**IV: Backtracking & Branch and bound Date:**

**Aim:-** Write algorithm and C program to implement the following problems using backtracking algorithms

1. N-queens
2. Sum of Subsets
3. Graph colouring (m-colouring)
4. Hamiltonian cycles
5. 0/1 knapsack

Theory:

Backtracking is widely regarded as one of the fundamental techniques in algorithm design, particularly useful for solving problems involving the search for a set of solutions or an optimal solution under certain constraints. The term "backtrack" was first coined by D.H. Lehmer in the 1950s, with early algorithmic accounts provided by R.J. Walker in 1960 and by

S. Golomb and L. Baumert, who presented a comprehensive description and various applications.

In many instances where backtracking is applied, the desired solution can be represented as an n-tuple (x1, ..., xn), with each xi chosen from a finite set Si. Frequently, the problem requires finding a vector that maximizes, minimizes, or satisfies a specific criterion function P(x1, ..., xn). This could involve finding all vectors that meet certain criteria.

For example, consider the problem of sorting an array of integers a[l:n]. Here, the solution can be expressed as an n-tuple, where xi represents the index in a of the ith smallest element. The criterion function P could be the inequality a[xj] < a[xi+1] for 1 < j < n, ensuring that the elements are sorted in ascending order.

The backtracking method typically involves recursively constructing partial solutions, making decisions at each step and potentially undoing those decisions (backtracking) if they lead to dead ends. To efficiently navigate the search space, bounding functions are often employed to determine whether a partial solution has any chance of leading to an optimal solution. One significant advantage of backtracking is its ability to prune the search space by eliminating branches that cannot possibly lead to an optimal solution.

Many problems tackled using backtracking involve satisfying a complex set of constraints. These constraints can be classified into two categories: explicit and implicit. Explicit constraints are those explicitly defined by the problem statement, while implicit constraints are additional restrictions that may not be explicitly stated but must still be satisfied by the solutions.

Overall, backtracking provides a systematic approach to exploring solution spaces, making it a powerful tool for solving a wide range of combinatorial optimization problems.

Backtracking is approached in a general manner, assuming the retrieval of all answer nodes rather than just one. In this context, let (x1, x2, ..., xi) represent a path from the root to a node in a state space tree. T(x1, x2, ..., xi+1) denotes the set of all possible values for xi+1 such

that (x1, x2, ..., xi+1) forms a path to a problem state. If T(x1, x2, ..., xn) is an empty set, it implies that no further extension of the path is possible.

The existence of bounding functions Bi+1, expressed as predicates, is assumed. These functions determine whether a path (x1, x2, ..., xi+1) from the root node to a problem state can be extended to reach an answer node. Thus, the candidates for position i+1 of the solution vector (x1, ..., xn) are those values generated by T and satisfy Bi+1.

Algorithm 7.1 provides a recursive formulation of the backtracking technique, which essentially performs a postorder traversal of a tree. The algorithm starts with an initial call to Backtrack(1), where the solution vector (x1, ..., xn) is treated as a global array. At each step, the algorithm generates possible values for the next position in the vector, checks if they satisfy the bounding function, and recursively continues the search. Solutions are printed as they are found.

Algorithm 7.2 presents a general iterative version of Algorithm 7.1, where solutions are generated and printed iteratively. This version maintains a loop to traverse the tree depth-first, incrementally building the solution vector until either a solution is found or no untried value remains. If backtracking is necessary, the algorithm decrements k to revisit the previous set of values for exploration.

The efficiency of these backtracking algorithms depends on several factors: the time taken to generate the next value of xi, the number of values satisfying explicit constraints, the time required for bounding functions, and the number of values satisfying the Bi constraints.

These factors collectively influence the algorithm's performance in exploring the solution space.

1. Algorithm Backtrack(k)
2. // This scheme describes the backtracking process using recursion. On entering, the first k - 1 values of the solution vector x[1], x[2], ..., x[k-1] have been assigned. x[] and n are global. 3 // Initialize the recursion with the current value of k.

