value 0 is used for empty space
        int initial[N][N] =
        {
                {1, 2, 3},
                {8, 5, 6},
                {4, 7, 0}
        };

        // Solvable Final configuration
        // Value 0 is used for empty space
        int final[N][N] =
        {
                {1, 2, 3},
                {4, 5, 6},
                {7, 8, 0}
        };

        // Blank tile coordinates in initial configuration
        int x = 1, y = 2;
        solve(initial, x, y, final);
        return 0;
}
```

OUTPUT:



"C:\Users\CG-DTE\Documents\8 puzzle\bin\Debug\8 puzzle.exe"

```
1  2  3
8  5  6
4  7  0

1  2  3
8  6  5
4  7  0

1  2  3
6  8  5
4  7  0

1  2  3
4  8  5
6  7  0

1  2  3
4  8  5
7  6  0

1  2  3
4  6  5
7  8  0

1  2  3
4  5  6
7  8  0


Process returned 0 (0x0)   execution time : 0.438 s
Press any key to continue.
```

Q5.

Solve the classical water jug problem, given 4 litre, 3 litre jugs. Neither have any measuring markers on it. There is tap that can be used to fill the jugs with water. How can you get exactly 2 litre of water into 4 litre jug. Use any search of algorithm.

DESCRIPTION:
Some jugs are given which should have non-calibrated properties. At least any one of the jugs should have filled with water. Then the process through which we can divide the whole water into different jugs according to the question can be called as water jug problem.

PROCEDURE:
Suppose that you are given 3 jugs A,B,C with capacities 8,5 and 3 liters respectively but are not calibrated (i.e. no measuring mark will be there). Jug A is filled with 8 liters of water. By a series of pouring back and forth among the 3 jugs, divide the 8 liters into 2 equal parts i.e. 4 liters in jug A and 4 liters in jug B. How?
In this problem, the start state is that the jug A will contain 8 liters water whereas jug B and jug C will be empty. The production rules involve filling a jug with some amount of water, taking from the jug A. The search will be finding the sequence of production rules which transform the initial state to final state. The state space for this problem can be described by set of ordered pairs of three variables (A, B, C) where variable A represents the 8 liter jug, variable B represents the 5 liter and variable C represents the 3 liters jug respectively.

You are given an m liter jug and a n liter jug. Both the jugs are initially empty. The jugs don't have markings to allow measuring smaller quantities. You have to use the jugs to measure d liters of water where d is less than n.
(X, Y) corresponds to a state where X refers to the amount of water in Jug1 and Y refers to the amount of water in Jug2
Determine the path from the initial state (xi, yi) to the final state (xf, yf), where (xi, yi) is (0, 0) which indicates both Jugs are initially empty and (xf, yf) indicates a state which could be (0, d) or (d, 0).

The **operations** you can perform are:

1. Empty a Jug, (X, Y)->(0, Y) Empty Jug 1
2. Fill a Jug, (0, 0)->(X, 0) Fill Jug 1
3. Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

Here, we keep exploring all the different valid cases of the states of water in the jug simultaneously until and unless we reach the required target water.

At any given state we can do either of the following operations:

1. **Fill** a jug
2. **Empty** a jug
3. **Transfer** water from one jug to another until either of them gets completely filled or empty.

COMPLEXITY:

- Time Complexity: **O(n*m)**.
- Space Complexity: **O(n*m)** Where n and m are the quantity of jug1 and jug2 respectively.

COMMENTS:

- This problem takes a lot of time to find the goal state.
- This process of searching in this problem is very lengthy.
- At each step of the problem the user have to strictly follow the production rules. Otherwise the problem may go to infinity step.

IMPLEMENTATION:

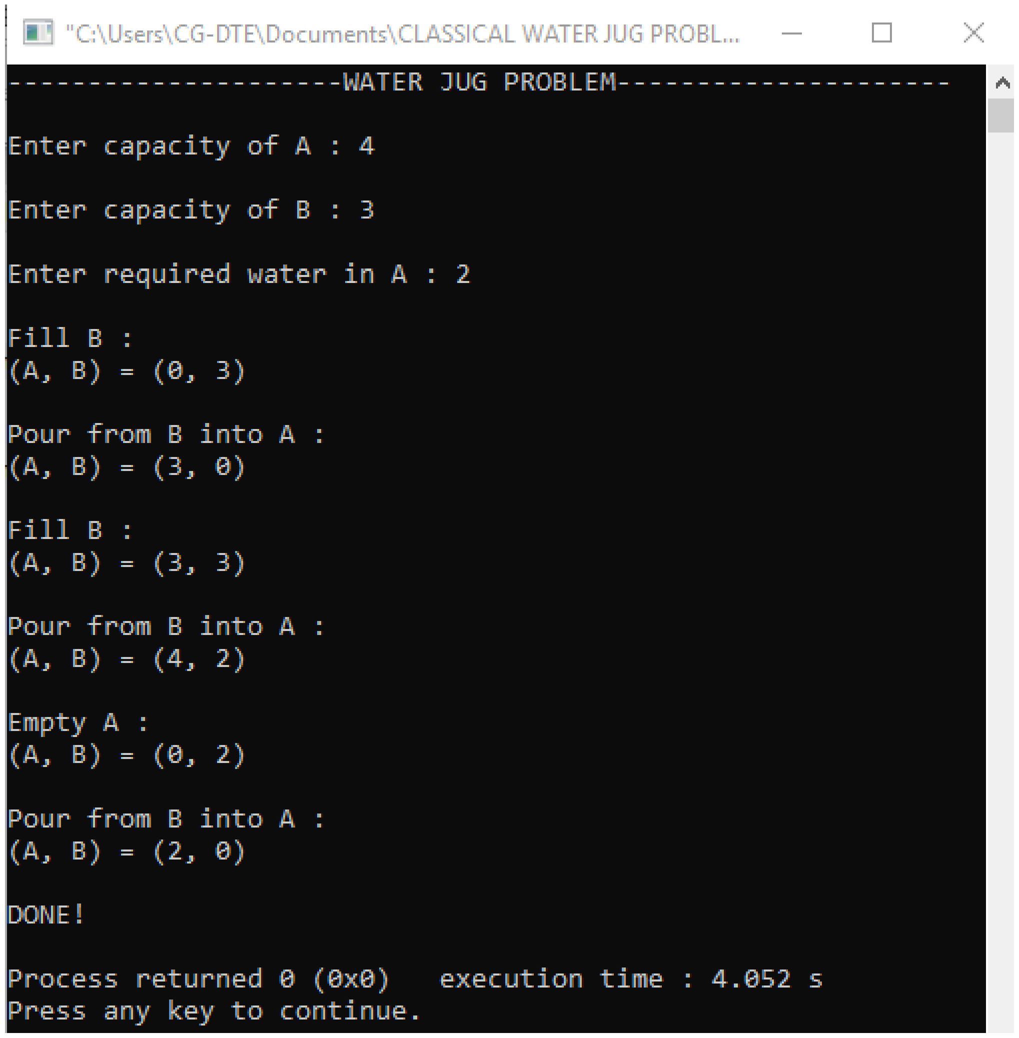
CODE:

```
/* Solve the classical water jug problem, given 4 litre, 3 litre jugs.Neither have any measuring  markers on it. There is tap that can be used to fill the jugs with water. How can you get exactly 2 litre of water into 4 litrejug. Use any search of algorithm.*/

#include <bits/stdc++.h>
using namespace std;
class Jug
{
            int capacity;
            int value;
        public:
            Jug(int n)
            {
                        capacity = n;
                        value = 0;
            }
            void Fill()
            {
                    value = capacity;
            }
            void Empty()
            {
                     value = 0;
            }
            bool isFull()
            {
                     return value >= capacity;
            }
            bool isEmpty()
            {
                     return value == 0;
            }
            //A[B] -> Transfer contents of B to A until A is full
            void operator[](Jug &B)
            {
                        int old_value = value;
                        value = value + B.value;
                        value = value > capacity?capacity:value;
                        B.value = B.value - (value - old_value);
            }
            int getValue()
            {
                        return value;
            }
};
int gcd(int n,int m)
{
            if(m<=n && n%m == 0)
               return m;
            if(n < m)
               return gcd(m,n);
            else
               return gcd(m,n%m);
}
bool check(int a,int b,int c)
{
            //boundary cases
            if(c>a)
            {
                        cout << "A can't hold more water than it's capacity!\n";
                        return false;
```

```
        }
        //if c is multiple of HCF of a and b, then possible
        if(c % gcd(a,b) == 0)
        {
                return true;
        }
        //if c is not multiple of HCF of a and b, then not possible
        cout << "Can't reach this state with the given jugs\n";
        return false;
}
void solve(Jug A, Jug B, int result)
{
        while(A.getValue() != result)
        {
                if(!A.isFull() && B.isEmpty())
                {
                        cout << "Fill B :\n";
                        B.Fill();
                        cout << "(A, B) = (" << A.getValue() << ", " << B.getValue()
                        << ")\n\n";
                }
                if(A.isFull())
                {
                        cout << "Empty A :\n";
                        A.Empty();
                        cout << "(A, B) = (" << A.getValue() << ", " << B.getValue()
                         << ")\n\n";
                }
                cout << "Pour from B into A :\n";
                A[B];
                cout << "(A, B) = (" << A.getValue() << ", " << B.getValue() << ")\n\n";
        }
}

int main()
{
        int a, b, result;
        cout<<"---------------------WATER JUG PROBLEM---------------------\n\n";
        cout << "Enter capacity of A : ";
        cin >> a;
```

```
            cout << "\nEnter capacity of B : ";
            cin >> b;
            do
            {
                        cout << "\nEnter required water in A : ";
                        cin >> result;
            }
            while(!check(a,b,result));
            Jug A(a), B(b);
            cout << endl;
            solve(A, B, result);
            cout << "DONE!"<<endl;
            return 0;
}
```

OUTPUT:



```
"C:\Users\CG-DTE\Documents\CLASSICAL WATER JUG PROBL...
---------------------WATER JUG PROBLEM---------------------

Enter capacity of A : 4

Enter capacity of B : 3

Enter required water in A : 2

Fill B :
(A, B) = (0, 3)

Pour from B into A :
(A, B) = (3, 0)

Fill B :
(A, B) = (3, 3)

Pour from B into A :
(A, B) = (4, 2)

Empty A :
(A, B) = (0, 2)

Pour from B into A :
(A, B) = (2, 0)

DONE!

Process returned 0 (0x0)   execution time : 4.052 s
Press any key to continue.
```

Q6.
Solve the classical Monkey Banana problem of AI.

DESCRIPTION:
A hungry monkey finds himself in a room in which A bunch of bananas is hanging from the ceiling.The monkey ,unfortunately,cannot reach the bananas .However,in the room there are also a chair and a stick.the ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick.the monkey knows how to move around, carry other things aroumnd ,reach for the bananas,and wave a stick in the air.What is the best sequence of actions for the monkey to take to acquire lunch?

PROCEDURE:
The solution of the problem is of course that the monkey must push the box under the bananas, then stand on the box and grab the bananas. But the solution procedure requires a lot of planning algorithms. The purpose of the problem is to raise the question: Are monkeys intelligent? Both humans and monkeys have the ability to use mental maps to remember things like where to go to find shelter or how to avoid danger. They can also remember where to go to gather food and water, as well as how to communicate with each other. Monkeys have the ability not only to remember how to hunt and gather but they also have the ability to learn new things, as is the case with the monkey and the bananas. Even though that monkey may never have entered that room before or had only a box for a tool to gather the food available, that monkey can learn that it needs to move the box across the floor, position it below the bananas and climb the box to reach for them.
Initially, the monkey is at location ‘A’, the banana is at location ‘B’ and the box is at location ‘C’. The monkey and box have height “low”; but if the monkey climbs onto the box will have height “High”, the same as the bananas.

The **action** available to the monkey include:
- **“GO”** from one place to another.
- **“PUSH”** an object from one place to another.
- **“Climb”** onto an object.
- **“Grasp”** an object.

Grasping results in holding the object if the monkey and the object are in the same place at the same height.

PROLOG:
Let us see how we can solve this using Prolog. We have some predicates that will move from one state to another state, by performing action.
- When the block is at the middle, and monkey is on top of the block, and monkey

does not have the banana (i.e. ***has not*** state), then using the ***grasp*** action, it will change from ***has not*** state to ***have*** state.
- From the floor, it can move to the top of the block (i.e. ***on top*** state), by performing the action ***climb***.
- The **push** or **drag** operation moves the block from one place to another.
- Monkey can move from one place to another using **walk** or **move** clauses.

Another predicate will be canget(). Here we pass a state, so this will perform move predicate from one state to another using different actions, then perform canget() on state 2. When we have reached to the state 'has>', this indicates 'has banana'. We will stop the execution.

PROLOG PROGRAM :