4 {

5 for each value x[k] in T(x[1], ..., x[k-1]) do 6 {

7 // Check if the bounding function allows continuing with this value. 8 if (Bk(x[1], x[2], ..., x[k]) is true) then

9 {

1. // Check if the current partial solution forms a path to an answer node.
2. if (x[1], x[2], ..., x[k] is a path to an answer node) then
3. Write (x[1:k]); 13
4. // Recursively call Backtrack to explore further possibilities.
5. if (k < n) then
6. Backtrack(k + 1); 17 }

18 }

19 }

Algorithm7.1Recursive backtracking algorithm

Based on the provided context, here's a summary of how the solution vector is handled within the backtracking algorithm:

1. The solution vector (x1, ..., xn) is treated as a global array x[1:n].
2. All possible elements for the kth position of the tuple that satisfy the bounding function Bk are generated one by one.
3. Each generated value for xk is appended to the current vector (x1, ..., xk-1).
4. After attaching xk, a check is made to determine if a solution has been found.
5. If a solution has been found, it is processed accordingly (e.g., printed).
6. The algorithm is recursively invoked to continue exploring further possibilities.
7. When the loop of line 7 is exited, it indicates that no more values for xk exist, and the current instance of Backtrack ends.
8. The last unresolved call now resumes, which continues examining the remaining elements assuming only k - 1 values have been set.

This process allows the algorithm to systematically explore the solution space by recursively constructing and examining partial solutions until all possibilities are exhausted or a solution is found.

This algorithm prints all solutions and assumes that solutions may consist of tuples of various sizes. If only a single solution is desired, a flag can be added as a parameter to indicate the first occurrence of success.

1. Algorithm Backtrack(n)
2. // This scheme describes the backtracking process.
3. // All solutions are generated in x[1:n] and printed as soon as they are determined. 4 {

5 k := 1;

6 while (k != 0) do 7 {

8 if (there remains an untried x[k] in T(x[1], x[2], ..., x[k-1]) and Bi(x[1], ..., x[k]) is true) then

9 {

1. if (x[1], ..., x[k] is a path to an answer node) then
2. Write (x[1:k]);
3. k := k + 1; // Consider the next set. 13 }
4. else
5. k := k - 1; // Backtrack to the previous set. 16 }

17 }

Algorithm7.2General iterative backtracking method

This algorithm iterates over each position in the solution vector x[1:n], exploring possibilities until all solutions are found. It backtracks when necessary to explore alternative paths in the solution space. Solutions are printed as soon as they are determined.

An iterative version of Algorithm 7.1 is presented as Algorithm 7.2. Note that T() yields the set of all possible values that can be placed as the first component x1 of the solution vector. The component x1 will take on those values for which the bounding function B1(xi) is true. Additionally, note the depth-first manner in which elements are generated. The variable k is continually incremented, and a solution vector is grown until either a solution is found or no untried value of xk remains. When k is decremented, the algorithm must resume the generation of possible elements for the kth position that have not yet been tried. Therefore, one must develop a procedure that generates these values in some order. If only one solution is desired, replacing write(x[1:k]); with {write(x[1:k]); return;} suffices.

The efficiency of both the backtracking algorithms we've just seen depends very much on four factors: (1) the time to generate the next value of xk, (2) the number of values satisfying the explicit constraints, (3) the time for the bounding functions, and (4) the number of values satisfying the Bi constraints. These factors collectively influence the algorithm's performance in exploring the solution space.

Backtracking is a technique used in algorithm design for systematically searching for solutions to combinatorial problems, such as puzzles, constraint satisfaction problems, and optimization problems. Here are some key terminologies associated with backtracking:

1. Solution Space:The solution space refers to the set of all possible solutions to a given problem. It is the space that the backtracking algorithm explores in search of valid solutions.
2. State Space Tree: The state space tree, also known as the search tree or decision tree, is a tree-like data structure that represents all possible states of a problem and the transitions between them. Each node in the tree represents a partial or complete solution, and the edges represent the choices made to arrive at that state.
3. Partial Solution: A partial solution is an intermediate solution that does not yet satisfy all constraints of the problem. Backtracking algorithms recursively build and explore partial solutions until a complete solution is found.
4. Complete Solution:A complete solution is a solution that satisfies all constraints of the problem. In the context of backtracking, the algorithm terminates when a complete solution is found, or all possible paths have been explored without finding a valid solution.
5. Decision Variable: Decision variables are the variables whose values need to be determined in order to find a solution. In the context of backtracking, decision variables are often represented by the components of the solution vector.
6. Bounding Function: Bounding functions, also known as pruning functions, are used to eliminate portions of the search space that are known to be invalid. These functions help improve the efficiency of the backtracking algorithm by avoiding unnecessary exploration of unpromising paths.
7. Feasible Solution: A feasible solution is a solution that satisfies all constraints of the problem but may not necessarily be optimal. Backtracking algorithms aim to find feasible solutions efficiently.
8. Optimal Solution: An optimal solution is the best possible solution to the problem according to some criteria, such as maximizing or minimizing an objective function. Backtracking algorithms may be used to find optimal solutions by exploring the solution space systematically.

Understanding these terminologies is essential for effectively designing and implementing backtracking algorithms to solve various types of problems.

Backtracking is a powerful technique for solving combinatorial problems, but like any algorithmic approach, it comes with its own set of advantages and disadvantages.

Advantages:

1. Systematic Exploration: Backtracking systematically explores the solution space, ensuring that all possible solutions are considered. This makes it suitable for problems where exhaustive search is necessary.
2. Memory Efficiency: Backtracking typically requires less memory compared to other search algorithms like breadth-first search or depth-first search, as it only needs to store the current path being explored rather than the entire search tree.
3. Optimization Potential: Backtracking allows for the incorporation of bounding functions or pruning strategies, which can help eliminate portions of the search space early on. This can significantly reduce the time required to find a solution.
4. Flexibility: Backtracking can be applied to a wide range of combinatorial problems, including constraint satisfaction problems, optimization problems, and puzzles. It offers a versatile approach that can be adapted to different problem domains.
5. Ease of Implementation: Backtracking algorithms are often relatively simple to implement, especially for problems with well-defined state transitions and constraints. This makes them accessible to programmers of varying skill levels.

Disadvantages:

1. Exponential Time Complexity: In the worst case, backtracking can have exponential time complexity, especially for problems with a large solution space or complex constraints. This can make it impractical or infeasible for certain instances of the problem.
2. No Guarantee of Efficiency: While bounding functions and pruning strategies can help improve efficiency, there is no guarantee that backtracking will always find a solution quickly. The effectiveness of these optimizations depends on the problem instance and the quality of the bounding functions.
3. Difficulty in Designing Bounding Functions: Designing effective bounding functions can be challenging, especially for complex problems. Poorly designed bounding functions may not effectively reduce the search space, limiting the efficiency gains of backtracking.
4. Backtracking Overhead: The recursive nature of backtracking algorithms can introduce overhead, especially for problems with deep search trees. This overhead includes function call overhead and maintaining state information during recursion.
5. Non-Deterministic Nature: Backtracking is a non-deterministic algorithm, meaning that the order in which solutions are explored may vary depending on factors such as the input data or implementation details. This can make it harder to predict or analyze the algorithm's behavior in certain cases.

Despite these drawbacks, backtracking remains a valuable technique for solving a wide range of combinatorial problems, thanks to its systematic approach and flexibility. Effective use of backtracking requires careful consideration of the problem's characteristics and the design of appropriate optimization strategies.

Backtracking is a versatile technique that finds applications across various domains. Here are some common applications of backtracking:

1. Puzzle Solving: Backtracking is widely used in solving puzzles such as Sudoku, crosswords, word searches, and various logic puzzles. It systematically explores possible combinations of elements until a solution is found.
2. Constraint Satisfaction Problems (CSPs): Backtracking is commonly used to solve CSPs, where variables must be assigned values subject to certain constraints. Examples include the n-queens problem, graph coloring, and scheduling problems.
3. Combinatorial Optimization: Backtracking can be applied to combinatorial optimization problems, where the goal is to find the best combination of elements to optimize an objective function. Examples include the traveling salesman problem (TSP) and the knapsack problem.
4. Generating Permutations and Combinations: Backtracking is often used to generate all possible permutations or combinations of a given set of elements. It efficiently explores the entire solution space without generating duplicates.
5. String Matching and Pattern Recognition: Backtracking algorithms are used in string matching and pattern recognition tasks, such as searching for a pattern in a text or matching DNA sequences.
6. Cryptarithmetic Puzzles: Cryptarithmetic puzzles involve assigning digits to letters in a mathematical expression to satisfy certain arithmetic rules. Backtracking is commonly used to solve such puzzles by systematically exploring possible assignments.
7. Game Playing Algorithms: Backtracking is used in game playing algorithms, especially in games with deterministic rules and finite state spaces. Examples include chess, checkers, and tic-tac-toe.
8. Optimal Route Planning: Backtracking algorithms can be applied to find the optimal route or path in a graph or network, considering constraints such as distance, time, or cost. Examples include finding the shortest path in a maze or a network of cities.
9. Solving Cryptographic Problems: Backtracking can be used in cryptographic applications, such as breaking encryption schemes or cracking password hashes, by systematically trying different combinations until the correct one is found.
10. Satisfiability Testing: Backtracking algorithms are used in satisfiability testing (SAT), where the goal is to determine whether a given Boolean formula can be satisfied by assigning truth values to its variables.