```
move(state(middle,onbox,middle,hasnot),
  grasp,
  state(middle,onbox,middle,has)).
move(state(P,onfloor,P,H),
  climb,
  state(P,onbox,P,H)).
move(state(P1,onfloor,P1,H),
  drag(P1,P2),
  state(P2,onfloor,P2,H)).
move(state(P1,onfloor,B,H),
  walk(P1,P2),
  state(P2,onfloor,B,H)).
canget(state(_,_,_,has)).
canget(State1) :-
  move(State1,_,State2),
  canget(State2).
```

COMMENTS:
- One major application of the monkey banana problem is the toy problem of computer science.
- One of the specialized purposes of the problem is to raise the question: Are monkeys intelligent?
- This problem is very useful in logic programming and planning

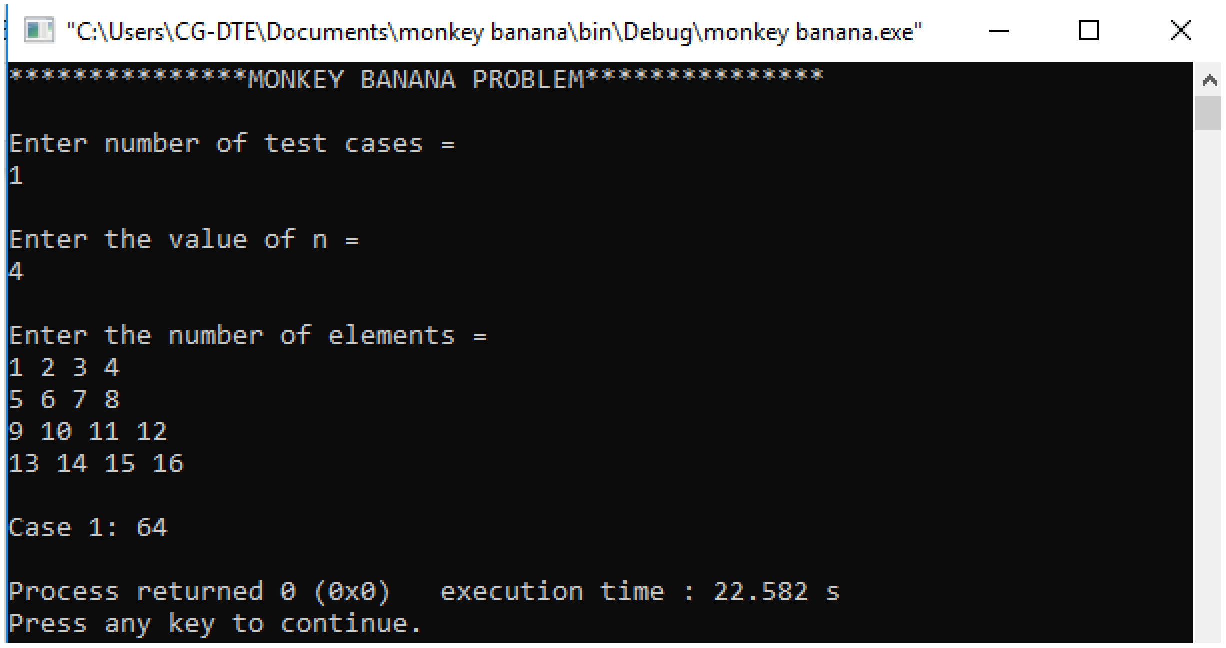
IMPLEMENTATION:

CODE:

```
#include <iostream>
#include <string.h>
#include <fstream>
using namespace std;
int main()
{
        int t;
        int jmax;
        int n;
        unsigned long long a[300][300];
        unsigned long long inp[300][300];
        n= 4;
        cout<<"***************MONKEY BANANA PROBLEM***************";
        cout<<"\n\nEnter number of test cases =\n";
        cin >> t;
        for (int k = 1; k <= t; k++)
        {
                cout<<"\nEnter the value of n =\n";
                cin >> n;
                memset(a, 0, sizeof(a));
                memset(inp, 0, sizeof(inp));
                cout<<"\nEnter the number of elements =\n";
                for (int i = 0; i <= ((2 * n) - 1); i++)
                {
                        if(i < n)
                          jmax = i;
                        else
                          jmax = 2 * n - i;
                        for(int j = 0; j < jmax; j++)
                        {
                             cin >> inp[i][j];
                        }
                 }
                for (int i = ((2 * n) - 1); i >= 0; i--)
                {
                        if(i < n)
                        {
                             jmax = i+1;
                        }
                        else
```

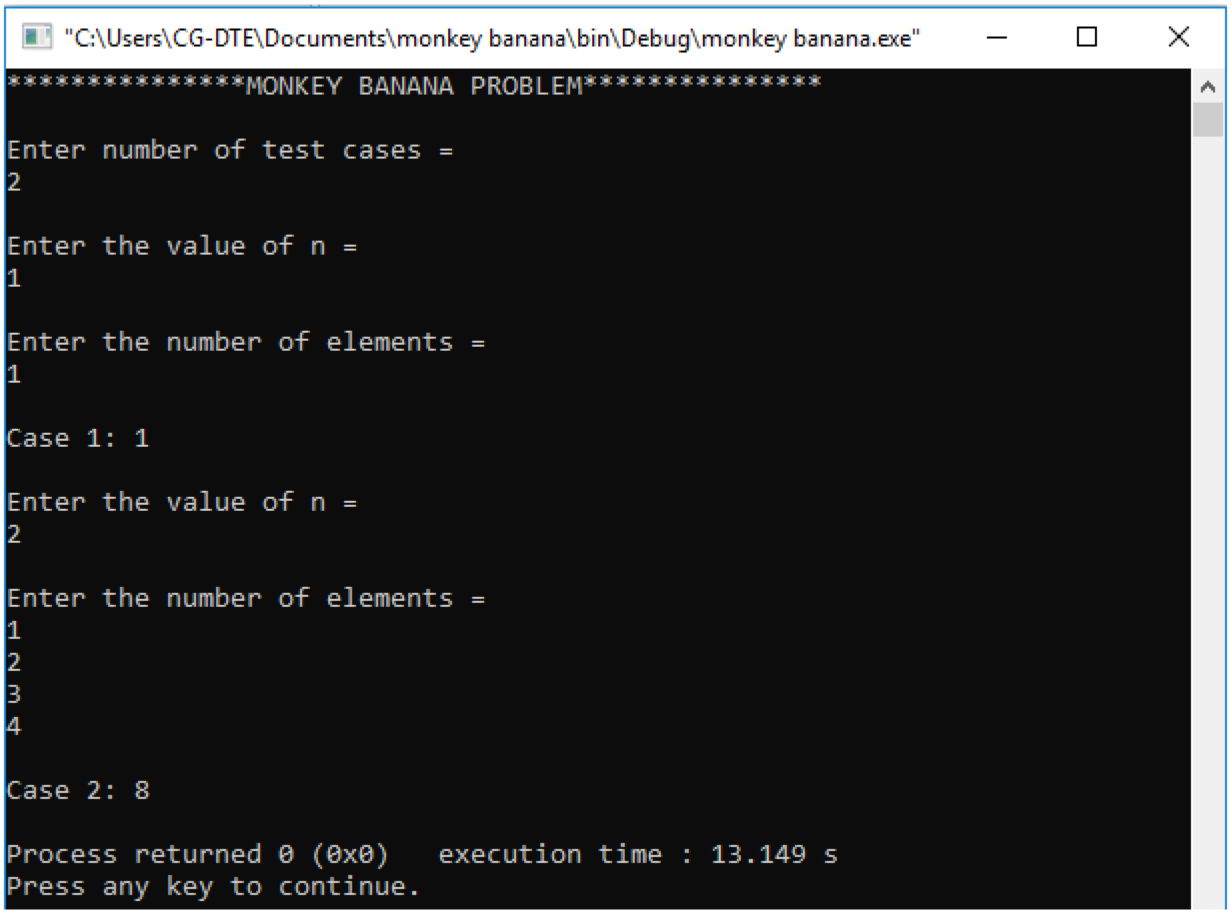
```
                    {
                            jmax = 2 * n - i;
                    }
                    if(i < n)
                    {
                            for(int j = 0; j < jmax; j++)
                            {
                                    if(i == ((2 * n) - 1))
                                    {
                                            a[i][j] =inp[i][j];
                                            continue;
                                    }
                                   a[i][j] = max((inp[i][j] + a[i+1][j]), (inp[i][j] +
                                   a[i+1][j+1]));
                            }
                    }
                    else
                    {
                            for (int j = jmax - 1; j >= 0; j--)
                            {
                                    if(i == ((2 * n) - 1))
                                    {
                                            a[i][j] =inp[i][j];
                                            continue;
                                    }
                                    if(j -1 >= 0)
                                    {
                                           a[i][j]=max((inp[i][j]+a[i+1][j])
                                          ,(inp[i][j]+a[i+1][j-1] ));
                                    }
                                    else
                                    {
                                           a[i][j] = a[i+1][j] + inp[i][j];
                                    }
                             }
                     }
            }
            cout<<"\n";
            cout <<"Case "<<k<<": "<< a[0][0]<<endl;
    }
}
```

OUTPUT 1:


"C:\Users\CG-DTE\Documents\monkey banana\bin\Debug\monkey banana.exe"

```
***************MONKEY BANANA PROBLEM***************

Enter number of test cases =
1

Enter the value of n =
4

Enter the number of elements =
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

Case 1: 64

Process returned 0 (0x0)   execution time : 22.582 s
Press any key to continue.
```

OUTPUT 2:


"C:\Users\CG-DTE\Documents\monkey banana\bin\Debug\monkey banana.exe"

```
***************MONKEY BANANA PROBLEM***************

Enter number of test cases =
2

Enter the value of n =
1

Enter the number of elements =
1

Case 1: 1

Enter the value of n =
2

Enter the number of elements =
1
2
3
4

Case 2: 8

Process returned 0 (0x0)   execution time : 13.149 s
Press any key to continue.
```

Q7.
Implement the Simple hill climbing for finding the goal node.

DESCRIPTION:
Hill climbing search algorithm is simply a loop that continuously moves in the direction of increasing value. It stops when it reaches a "peak" where no neighbour has higher value. This algorithm is considered to be one of the simplest procedures for implementing heuristic search. The hill climbing comes from that idea if you are trying to find the top of the hill and you go up direction from where ever you are. This heuristic combines the advantages of both depth first and breadth first searches into a single method

ALGORITHM:
1. Step 1: Evaluate the initial state. If it is a goal state,then stop and return success.
2. Step 2: Else, continue with the starting state as considering it as a current state.
3. Step 3: Continue Step 4 until a solution is found i.e. until there are no new operators left to be applied in the current state.
4. Step 4:
    a) Select a operator that has not been yet applied to the current state and apply it to produce a new state.
    b) Procedure to evaluate a new state.
        i.If the current state is a goal state, then stop and return success.
        ii.If it is not a goal state but it is better than the current state, then make it current state and proceed further.
        iii. If it is not better than the current state, then continue in the loop until a solution is found.
5. Step 5: Exit.

ADVANTAGES:
- Hill climbing technique is useful in job shop scheduling, automatic programming, circuit designing, and vehicle routing and portfolio management.
- It is also helpful to solve pure optimization problems where the objective is to find the best state according to the objective function.
- It requires much less conditions than other search techniques.

DISADVANTAGES:
The question that remains on hill climbing search is whether this hill is the highest hill possible. Unfortunately without further extensive exploration, this question cannot be answered. This technique works but as it uses local information that's why it can be fooled. The algorithm doesn't maintain a search tree, so the current node data structure

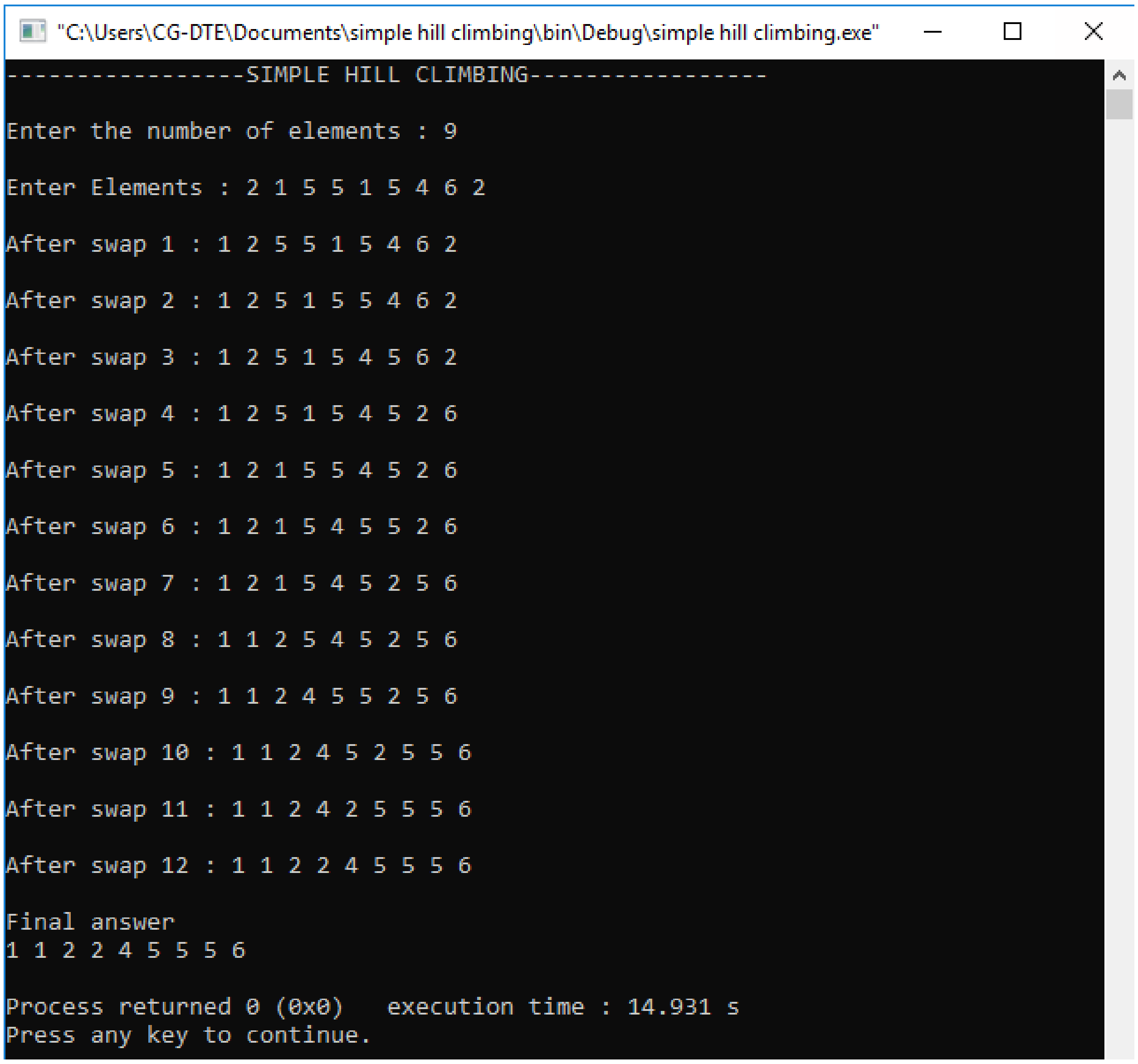
need only record the state and its objective function value. It assumes that local improvement will lead to global improvement.

IMPLEMENTATION:

CODE:

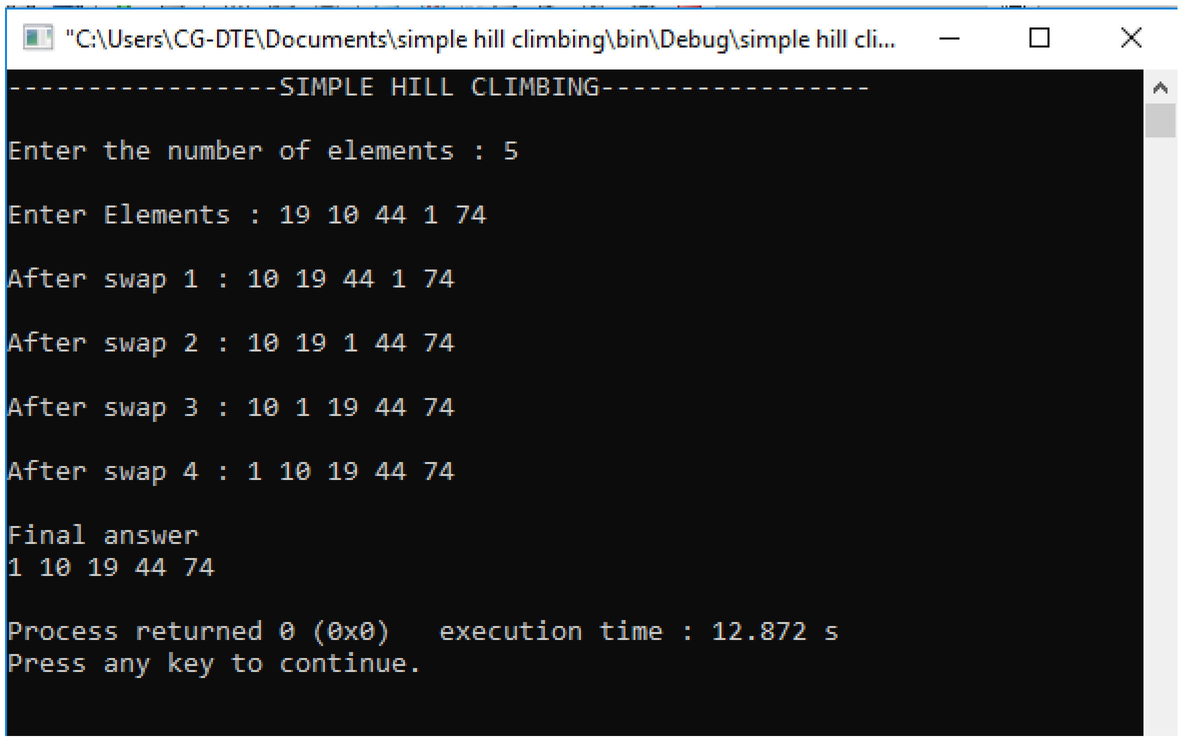
```
//Q7. Implement the Simple hill climbing for finding the goal node.

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

//Calculate final cost
int calcCost(int arr[], int N)
{
        int c = 0;
        for (int i = 0; i < N; i++)
        {
                for (int j = i + 1; j < N; j++)
                {
                        if (arr[j] < arr[i])
                        {
                                c++;
                        }
                }
        }
        return c;
}

//Specific swap function
void swap(int arr[], int i, int j)
{
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
}

//Driver Code
int main()
{
        int N;
        cout<<"-----------------SIMPLE HILL CLIMBING-----------------\n\n";
        cout << "Enter the number of elements : ";
        cin >> N;
        int arr[N];
        cout << "\nEnter Elements : ";
        for (int i = 0; i < N; i++)
        {
            cin >> arr[i];
        }
        cout << endl;
        int bestCost = calcCost(arr, N), newCost, swaps = 0;
        while (bestCost > 0)
        {
                for (int i = 0; i < N - 1; i++)
                {
                        swap(arr, i, i + 1);
                        newCost = calcCost(arr, N);
                        if (bestCost > newCost)
                        {
                                cout << "After swap " << ++swaps << " : ";
                                for (int i = 0; i < N; i++)
                                {
                                    cout << arr[i] << ' ';
                                }
                                cout << "\n\n";
                                bestCost = newCost;
                        }
                        else
                        {
                                 swap(arr, i, i + 1);
                        }
                }
        }
        cout << "Final answer\n";
        for (int i = 0; i < N; i++)
        {
            cout << arr[i] << ' ';
        }
        cout << endl;
        return 0;
}
```

OUTPUT 1:


"C:\Users\CG-DTE\Documents\simple hill climbing\bin\Debug\simple hill climbing.exe"

```
-----------------SIMPLE HILL CLIMBING-----------------

Enter the number of elements : 9

Enter Elements : 2 1 5 5 1 5 4 6 2

After swap 1 : 1 2 5 5 1 5 4 6 2

After swap 2 : 1 2 5 1 5 5 4 6 2

After swap 3 : 1 2 5 1 5 4 5 6 2

After swap 4 : 1 2 5 1 5 4 5 2 6

After swap 5 : 1 2 1 5 5 4 5 2 6

After swap 6 : 1 2 1 5 4 5 5 2 6

After swap 7 : 1 2 1 5 4 5 2 5 6

After swap 8 : 1 1 2 5 4 5 2 5 6

After swap 9 : 1 1 2 4 5 5 2 5 6

After swap 10 : 1 1 2 4 5 2 5 5 6

After swap 11 : 1 1 2 4 2 5 5 5 6

After swap 12 : 1 1 2 2 4 5 5 5 6

Final answer
1 1 2 2 4 5 5 5 6

Process returned 0 (0x0)   execution time : 14.931 s
Press any key to continue.
```

OUTPUT 2:



"C:\Users\CG-DTE\Documents\simple hill climbing\bin\Debug\simple hill cli...

```
-----------------SIMPLE HILL CLIMBING-----------------

Enter the number of elements : 5

Enter Elements : 19 10 44 1 74

After swap 1 : 10 19 44 1 74

After swap 2 : 10 19 1 44 74

After swap 3 : 10 1 19 44 74

After swap 4 : 1 10 19 44 74

Final answer
1 10 19 44 74

Process returned 0 (0x0)   execution time : 12.872 s
Press any key to continue.
```

Q8.
Implement the Steepest hill climbing for finding the goal node.

DESCRIPTION:
A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. This method is called steepest-ascent hill climbing or gradient search. Notice that this contrasts with the basic method in which the first state that is better than the current state is selected. The algorithm works as follows.

ALGORITHM: STEEPEST- HILL CLIMBING

```
1. Evaluate the initial state. If it is also a goal state, then return it and quit.
   Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or until a complete iteration produces no change to
   current state:
     (a) Let SUCC be a state such that any possible successor of the current
         state will be better than SUCC.
     (b) For each operator that applies to the current state do
         (i) Apply the operator and generate a new state.
         (ii) Evaluate the new state. If it is a goal state, then return it and quit. If
          not, compare it to SUCC. If it is better, then set SUCC to this state. If it
          is  not better, leave SUCC alone.
     (c) If the SUCC is better than current state, then set current state to SUCC.
```

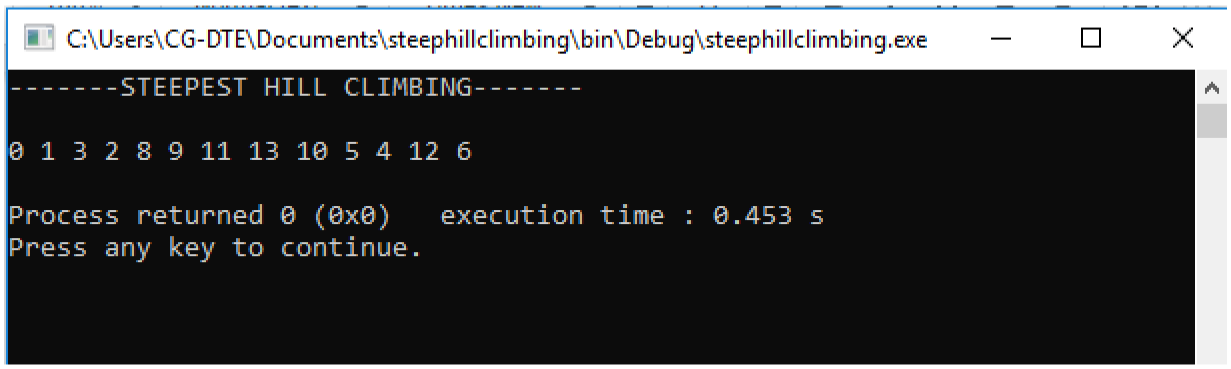
IMPLEMENTATION:

CODE:

```
//8. Implement the Steepest hill climbing for finding the goal node.
#include <bits/stdc++.h>
using namespace std;
typedef pair < int, int > pi;

vector < vector < pi > > graph;

// Function for adding edges to graph
void addedge(int x, int y, int cost)
{
		graph[x].push_back(make_pair(cost, y));
		graph[y].push_back(make_pair(cost, x));
}
```

```
// Gives output path having lowest cost
void steep_hill_climb(int source, int target, int n)
 {
                vector < bool > visited(n, false);

                // MIN HEAP priority queue
                priority_queue < pi, vector < pi > , greater < pi > > pq;

                // sorting in pq gets done by first value of pair
                pq.push(make_pair(0, source));
                int s = source;
                visited[s] = true;
                while (!pq.empty())
                {
                            int x = pq.top().second;
                            // Displaying the path having lowest cost
                            cout << x << " ";
                            pq.pop();
                            if (x == target)
                               break;

                            for (int i = 0; i < graph[x].size(); i++)
                            {
                                        if (!visited[graph[x][i].second])
                                         {
                                                    visited[graph[x][i].second] = true;
                                                    pq.push(make_pair(graph[x][i].first,
                                                    graph[x][i].second));
                                        }
                            }
                }
}

// Driver code to test above methods
int main()
{
        // No. of Nodes
        int v = 14;
        graph.resize(v);

        // The nodes shown in above example(by alphabets) are
        // implemented using integers addedge(x,y,cost);
        addedge(0, 1, 3);
        addedge(0, 2, 6);
        addedge(0, 3, 5);
        addedge(1, 4, 9);
        addedge(1, 5, 8);
        addedge(2, 6, 12);
        addedge(2, 7, 14);
        addedge(3, 8, 7);
        addedge(8, 9, 5);
        addedge(8, 10, 6);
        addedge(9, 11, 1);
        addedge(9, 12, 10);
        addedge(9, 13, 2);

        int source = 0;
        int target = 6;

        cout<<"-------STEEPEST HILL CLIMBING-------\n\n";
        // Function call
        steep_hill_climb(source, target, v);

        cout << endl;

        return 0;
}
```

OUTPUT :



```
C:\Users\CG-DTE\Documents\steephillclimbing\bin\Debug\steephillclimbing.exe
-------STEEPEST HILL CLIMBING-------