These are just a few examples of the many applications of backtracking. Its systematic approach makes it a valuable technique for solving a wide range of combinatorial and optimization problems.

Theory of Backtracking:

1. Exploration of Solution Space: Backtracking is a systematic method used to explore the solution space of a problem by recursively trying different possibilities. It incrementally builds potential solutions, backtracks when a dead end is encountered, and continues until a valid solution is found or all possibilities have been explored.
2. State Space Tree: Backtracking can be visualized as traversing a state space tree, also known as a search tree or decision tree. Each node in the tree represents a partial solution or a state of the problem, and the edges represent the choices made to reach that state. The goal is to traverse this tree to find one or more valid solutions.
3. Backtracking Algorithm:
   * Initialization: The backtracking algorithm typically starts with an initial call, often with parameters representing the current state or decision variables.
   * Decision Making: At each step, the algorithm makes a decision to choose one option from the set of available options. This decision can be based on various criteria, such as feasibility, optimality, or heuristics.
   * Recursion: After making a decision, the algorithm recurses to explore further possibilities based on the chosen option. This recursive process continues until either a solution is found, or it becomes clear that the current path cannot lead to a solution.
   * Backtracking: If a dead end is reached (i.e., no more options are available), the algorithm backtracks to the previous decision point and tries a different option. This process continues until all possibilities have been explored or a valid solution is found.
   * Termination: The algorithm terminates when a valid solution is found, or all possibilities have been exhausted without finding a solution. In some cases, the algorithm may terminate early if certain criteria are met (e.g., a time limit or a maximum number of solutions).
4. Bounding Functions (Pruning: To improve efficiency, backtracking algorithms often use bounding functions, also known as pruning strategies, to eliminate portions of the search space that are known to be invalid. These functions help reduce the number of unnecessary explorations, speeding up the search process.
5. Completeness and Optimality: Backtracking algorithms can be complete, meaning they are guaranteed to find a solution if one exists, or they can be incomplete, where they may not always find a solution. Similarly, backtracking algorithms can be optimal if they always find the best possible solution, or they may find a suboptimal solution in some cases.
6. Time and Space Complexity: The time and space complexity of backtracking algorithms depend on factors such as the size of the solution space, the branching factor of the state space tree, the efficiency of bounding functions, and the complexity of decision- making criteria. In the worst case, backtracking algorithms can have exponential time complexity.

Practical Considerations:

1. Bounding Function Design : Designing effective bounding functions is crucial for improving the efficiency of backtracking algorithms. These functions should accurately identify invalid portions of the search space while minimizing false positives.
2. Optimization Techniques : Various optimization techniques, such as memoization, dynamic programming, and heuristic pruning, can be combined with backtracking to further improve efficiency and reduce exploration time.
3. Problem-Specific Strategies: Backtracking algorithms can often be customized with problem-specific strategies and heuristics to exploit domain knowledge and further accelerate the search process.

By understanding the theory and principles of backtracking, along with practical considerations and optimization techniques, developers can effectively apply this powerful algorithmic technique to solve a wide range of combinatorial and optimization problems.