0 1 3 2 8 9 11 13 10 5 4 12 6

Process returned 0 (0x0)   execution time : 0.453 s
Press any key to continue.
```

Q9.
Implement the A* algorithm for finding the goal node for OR Graph.

DESCRIPTION:

A* is a cornerstone name of many AI systems and has been used since it was developed in 1968 by Peter Hart; Nils Nilsson and Bertram Raphael. It is the combination of Dijkstra's algorithm and Best first search. It can be used to solve many kinds of problems. A* search finds the shortest path through a search space to goal state using heuristic function. This technique finds minimal cost solutions and is directed to a goal state called A* search. In A*, the * is written for optimality purpose. The A* algorithm also finds the lowest cost path between the start and goal state, where changing from one state to another requires some cost. A* requires heuristic function to evaluate the cost of path that passes through the particular state. This algorithm is complete if the branching factor is finite and every action has fixed cost. A* requires heuristic function to evaluate the cost of path that passes through the particular state.The implementation of A* algorithm is 8-puzzle game

It can be defined by following formula.

f (n)= g (n)+ h(n)

Where

g (n): The actual cost path from the start state to the current state.

h (n): The actual cost path from the current state to goal state.

f (n): The actual cost path from the start state to the goal state.

For the implementation of A* algorithm we will use two arrays namely OPEN and CLOSE.

- **OPEN**: An array which contains the nodes that has been generated but has not been yet examined.
- **CLOSE:** An array which contains the nodes that have been examined.

ALGORITHM:

- Step 1: Place the starting node into OPEN and find its f (n) value.
- Step 2: Remove the node from OPEN, having smallest f (n) value. If it is a goal node then stop and return success.
- Step 3: Else remove the node from OPEN, find all its successors.
- Step 4: Find the f (n) value of all successors; place them into OPEN and place the removed node into CLOSE.
- Step 5: Go to Step-2.
- Step 6: Exit.

ADVANTAGES:

- It is complete and optimal.
- It is the best one from other techniques.
- It is used to solve very complex problems.
- It is optimally efficient, i.e. there is no other optimal algorithm guaranteed to expand fewer nodes than A*.

DISADVANTAGES:

- This algorithm is complete if the branching factor is finite and every action has fixed cost.
- The speed execution of A* search is highly dependant on the accuracy of the heuristic algorithm that is used to compute h (n).
- It has complexity problems.

IMPLEMENTATION:

CODE:

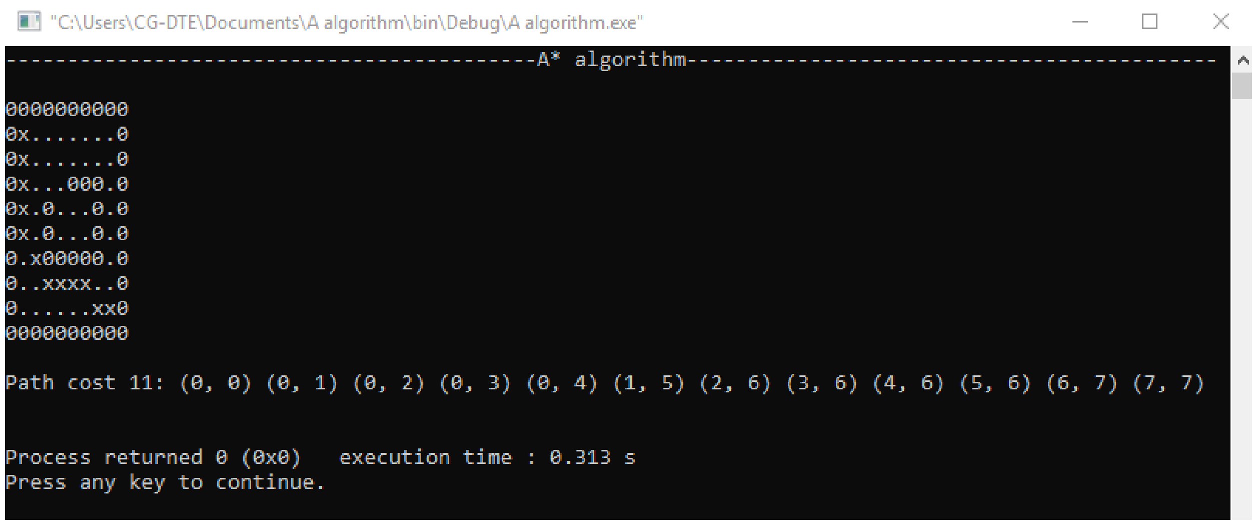
```
//9. Implement the A* algorithm for finding the goal node for OR Graph.

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

class point
{
        public:
           point( int a = 0, int b = 0 )
           {
                       x = a;
                       y = b;
           }
           bool operator ==( const point& o )
           {
                 return o.x == x && o.y == y;
           }
           point operator +( const point& o )
           {
                 return point( o.x + x, o.y + y );
           } int x, y;
```

```
};

class map
{
        public:
           map()
           {
                       char t[8][8] =
                       {
                                   {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0},
                                   {0, 0, 0, 0, 1, 1, 1, 0}, {0, 0, 1, 0, 0, 0, 1, 0},
                                   {0, 0, 1, 0, 0, 0, 1, 0}, {0, 0, 1, 1, 1, 1, 1, 0},
                                   {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0}
                       };
                       w = h = 8;
                       for( int r = 0; r < h; r++ )
                          for( int s = 0; s < w; s++ )
                             m[s][r] = t[r][s];
           }

           int operator() ( int x, int y )
           {
                return m[x][y];
           }
           char m[8][8];
           int w, h;
};

class node
{
        public:
           bool operator == (const node& o )
           {
                return pos == o.pos;
           }
           bool operator == (const point& o )
           {
                return pos == o;
           }
           bool operator < (const node& o )
           {
```

```
                        return dist + cost < o.dist + o.cost;
                }
                point pos, parent;
                int dist, cost;
};

class aStar
{
            public:
                aStar()
                {
                                neighbours[0] = point( -1, -1 );
                                neighbours[1] = point( 1, -1 );
                                neighbours[2] = point( -1, 1 );
                                neighbours[3] = point( 1, 1 );
                                neighbours[4] = point( 0, -1 );
                                neighbours[5] = point( -1, 0 );
                                neighbours[6] = point( 0, 1 );
                                neighbours[7] = point( 1, 0 );
                }

                int calcDist( point& p )
                {
                                int x = end.x - p.x, y = end.y - p.y;
                                return( x * x + y * y );
                }

                bool isValid( point& p )
                {
                               return ( p.x >-1 && p.y > -1 && p.x < m.w && p.y < m.h );
                }

                bool existPoint( point& p, int cost )
                {
                                std::list<node>::iterator i;
                                i = std::find( closed.begin(), closed.end(), p );
                                if( i!= closed.end() )
                                {
                                                if( ( *i ).cost + ( *i ).dist < cost )
                                                        return true;
                                                else
                        {
                                closed.erase( i );
                                return false;
                        }
                }
                i = std::find( open.begin(), open.end(), p );
                if( i!= open.end() )
                {
                        if( ( *i ).cost + ( *i ).dist < cost )
                           return true;
                        else
                        {
                                open.erase( i );
                                return false;
                        }
                }
                return false;
        }

        bool fillOpen( node& n )
        {
                int stepCost, nc, dist;
                point neighbour;
                for( int x = 0; x < 8; x++ )
                {
                        stepCost = x < 4 ? 1 : 1;
                        neighbour = n.pos + neighbours[x];
                        if(neighbour == end )
                           return true;
                       if( isValid( neighbour ) && m( neighbour.x, neighbour.y ) != 1 )
                        {
                                nc= stepCost + n.cost;
                                dist = calcDist( neighbour );
                                if( !existPoint( neighbour, nc + dist ) )
                                {
                                        node m;
                                        m.cost = nc;
                                        m.dist = dist;
                                        m.pos = neighbour;
                                        m.parent = n.pos;
                                        open.push_back( m );
                                }
                        }
                }
                return false;
        }

        bool search( point& s, point& e, map& mp )
        {
                node n;
                end = e;
                start = s;
                m = mp;
                n.cost = 0;
                n.pos = s;
                n.parent = 0;
                n.dist = calcDist( s );
                open.push_back( n );
                while( !open.empty() )
                {
                        //open.sort();
                        node n = open.front();
                        open.pop_front();
                        closed.push_back( n );
                        if( fillOpen( n ) )
                                return true;
                }
                return false;
        }

        int path( std::list<point>& path )
        {
                path.push_front( end );
                int cost = 1 + closed.back().cost;
                path.push_front( closed.back().pos );
                point  parent = closed.back().parent;
                for( std::list<node>::reverse_iterator i = closed.rbegin(); i !=
                closed.rend(); i++ )
                {
                        if( ( *i ).pos == parent && !( ( *i ).pos == start ) )
                        {
                                path.push_front( ( *i ).pos );
```

```
                                        parent = ( *i ).parent;
                                }
                        }
                        path.push_front( start );
                        return cost;
                }
                map m;
                point end, start;
                point neighbours[8];
                std::list<node> open;
                std::list<node> closed;
};

int main( int argc, char* argv[] )
{
                map m;
                point s, e( 7, 7 );
                aStar as;
                std::cout<<"------------------------------------------A*algorithm---------------------
----------------------";
                std::cout<<"\n\n";
                if( as.search( s, e, m ) )
                {
                        std::list<point> path;
                        int c = as.path( path );
                        for( int y = -1; y < 9; y++ )
                        {
                                for( int x = -1; x < 9; x++ )
                                {
                                        if( x < 0 || y < 0 || x > 7 || y > 7 || m( x, y ) == 1 )
                                           std::cout << '0';
                                        else
                                        {

                                              if( std::find( path.begin(), path.end(), point(
                                              x, y ) )!= path.end() )
                                                  std::cout << "x";
                                               else std::cout << ".";
                                        }
                                }
                                std::cout << "\n";
                }
                std::cout << "\nPath cost " << c << ": ";
                for( std::list<point>::iterator i = path.begin(); i != path.end(); i++ )
                {
                        std::cout<< "(" << ( *i ).x << ", " << ( *i ).y << ") ";
                }
        }
        std::cout << "\n\n";
        return 0;
}
```

OUTPUT:



"C:\Users\CG-DTE\Documents\A algorithm\bin\Debug\A algorithm.exe"

```
-----------------------------------------A* algorithm-----------------------------------------

0000000000
0x.......0
0x.......0
0x...000.0
0x.0...0.0
0x.0...0.0
0.x00000.0
0..xxxx..0
0......xx0
0000000000

Path cost 11: (0, 0) (0, 1) (0, 2) (0, 3) (0, 4) (1, 5) (2, 6) (3, 6) (4, 6) (5, 6) (6, 7) (7, 7)


Process returned 0 (0x0)   execution time : 0.313 s
Press any key to continue.
```

Q10.
Solve the classical missionary & cannibals problem by any search algorithm.

DESCRIPTION:
Three missionaries and three cannibals find themselves on one side of a river.they have agreed that they would all like to get to the other side.But the missionaries are not sure what else the cannibals have agreed to.so the missionaries want to manage the trip across the river in such a way that the number of missionaries on either side of the river is never less than the number of cannibals who are on the same side.the only boat available holds only two people at a time.How can everyone get across the river without the missionaries risking being eaten?So for solving the problem and to find out the solution on different states is called the Missionaries and Carnival Problem.

PROCEDURE:
Let us take an example. Initially a boatman, Grass, Tiger and Goat is present at the left bank of the river and want to cross it. The only boat available is one capable of carrying 2 objects of portions at a time. The condition of safe crossing is that at no time the tiger present with goat, the goat present with the grass at the either side of the river. How they will cross the river?
The objective of the solution is to find the sequence of their transfer from one bank of the river to the other using the boat sailing through the river satisfying these constraints.

COMMENTS:
- This problem requires a lot of space for its state implementation.
- It takes a lot of time to search the goal node.
- The production rules at each level of state are very strict.

IMPLEMENTATION:

CODE:

```
//10. Solve the classical missionary &cannibals problem by any search algorithm.
#include<bits/stdc++.h>
using namespace std;

struct StateSTR
{
        int Mlhs; //nr missionaries on LHS    of river
```

```
            int Clhs; //nr cannibals    on LHS    of river
            int pos; //boat on LHS (0) or RHS(1) of river
            int Mrhs; //nr missionaries on RHS    of river
            int Crhs; //nr cannibals    on RHS    of river
            StateSTR * parent; //pointer to parent state
            int opUsed;
            bool operator == (const StateSTR & rhs) const
            {
                        return ((Mlhs == rhs.Mlhs) && (Clhs == rhs.Clhs) &&
                        (Mrhs == rhs.Mrhs) && (Crhs == rhs.Crhs) &&(pos == rhs.pos));
            }
};

ostream & operator << (ostream & out, const StateSTR & s)
{
            out << "Mlhs:" << s.Mlhs << endl;
            out << "Clhs:" << s.Clhs << endl;
            out << "Boat:" << s.pos << endl;
            out << "Mrhs:" << s.Mrhs << endl;
            out << "Crhs:" << s.Crhs << endl << endl;
            return out;
}

bool validState(StateSTR * S)
{
            if ((( * S).Clhs < 0) || (( * S).Clhs > 3)) return false;
            if ((( * S).Crhs < 0) || (( * S).Crhs > 3)) return false;
            if ((( * S).Mlhs < 0) || (( * S).Mlhs > 3)) return false;
            if ((( * S).Mrhs < 0) || (( * S).Mrhs > 3)) return false;
            if ((( * S).pos != 0) && (( * S).pos != 1)) return false;
            if (((( * S).Clhs > ( * S).Mlhs) && (( * S).Mlhs > 0)) ||
                  ((( * S).Crhs > ( * S).Mrhs) && (( * S).Mrhs > 0)))
               return false;
            return true;
}
StateSTR * nextState(StateSTR * Z,const int j)
{
            StateSTR * S = new StateSTR();
            ( * S) = ( * Z);
            ( * S).opUsed = j;
            switch (j)
```

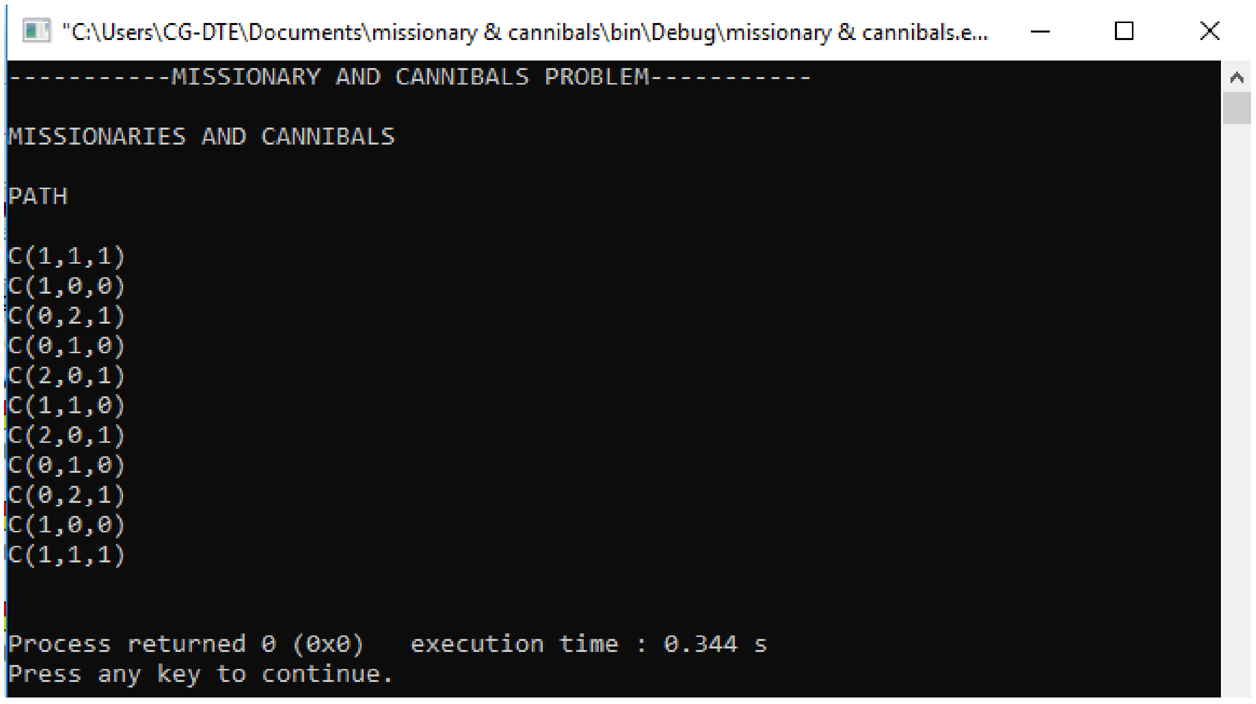
```
{
        case 0:
        {
                ( * S).pos -= 1;
                ( * S).Mlhs += 0;
                ( * S).Clhs += 1;
                ( * S).Mrhs -= 0;
                ( * S).Crhs -= 1;
        }
        break;
        case 1:
        {
                ( * S).pos -= 1;
                ( * S).Mlhs += 0;
                ( * S).Clhs += 2;
                ( * S).Mrhs -= 0;
                ( * S).Crhs -= 2;
        }
        break;
        case 2:
        {
                ( * S).pos -= 1;
                ( * S).Mlhs += 1;
                ( * S).Clhs += 0;
                ( * S).Mrhs -= 1;
                ( * S).Crhs -= 0;
        }
        break;
        case 3:
        {
                ( * S).pos -= 1;
                ( * S).Mlhs += 2;
                ( * S).Clhs += 0;
                ( * S).Mrhs -= 2;
                ( * S).Crhs -= 0;
        }
        break;
        case 4:
        {
                ( * S).pos -= 1;
                ( * S).Mlhs += 1;
```

```
		( * S).Clhs += 1;
		( * S).Mrhs -= 1;
		( * S).Crhs -= 1;
	}
	break;
	case 5:
	{
		( * S).pos += 1;
		( * S).Mrhs += 0;
		( * S).Crhs += 1;
		( * S).Mlhs -= 0;
		( * S).Clhs -= 1;
	}
	break;
	case 6:
	{
		( * S).pos += 1;
		( * S).Mrhs += 0;
		( * S).Crhs += 2;
		( * S).Mlhs -= 0;
		( * S).Clhs -= 2;
	}
	break;
	case 7:
	{
		( * S).pos += 1;
		( * S).Mrhs += 1;
		( * S).Crhs += 0;
		( * S).Mlhs -= 1;
		( * S).Clhs -= 0;
	}
	break;
	case 8:
	{
		( * S).pos += 1;
		( * S).Mrhs += 2;
		( * S).Crhs += 0;
		( * S).Mlhs -= 2;
		( * S).Clhs -= 0;
	}
	break;
```

```
                case 9:
                {
                        ( * S).pos += 1;
                        ( * S).Mrhs += 1;
                        ( * S).Crhs += 1;
                        ( * S).Mlhs -= 1;
                        ( * S).Clhs -= 1;
                }
                 break;
        }
        return S;
}

bool  notFound(StateSTR  *  Y,  list  <  StateSTR  *  >  OPEN,  list  <  StateSTR  *  >
CLOSED)
{
        list < StateSTR * > ::iterator itr1 = OPEN.begin();
        list < StateSTR * > ::iterator itr2 = CLOSED.begin();
        for (; itr1 != OPEN.end(); itr1++)
          if (( * ( * itr1)) == ( * Y)) break;
        for (; itr2 != CLOSED.end(); itr2++)
          if (( * ( * itr2)) == ( * Y)) break;
        if ((itr1 == OPEN.end()) && (itr2 == CLOSED.end()))
          return true;
        return false;
}

void addChildren(list < StateSTR * > & OPEN,list < StateSTR * > & CLOSED,
StateSTR * Y)
{
        StateSTR * tState;
        for (int i = 0; i < 10; i++)
        {
                tState = nextState(Y, i);
                if ((validState(tState)) &&(notFound(tState, OPEN, CLOSED)))
                {
                        ( * tState).parent = Y;
                        OPEN.push_front(tState);
                }
                else
                  delete tState;
        }
        return;
}

void printOP(int n)
{
        switch (n)
        {
              case 0:
                      cout << "C(0,1,0)" << endl;
                      break;
              case 1:
                      cout << "C(0,2,0)" << endl;
                      Break;
              case 2:
                      cout << "C(1,0,0)" << endl;
                      break;
              case 3:
                      cout << "C(2,0,0)" << endl;
                      break;
              case 4:
                      cout << "C(1,1,0)" << endl;
                      break;
              case 5:
                      cout << "C(0,1,1)" << endl;
                      break;
              case 6:
                      cout << "C(0,2,1)" << endl;
                      break;
              case 7:
                      cout << "C(1,0,1)" << endl;
                      break;
              case 8:
                      cout << "C(2,0,1)" << endl;
                      break;
              case 9:
                      cout << "C(1,1,1)" << endl;
                      break;
        }
}
int main()
```

```
{
        cout<<"-----------MISSIONARY AND CANNIBALS PROBLEM-----------\n";
        cout << "\nMISSIONARIES AND CANNIBALS";
        bool searchResult = false;
        stack < int > opsUsed;
        StateSTR START =
        {
          3,
          3,
          0,
          0,
          0,
          NULL,
          -1
        };

        StateSTR GOAL =
        {
          0,
          0,
          1,
          3,
          3,
          NULL
        };

        StateSTR * X;
        StateSTR * tempState;
        list < StateSTR * > OPEN;
        list < StateSTR * > CLOSED;
        OPEN.push_front( & START);

        while (!OPEN.empty())
        {
                X = OPEN.front(); //stack-like operation
                OPEN.pop_front();
                if (( * X) == GOAL)
                {
                        searchResult = true;
                        break;
                }
                else
                {
                                addChildren(OPEN, CLOSED, X);
                                CLOSED.push_back(X);
                }
        }

        //Display results
        if (searchResult == true)
        {
                cout << endl<<endl << "PATH" <<endl<< endl;
                for (StateSTR * p = X; p != NULL; p = ( * p).parent)
                   opsUsed.push(( * p).opUsed);
        }
        while (!opsUsed.empty())
        {
                printOP(opsUsed.top());
                opsUsed.pop();
        }
        cout << endl;
        return 0;
}
```

OUTPUT:



"C:\Users\CG-DTE\Documents\missionary & cannibals\bin\Debug\missionary & cannibals.e...

```
-----------MISSIONARY AND CANNIBALS PROBLEM-----------

MISSIONARIES AND CANNIBALS

PATH

C(1,1,1)
C(1,0,0)
C(0,2,1)
C(0,1,0)
C(2,0,1)
C(1,1,0)
C(2,0,1)
C(0,1,0)
C(0,2,1)
C(1,0,0)
C(1,1,1)


Process returned 0 (0x0)   execution time : 0.344 s
Press any key to continue.
```

Q11.
Solve the classical Travelling Salesman Problem of AI by heuristic approach.

DESCRIPTION:

The traveling salesman problem is a classic problem in combinatorial optimization. This problem is to find the shortest path that a salesman should take to traverse through a list of cities and return to the origin city. The list of cities and the distance between each pair are provided.

TSP is useful in various applications in real life such as planning or logistics. For example, a concert tour manager who wants to schedule a series of performances for the band must determine the shortest path for the tour to ensure reducing traveling costs and not making the band unnecessarily exhausted.

This is an NP-hard problem. In simple words, it means you can not guarantee to find the shortest path within a reasonable time limit. This is not unique to TSP though. In real-world optimization problems, you frequently encounter problems for which you must find sub-optimal solutions instead of optimal ones.

IMPLEMENTATION:

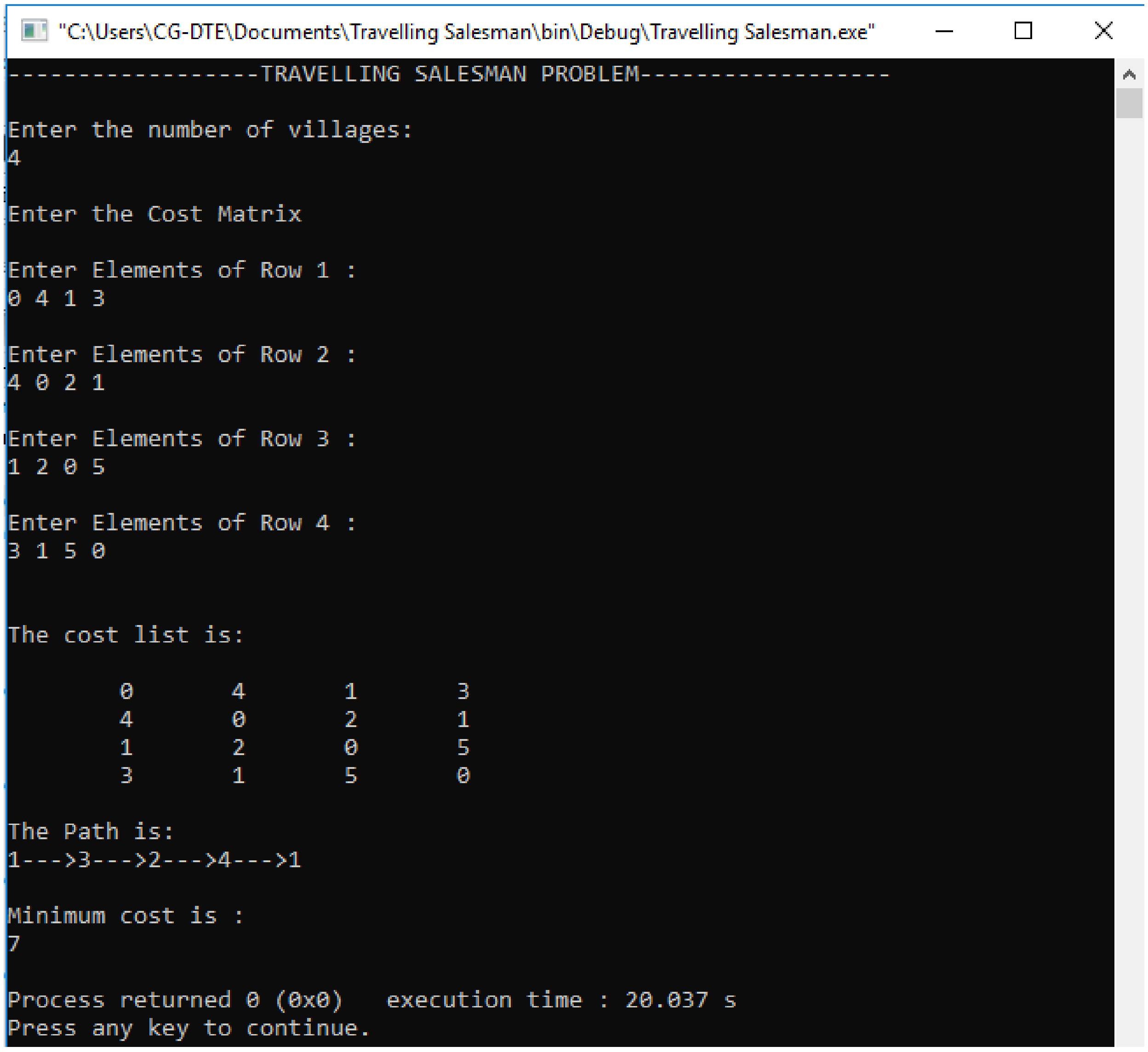
CODE:

```
// Solve the classical Travelling Salesman Problem of AI by heuristic approach.
#include<stdio.h>
#include <iostream>
using namespace std;
int ary[10][10],completed[10],n,cost=0;

void takeInput()
{
          int i,j;
          cout<<"------------------TRAVELLING SALESMAN PROBLEM
          ------------------";
          cout<<"\n\nEnter the number of villages:\n";
          cin>>n;
          cout<<"\nEnter the Cost Matrix\n" ;
          for(i=0; i < n; i++)
          {
                  cout<<"\nEnter Elements of Row "<<i+1<<" :\n";
                  for( j=0; j < n; j++)
                    cin>>ary[i][j];
```

```
                completed[i]=0;
        }
        cout<<"\n\nThe cost list is:\n";
        for( i=0; i < n; i++)
        {
                cout<<"\n";
                for(j=0; j < n; j++)
                    cout<<"\t"<<ary[i][j];
        }
}
int least(int c)
{
        int i,nc=999;
        int min=999,kmin;
        for(i=0; i < n; i++)
        {
                if((ary[c][i]!=0)&&(completed[i]==0))
                    if(ary[c][i]+ary[i][c] < min)
                    {
                            min=ary[i][0]+ary[c][i];
                            kmin=ary[c][i];
                            nc=i;
                    }
        }
        if(min!=999)
            cost+=kmin;
        return nc;
}
void mincost(int city)
{
        int i,ncity;
        completed[city]=1;
        cout<<city+1<<"--->";
        ncity=least(city);
        if(ncity==999)
        {
                ncity=0;
                cout<<ncity+1;
                cost+=ary[city][ncity];
                return;
        }
```

```
            mincost(ncity);
}

int main()
{
            takeInput();
            cout<<"\n\nThe Path is:\n";
            mincost(0); //passing 0 because starting vertex
            cout<<"\n\nMinimum cost is :\n"<<cost;
            cout<<"\n";
            return 0;
}
```

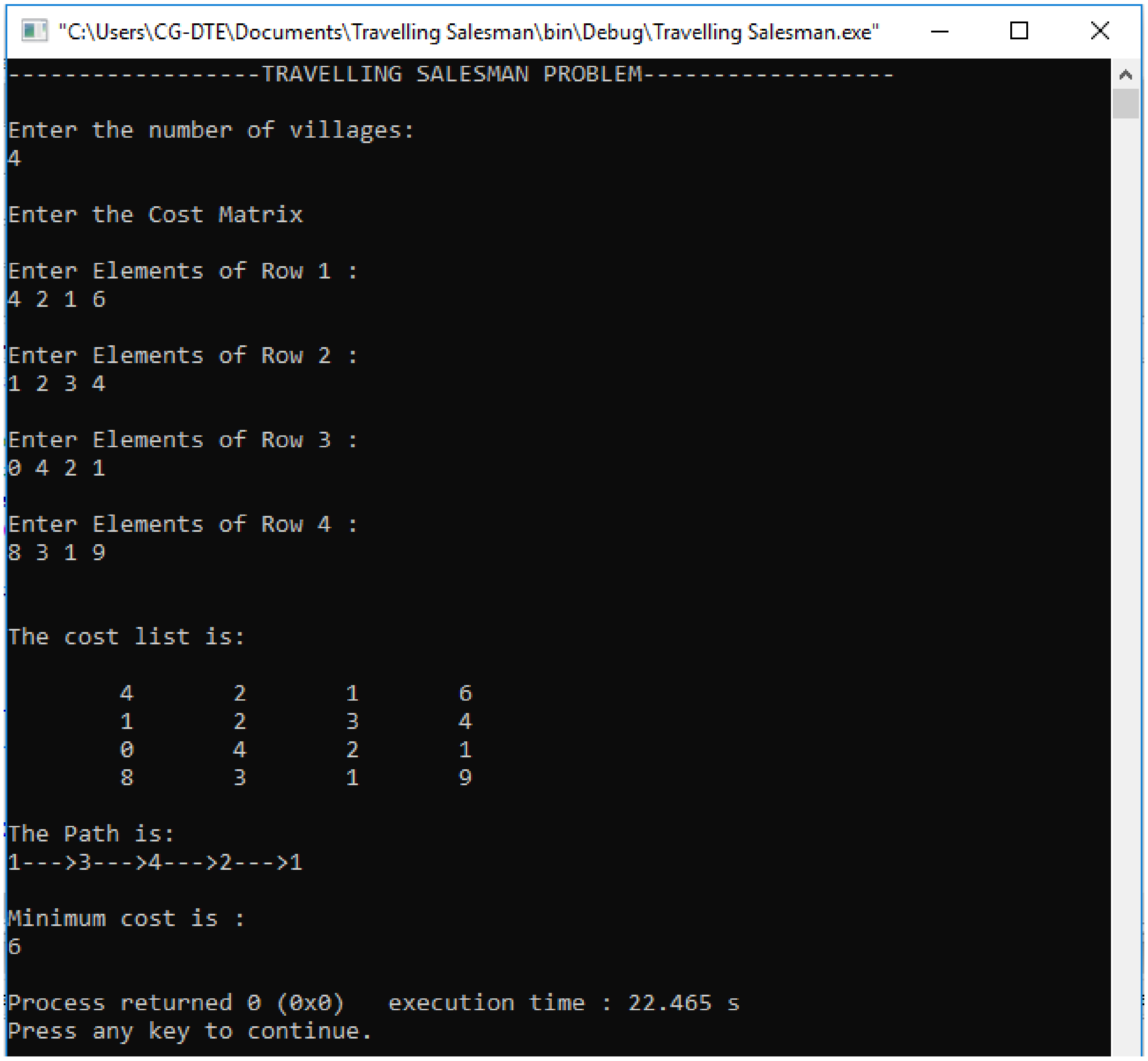
OUTPUT 1:



"C:\Users\CG-DTE\Documents\Travelling Salesman\bin\Debug\Travelling Salesman.exe"

```
------------------TRAVELLING SALESMAN PROBLEM------------------

Enter the number of villages:
4

Enter the Cost Matrix

Enter Elements of Row 1 :
0 4 1 3

Enter Elements of Row 2 :
4 0 2 1

Enter Elements of Row 3 :
1 2 0 5

Enter Elements of Row 4 :
3 1 5 0


The cost list is:

        0       4       1       3
        4       0       2       1
        1       2       0       5
        3       1       5       0

The Path is:
1--->3--->2--->4--->1

Minimum cost is :
7

Process returned 0 (0x0)   execution time : 20.037 s
Press any key to continue.
```

OUTPUT 2:



"C:\Users\CG-DTE\Documents\Travelling Salesman\bin\Debug\Travelling Salesman.exe"

```
------------------TRAVELLING SALESMAN PROBLEM------------------

Enter the number of villages:
4

Enter the Cost Matrix

Enter Elements of Row 1 :
4 2 1 6

Enter Elements of Row 2 :
1 2 3 4

Enter Elements of Row 3 :
0 4 2 1

Enter Elements of Row 4 :
8 3 1 9


The cost list is:

        4       2       1       6
        1       2       3       4
        0       4       2       1
        8       3       1       9

The Path is:
1--->3--->4--->2--->1

Minimum cost is :
6

Process returned 0 (0x0)   execution time : 22.465 s
Press any key to continue.
```

Q 12.

Write a program to search any goal given an input graph using AO* algorithm in AO graph.

DESCRIPTION:

(And-Or) Graph The Depth first search and Breadth first search given earlier for OR trees or graphs can be easily adopted by AND-OR graph. The main difference lies in the way termination conditions are determined, since all goals following an AND nodes must be realized; where as a single goal node following an OR node will do. So for this purpose we are using AO* algorithm.

Like A* algorithm here we will use two arrays and one heuristic function.
OPEN: It contains the nodes that has been traversed but yet not been marked solvable or unsolvable.
CLOSE: It contains the nodes that have already been processed.
h(n): The distance from current node to goal node.

ALGORITHM:

- Step 1: Place the starting node into OPEN.
- Step 2: Compute the most promising solution tree say T0.
- Step 3: Select a node n that is both on OPEN and a member of T0. Remove it from OPEN and place it in CLOSE
- Step 4: If n is the terminal goal node then leveled n as solved and leveled all the ancestors of n as solved. If the starting node is marked as solved then success and exit.
- Step 5: If n is not a solvable node, then mark n as unsolvable. If starting node is marked as unsolvable, then return failure and exit.
- Step 6: Expand n. Find all its successors and find their h (n) value, push them into OPEN.
- Step 7: Return to Step 2.
- Step 8: Exit.

IMPLEMENTATION:

CODE:

```
#include<bits/stdc++.h>
using namespace std;
struct node
{
        int data;
        vector< vector<node* >* >v;
        bool mark;
        bool solved;
};
int edge_cost=0;

void insert(node* root)
{
        cout<<"\nEnter data of node : ";
        cin>>root->data;
        cout<<"\nEnter number of OR nodes for value "<<root->data<<" : ";
        int or_no;
        cin>>or_no;
        for(int i=0; i<or_no; i++)
        {
                vector<node*>* ans=new vector<node*>;
                cout<<"\nEnter number of AND nodes for "<<i+1<<" or node for
                value "<<root->data<<" : ";
                int and_no;
                cin>>and_no;
                for(int j=0; j<and_no; j++)
                {
                        node* n=new node;
                        n->solved=false;
                        n->mark=false;
                        insert(n);
                        (*ans).push_back(n);
                }
                root->v.push_back(ans);
        }
}
void aostar(node* root)
{
        vector<node*>* min_ans=new vector<node*>;
        (*min_ans).push_back(root);
        while(!root->solved)
        {
                node* next_node=root;
                stack<node*>st;
                while(next_node && next_node->mark)
                {
                        if((next_node->v).size()==0)
```

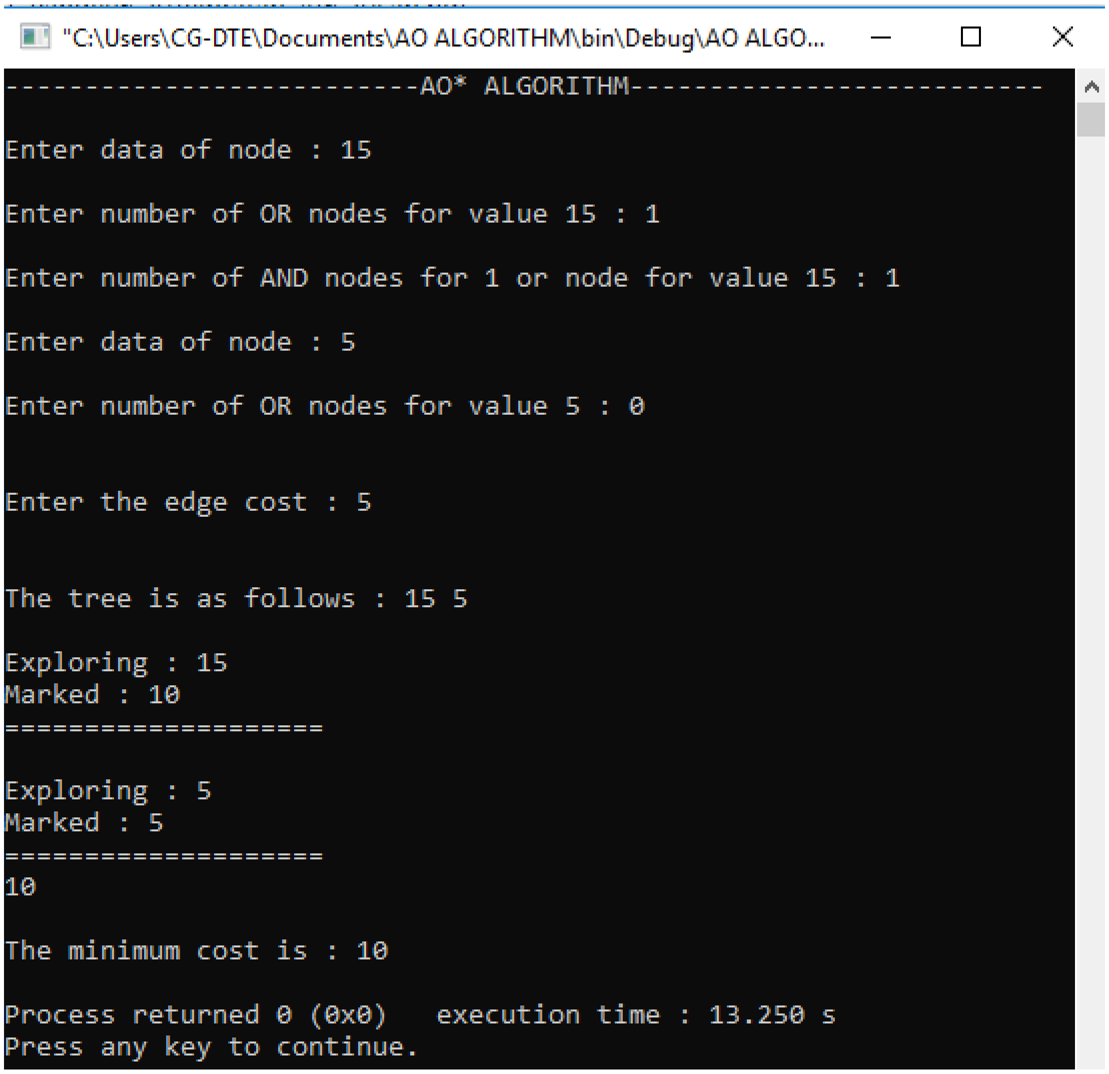
```
            {
                        root->solved=true;
                        return;
            }
            int cost=INT_MAX;
            st.push(next_node);
            for(unsigned int i=0; i<next_node->v.size(); i++)
            {
                        vector<node*>*ans=(next_node->v)[i];
                        vector<node*> ans_v=*ans;
                        int temp_cost=0;
                        for(unsigned int j=0; j<(ans_v.size()); j++)
                        {
                                    node* n=ans_v[j];
                                    temp_cost+=n->data;
                        }
                        if(temp_cost<cost)
                        {
                                    min_ans=ans;
                                    cost=temp_cost;
                        }
            }
            vector<node*> min_ans_v=*min_ans;
            next_node=NULL;
            for(unsigned int j=0; j<min_ans_v.size(); j++)
            {
                        if(min_ans_v[j]->mark)
                        {
                                    next_node=min_ans_v[j];
                                    break;
                        }
            }
}
vector<node*> min_ans_v=*min_ans;
for(unsigned int j=0; j<min_ans_v.size(); j++)
{
            node* n=min_ans_v[j];
            cout<<"Exploring : "<<n->data<<endl;
            int final_cost=INT_MAX;
            if(n->v.size()==0)
            {
```

```
                        n->mark=true;
                }
                else
                {
                        for(unsigned int i=0; i<n->v.size(); i++)
                        {
                                vector<node*>*ans=(n->v)[i];
                                vector<node*> ans_v=*ans;
                                int temp_cost=0;
                            for(unsigned int j=0; j<(ans_v.size()); j++)
                                {
                                        node* n=ans_v[j];
                                        temp_cost+=n->data;
                                        temp_cost+=edge_cost;
                                }
                                if(temp_cost<final_cost)
                                {
                                       final_cost=temp_cost;
                                }
                        }
                        n->data=final_cost;
                         n->mark=true;
               }
                cout<<"Marked : "<<n->data<<endl;
        }
        for(int i=0; i<20; i++)
        cout<<"=";
        cout<<endl;
        while(!st.empty())
        {
                node* n=st.top();
                cout<<n->data<<" ";
                st.pop();
                int final_cost=INT_MAX;
                for(unsigned int i=0; i<n->v.size(); i++)
                {
                        vector<node*>*ans=(n->v)[i];
                        vector<node*> ans_v=*ans;
                        int temp_cost=0;
                        for(unsigned int j=0; j<(ans_v.size()); j++)
                        {
```

```
                    node* n=ans_v[j];
                    temp_cost+=n->data;
                    temp_cost+=edge_cost;
                }
                if(temp_cost<final_cost)
                {
                    min_ans=ans;
                    final_cost=temp_cost;
                }
            }
            n->data=final_cost;
        }
        cout<<endl;
        next_node=root;
    }
}
void print(node* root)
{
    if(root)
    {
        cout<<root->data<<" ";
        vector<vector<node*>* >vec=root->v;
        for(unsigned int i=0; i<(root->v).size(); i++)
        {
            vector<node*>* ans=(root->v)[i];
            vector<node*> ans_v=*ans;
            for(unsigned int j=0; j<ans_v.size(); j++)
            {
                node* n=ans_v[j];
                print(n);
            }
        }
    }
    return;
}

int main()
{
    cout<<"--------------------------AO* ALGORITHM--------------------------\n";
    node* root=new node;
    root->solved=false;
```

```
                root->mark=false;
                insert(root);
                cout<<endl;
                cout<<"\nEnter the edge cost : ";
                cin>>edge_cost;
                cout<<endl;
                cout<<"\nThe tree is as follows : ";
                print(root);
                cout<<endl<<endl;
                aostar(root);
                cout<<"\nThe minimum cost is : "<<root->data<<endl;
                return 0;
}
```

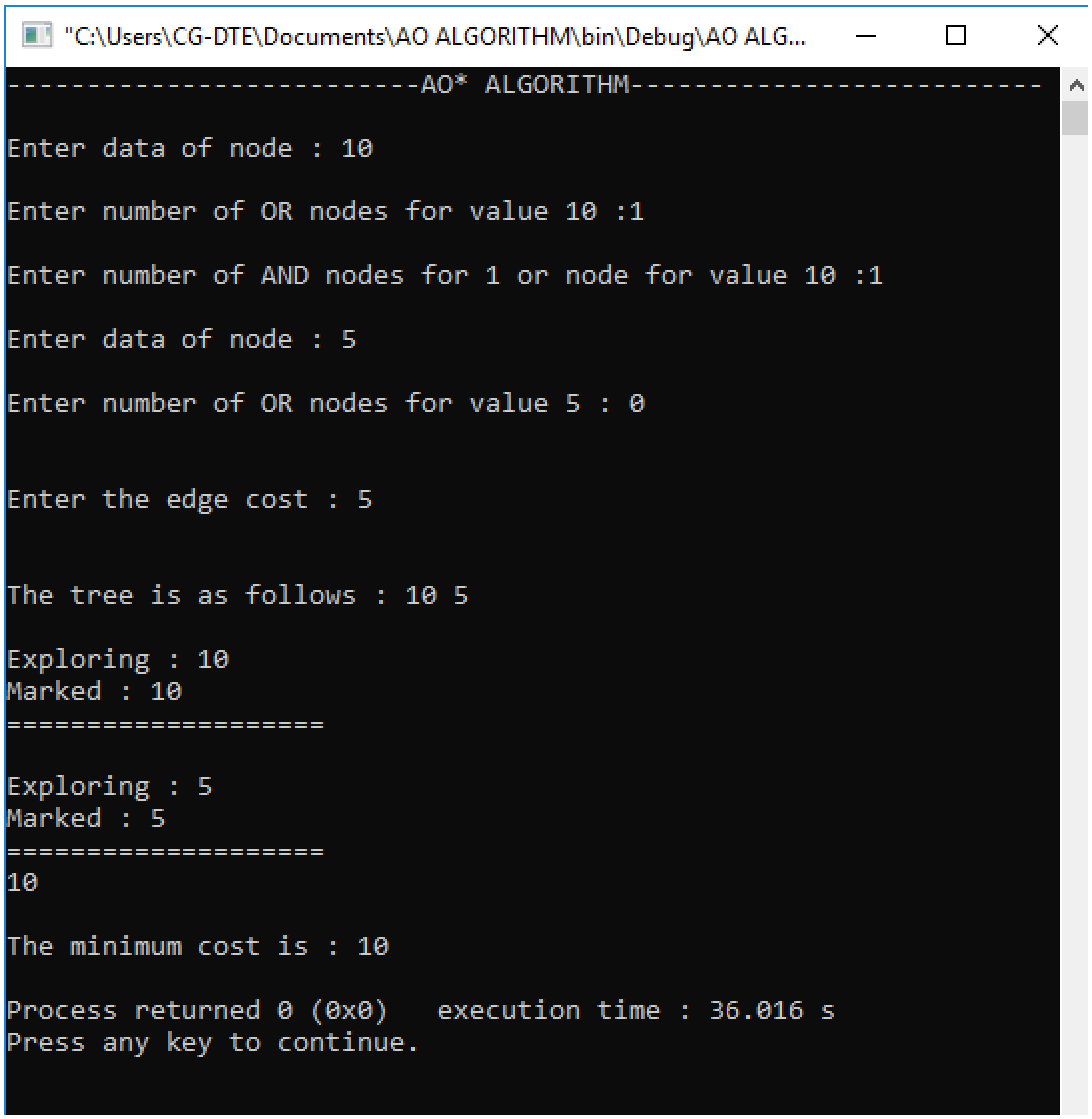
OUTPUT 1:



"C:\Users\CG-DTE\Documents\AO ALGORITHM\bin\Debug\AO ALGO...

```
--------------------------AO* ALGORITHM--------------------------

Enter data of node : 15

Enter number of OR nodes for value 15 : 1

Enter number of AND nodes for 1 or node for value 15 : 1

Enter data of node : 5

Enter number of OR nodes for value 5 : 0


Enter the edge cost : 5


The tree is as follows : 15 5

Exploring : 15
Marked : 10
====================

Exploring : 5
Marked : 5
====================
10

The minimum cost is : 10

Process returned 0 (0x0)   execution time : 13.250 s
Press any key to continue.
```

OUTPUT 2:



"C:\Users\CG-DTE\Documents\AO ALGORITHM\bin\Debug\AO ALG...

```
--------------------------AO* ALGORITHM--------------------------

Enter data of node : 10

Enter number of OR nodes for value 10 :1

Enter number of AND nodes for 1 or node for value 10 :1

Enter data of node : 5

Enter number of OR nodes for value 5 : 0


Enter the edge cost : 5


The tree is as follows : 10 5

Exploring : 10
Marked : 10
====================

Exploring : 5
Marked : 5
====================
10

The minimum cost is : 10

Process returned 0 (0x0)   execution time : 36.016 s
Press any key to continue.
```

Q13.
Implement the MINIMAX algorithm for any game playing.

DESCRIPTION:
Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.In Minimax the two players are called maximizer and minimizer. The **maximizer** tries to get the highest score possible while the **minimizer** tries to do the opposite and get the lowest score possible.Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

Let us combine minimax and evaluation function to write a proper Tic-Tac-Toe **AI** (**A**rtificial **I**ntelligence) that plays a perfect game.This AI will consider all possible scenarios and makes the most optimal move.

Finding the Best Move :
We shall be introducing a new function called findBestMove(). This function evaluates all the available moves using minimax() and then returns the best move the maximizer can make. The pseudocode is as follows :

```
function findBestMove(board):
   bestMove = NULL
   for each move in board :
      if current move is better than bestMove
         bestMove = current move
   return bestMove
```

Minimax :
To check whether or not the current move is better than the best move we take the help of minimax() function which will consider all the possible ways the game can go and returns the best value for that move, assuming the opponent also plays optimally

The code for the maximizer and minimizer in the minimax() function is similar to findBestMove(), the only difference is, instead of returning a move, it will return a value. Here is the pseudocode :

```
function minimax(board, depth, isMaximizingPlayer):
   if current board state is a terminal state :
      return value of the board
```

```
    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each move in board :
            value = minimax(board, depth+1, false)
            bestVal = max( bestVal, value)
        return bestVal

    else :
        bestVal = +INFINITY
        for each move in board :
            value = minimax(board, depth+1, true)
            bestVal = min( bestVal, value)
        return bestVal
```

Checking for GameOver state :
To check whether the game is over and to make sure there are no moves left we use isMovesLeft() function. It is a simple straightforward function which checks whether a move is available or not and returns true or false respectively.Pseudocode is as follows:

```
function isMovesLeft(board):
    for each cell in board:
        if current cell is empty:
            return true
    return false
```

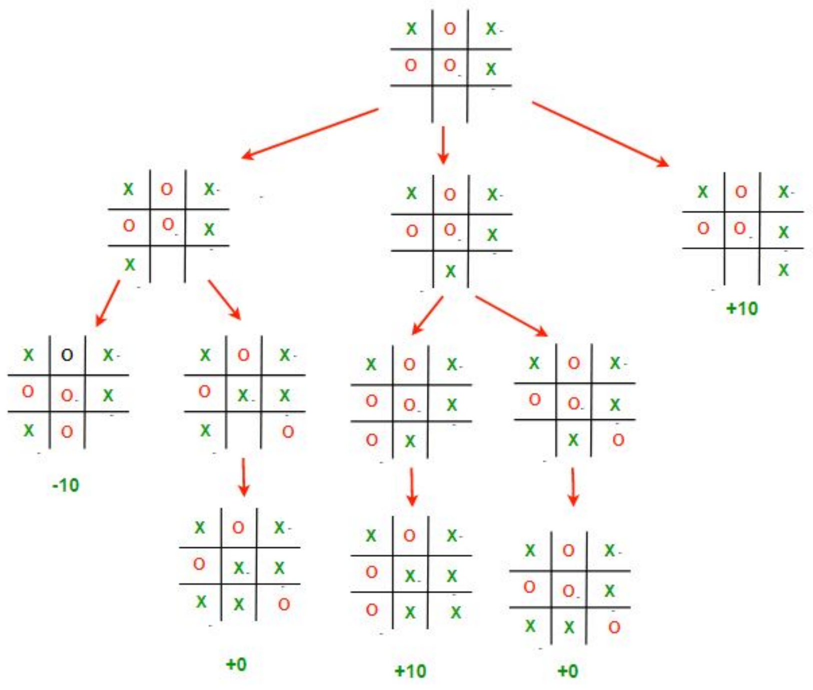
Making our AI smarter :
One final step is to make our AI a little bit smarter. Even though the following AI plays perfectly, it might choose to make a move which will result in a slower victory or a faster loss. Lets take an example and explain it.
Assume that there are 2 possible ways for X to win the game from a give board state.
Move A : X can win in 2 move
Move B : X can win in 4 moves

Our evaluation function will return a value of +10 for both moves A and B. Even though the move A is better because it ensures a faster victory, our AI may choose B sometimes. To overcome this problem we subtract the depth value from the evaluated score. This means that in case of a victory it will choose a the victory which takes least number of moves and in case of a loss it will try to prolong the game and play as many moves as possible. So the new evaluated value will be
Move A will have a value of +10 – 2 = 8
Move B will have a value of +10 – 4 = 6

Now since move A has a higher score compared to move B our AI will choose move A over move B. The same thing must be applied to the minimizer. Instead of subtracting the depth we add the depth value as the minimizer always tries to get, as negative a value as possible. We can subtract the depth either inside the evaluation function or outside it. Anywhere is fine. I have chosen to do it outside the function. Pseudocode implementation is as follows.

```
if maximizer has won:
   return WIN_SCORE – depth

else if minimizer has won:
   return LOOSE_SCORE + depth
```

EXPLANATION: GAME TREE



IMPLEMENTATION:

CODE:

```
// C++ program to find the next optimal move for a player
#include<bits/stdc++.h>
using namespace std;

struct Move
{
        int row, col;
```

```
};

char player = 'x', opponent = 'o';

// This function returns true if there are moves remaining on the board.
// It returns false if there are no moves left to play.
bool isMovesLeft(char board[3][3])
{
        for (int i = 0; i<3; i++)
           for (int j = 0; j<3; j++)
              if (board[i][j]=='_')
                 return true;
        return false;
}

// This is the evaluation function
int evaluate(char b[3][3])
{
            // Checking for Rows for X or O victory.
            for (int row = 0; row<3; row++)
            {
                        if (b[row][0]==b[row][1] && b[row][1]==b[row][2])
                        {
                                    if (b[row][0]==player)
                                           return +10;
                                    else if (b[row][0]==opponent)
                                       return -10;
                        }
            }

           // Checking for Columns for X or O victory.
           for (int col = 0; col<3; col++)
           {
                            if (b[0][col]==b[1][col] && b[1][col]==b[2][col])
                            {
                                        if (b[0][col]==player)
                                           return +10;
                                        else if (b[0][col]==opponent)
                                           return -10;
                            }
            }
        // Checking for Diagonals for X or O victory.
        if (b[0][0]==b[1][1] && b[1][1]==b[2][2])
        {
                if (b[0][0]==player)
                   return +10;
                else if (b[0][0]==opponent)
                   return -10;
        }

        if (b[0][2]==b[1][1] && b[1][1]==b[2][0])
        {
                if (b[0][2]==player)
                   return +10;
                else if (b[0][2]==opponent)
                   return -10;
        }

        // Else if none of them have won then return 0
        return 0;
}

// This is the minimax function. It considers all the possible ways the game can go
and returns the value of the board
int minimax(char board[3][3], int depth, bool isMax)
{
     int score = evaluate(board);

     // If Maximizer has won the game return his/her evaluated score
     if (score == 10)
        return score;

     // If Minimizer has won the game return his/her evaluated score
     if (score == -10)
        return score;

     // If there are no more moves and no winner then  it is a tie
     if (isMovesLeft(board)==false)
        return 0;

     // If this maximizer's move
```

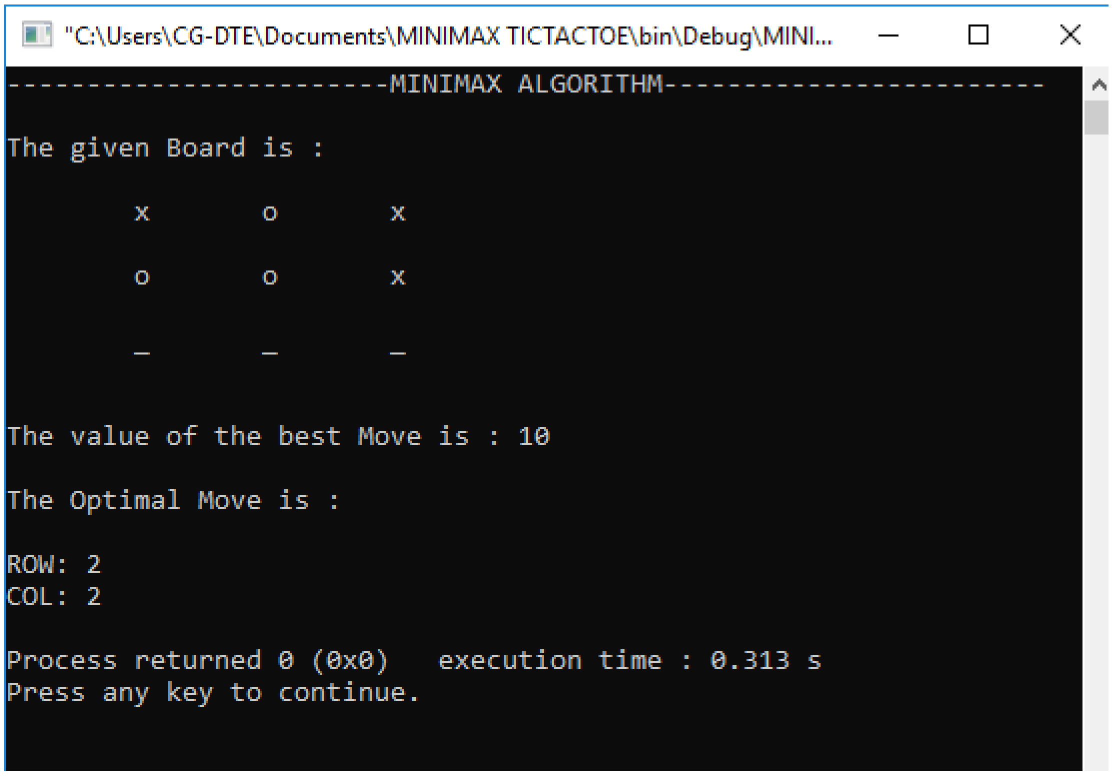
```
if (isMax)
{
            int best = -1000;

            // Traverse all cells
            for (int i = 0; i<3; i++)
            {
                        for (int j = 0; j<3; j++)
                        {
                                    // Check if cell is empty
                                    if (board[i][j]=='_')
                                    {
                                                // Make the move
                                                board[i][j] = player;

                                                // Call minimax recursively and
                                                //choose the maximum value
                                               best = max( best, minimax(board,
                                               depth+1, !isMax) );

                                                // Undo the move
                                                board[i][j] = '_';
                                    }
                        }
            }
            return best;
    }

    // If this minimizer's move
    else
    {
            int best = 1000;

            // Traverse all cells
            for (int i = 0; i<3; i++)
            {
                        for (int j = 0; j<3; j++)
                        {
                                    // Check if cell is empty
                                    if (board[i][j]=='_')
                                    {
```

```
                    // Make the move
                    board[i][j] = opponent;

                    // Call minimax recursively and
                    // choose the minimum value
                    best = min(best, minimax(board,
                    depth+1, !isMax));

                    // Undo the move
                    board[i][j] = '_';
                }
            }
        }
        return best;
    }
}

// This will return the best possible move for the player
Move findBestMove(char board[3][3])
{
    int bestVal = -1000;
    Move bestMove;
    bestMove.row = -1;
    bestMove.col = -1;

    // Traverse all cells, evaluate minimax function for all empty cells.
    // And return the cell with optimal value.
    for (int i = 0; i<3; i++)
    {
        for (int j = 0; j<3; j++)
        {
            // Check if cell is empty
            if (board[i][j]=='_')
            {
                // Make the move
                board[i][j] = player;

                //compute evaluation function for this move
                int moveVal = minimax(board, 0, false);

                // Undo the move
```

```
                        board[i][j] = '_';

                        // If the value of the current move is more
                        than the best value, then update best
                        if (moveVal > bestVal)
                        {
                                bestMove.row = i;
                                bestMove.col = j;
                                bestVal = moveVal;
                        }
                    }
                }
            }

            printf("\nThe value of the best Move is : %d\n\n", bestVal);
            return bestMove;
}

// Driver code
int main()
{
        char board[3][3] =
        {
            { 'x', 'o', 'x' },
            { 'o', 'o', 'x' },
            { '_', '_', '_' }
        };

        cout<<"------------------------MINIMAX ALGORITHM------------------------";
        cout<<"\n\n";
        cout << "The given Board is :\n\n";
        for (int i = 0; i < 3; ++i)
        {
                for (int j = 0; j < 3; ++j)
                {
                    cout << '\t' << board[i][j] << ' ';
                }
                cout << endl<<endl;
        }

        Move bestMove = findBestMove(board);
```

```
                printf("The Optimal Move is :\n\n");
                printf("ROW: %d\n", bestMove.row);
                printf("COL: %d\n", bestMove.col );
                return 0;
}
```

OUTPUT:



"C:\Users\CG-DTE\Documents\MINIMAX TICTACTOE\bin\Debug\MINI...

```
------------------------MINIMAX ALGORITHM------------------------

The given Board is :

        x       o       x

        o       o       x

        _       _       _


The value of the best Move is : 10

The Optimal Move is :

ROW: 2
COL: 2

Process returned 0 (0x0)   execution time : 0.313 s
Press any key to continue.
```

Q14.

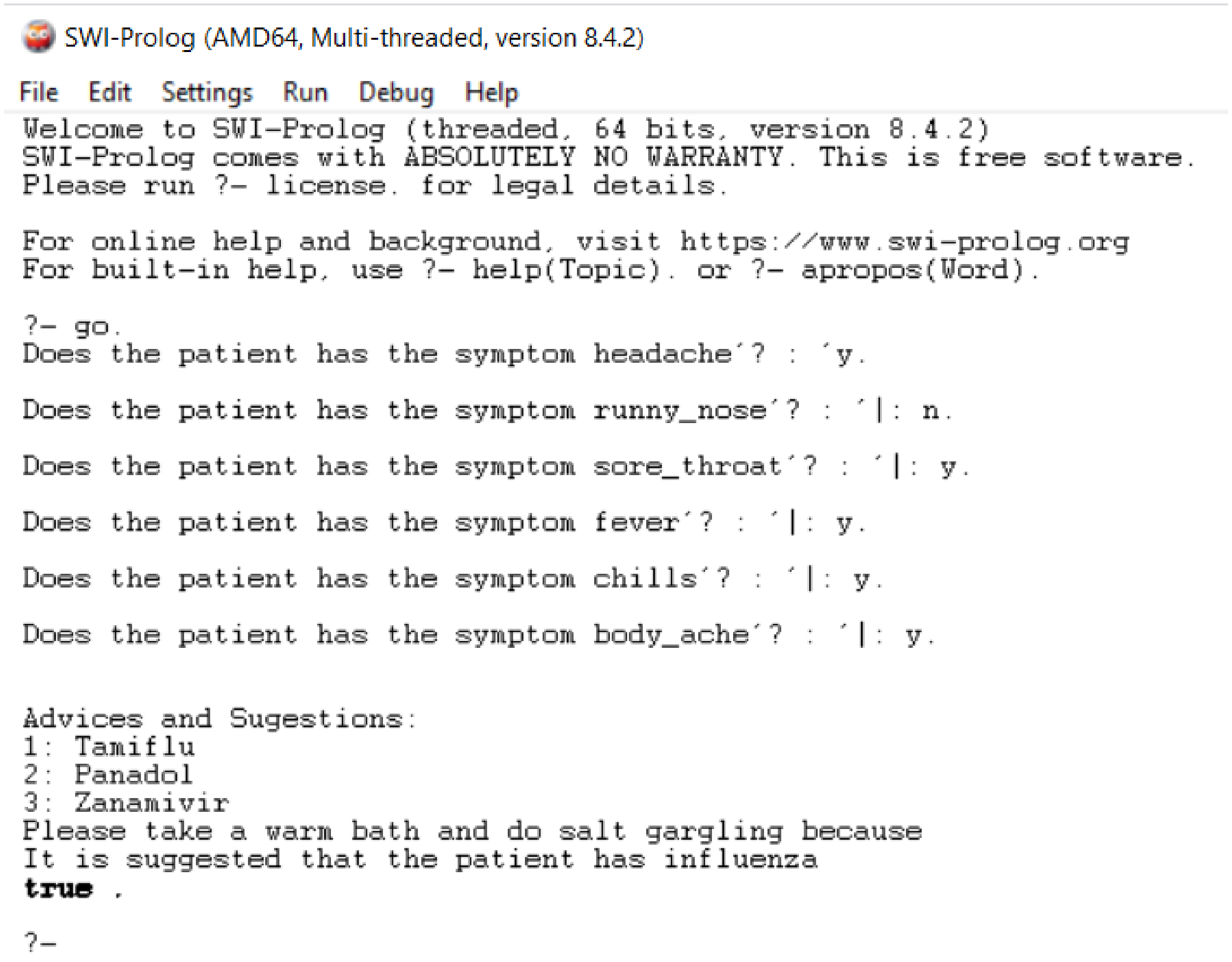
Design and implementation of miniproject on any expert system.

CODE:

PROLOG:

```
go:-
hypothesis(Disease),
write('It is suggested that the patient has '),
write(Disease),
nl,
undo;
write('Sorry, the system is unable to identify the disease'),nl,undo.

hypothesis(cold) :-
symptom(headache),
symptom(runny_nose),
symptom(sneezing),
symptom(sore_throat),
nl,
write('Advices and Sugestions:'),
nl,
write('1: Tylenol'),
nl,
write('2: Panadol'),
nl,
write('3: Nasal spray'),
nl,
write('Please weare warm cloths because'),
nl,!.

hypothesis(influenza) :-
symptom(sore_throat),
symptom(fever),
symptom(headache),
symptom(chills),
symptom(body_ache),
nl,
write('Advices and Sugestions:'),
nl,
write('1: Tamiflu'),
nl,
write('2: Panadol'),
nl,
write('3: Zanamivir'),
nl,
write('Please take a warm bath and do salt gargling because'),
nl,!.

hypothesis(typhoid) :-
symptom(headache),
symptom(abdominal_pain),
symptom(poor_appetite),
symptom(fever),
nl,
write('Advices and Sugestions:'),
nl,
write('1: Chloramphenicol'),
nl,
write('2: Amoxicillin'),
nl,
write('3: Ciprofloxacin'),
nl,
write('4: Azithromycin'),
nl,
write('Please do complete bed rest and take soft diet because'),
nl,!.

hypothesis(chicken_pox) :-
symptom(rash),
symptom(body_ache),
symptom(fever),
nl,
write('Advices and Sugestions:'),
nl,
write('1: Varicella vaccine'),
nl,
write('2: Immunoglobulin'),
nl,
write('3: Acetomenaphin'),
nl,
write('4: Acyclovir'),
nl,
write('Please do have oatmeal bath and stay at home because'),
nl.

hypothesis(measles) :-
symptom(fever),
symptom(runny_nose),
symptom(rash),
symptom(conjunctivitis),
nl,
write('Advices and Sugestions:'),
nl,
write('1: Tylenol'),
nl,
write('2: Aleve'),
nl,
write('3: Advil'),
nl,
write('4: Vitamin A'),
nl,
write('Please get rest and use more liquid because'),
nl,!.

hypothesis(malaria) :-
symptom(fever),
symptom(sweating),
symptom(headache),
symptom(nausea),
symptom(vomiting),
symptom(diarrhea),
nl,
write('Advices and Sugestions:'),
nl,
write('1: Aralen'),
nl,
write('2: Qualaquin'),
nl,
write('3: Plaquenil'),
nl,
write('4: Mefloquine'),
nl,
write('Please do not sleep in open air and cover your full skin because'),
nl,!.

ask(Question) :-
write('Does the patient has the symptom '),
write(Question),
write('? : '),
read(Response),
nl,
( (Response == yes ; Response == y)
->
assert(yes(Question)) ;
assert(no(Question)), fail).
:- dynamic yes/1,no/1.

symptom(S) :-
(yes(S)
->
true ;
(no(S)
->
fail ;
ask(S))).

undo :- retract(yes()),fail. undo :- retract(no()),fail.
undo.
```

OUTPUT:



SWI-Prolog (AMD64, Multi-threaded, version 8.4.2)

File Edit Settings Run Debug Help

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.2)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- go.
Does the patient has the symptom headache´? : ´y.

Does the patient has the symptom runny_nose´? : ´|: n.

Does the patient has the symptom sore_throat´? : ´|: y.

Does the patient has the symptom fever´? : ´|: y.

Does the patient has the symptom chills´? : ´|: y.

Does the patient has the symptom body_ache´? : ´|: y.


Advices and Sugestions:
1: Tamiflu
2: Panadol
3: Zanamivir
Please take a warm bath and do salt gargling because
It is suggested that the patient has influenza
true .

?-
```

<u>THE END</u>