**a) N-Queens**

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

typedef enum boolean {true = 1, false = 0}bool;

int \*x;

int n;

int solCount = 0, firstSol = 0;;

int \*y;

bool Place(int k,int i )

{

    for(int j = 0; j < k ; j++)

    {

        if((x[j] == i) || (abs(x[j]-i) == abs(j -k)))

            return false;

    }

    return true;

}

void N\_Quueens(int k)

{

    int i;

    if(k == n)

        return;

    for( i = 0; i < n; i++)

    {

        if(Place(k,i))

        {

            x[k] = i;

            if(k != n-1)

            {

                N\_Quueens(k+1);

            }

            else {

                if(firstSol == 0)

                {

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

                    {

                        y[j] = x[j];

                    }

                    firstSol = 1;

                }

                solCount++;

            }

        }

    }

}

int main()

{

    int totalNCount;

    printf("Enter the number of n queens : ");

    scanf("%d",&totalNCount); printf("\n");

    x = malloc(sizeof(int)\*totalNCount);

    y = malloc(sizeof(int)\*totalNCount);

    for(int i=0; i<totalNCount; i++)

    {

        x[i] = -1;

        y[i] = -1;

    }

    n = 1;

    for(int i=0; i<totalNCount; i++)

    {

        N\_Quueens(0);

        if(solCount !=0)

        {

            printf("%d - Queens problem soution\n",n);

            printf("Total Number of solutions : %d\n",solCount);

            printf("solution : { ");

            for(int j=0; j<n; j++)

            {

                if(j != n-1)

                    printf("%d,",y[j]+1);

                else

                    printf("%d",y[j]+1);

            }

            printf(" }\n");

            for(int j=0; j<n; j++)

            {

                for(int k=0; k<n ; k++)

                {

                    if(k == y[j])

                        printf(" Q ");

                    else

                        printf(" \* ");

                }

                printf("\n");

            }

            firstSol = solCount = 0;

        }

        else

        {

            printf("%d - Queens \n",n);

            printf("No solution for %d - Queens\n",n);

        }

        printf("\n");

        n++;

    }

}

**Output:**

Enter the number of n queens : 12

1 - Queens problem soution

Total Number of solutions : 1

solution : { 1 }

Q

2 - Queens

No solution for 2 - Queens

3 - Queens

No solution for 3 - Queens

4 - Queens problem soution

Total Number of solutions : 2

solution : { 2,4,1,3 }

\* Q \* \*

\* \* \* Q

Q \* \* \*

\* \* Q \*

5 - Queens problem soution

Total Number of solutions : 10

solution : { 1,3,5,2,4 }

Q \* \* \* \*

\* \* Q \* \*

\* \* \* \* Q

\* Q \* \* \*

\* \* \* Q \*

6 - Queens problem soution

Total Number of solutions : 4

solution : { 2,4,6,1,3,5 }

\* Q \* \* \* \*

\* \* \* Q \* \*

\* \* \* \* \* Q

Q \* \* \* \* \*

\* \* Q \* \* \*

\* \* \* \* Q \*

7 - Queens problem soution

Total Number of solutions : 40

solution : { 1,3,5,7,2,4,6 }

Q \* \* \* \* \* \*

\* \* Q \* \* \* \*

\* \* \* \* Q \* \*

\* \* \* \* \* \* Q

\* Q \* \* \* \* \*

\* \* \* Q \* \* \*

\* \* \* \* \* Q \*

8 - Queens problem soution

Total Number of solutions : 92

solution : { 1,5,8,6,3,7,2,4 }

Q \* \* \* \* \* \* \*

\* \* \* \* Q \* \* \*

\* \* \* \* \* \* \* Q

\* \* \* \* \* Q \* \*

\* \* Q \* \* \* \* \*

\* \* \* \* \* \* Q \*

\* Q \* \* \* \* \* \*

\* \* \* Q \* \* \* \*

9 - Queens problem soution

Total Number of solutions : 352

solution : { 1,3,6,8,2,4,9,7,5 }

Q \* \* \* \* \* \* \* \*

\* \* Q \* \* \* \* \* \*

\* \* \* \* \* Q \* \* \*

\* \* \* \* \* \* \* Q \*

\* Q \* \* \* \* \* \* \*

\* \* \* Q \* \* \* \* \*

\* \* \* \* \* \* \* \* Q

\* \* \* \* \* \* Q \* \*

\* \* \* \* Q \* \* \* \*

10 - Queens problem soution

Total Number of solutions : 724

solution : { 1,3,6,8,10,5,9,2,4,7 }

Q \* \* \* \* \* \* \* \* \*

\* \* Q \* \* \* \* \* \* \*

\* \* \* \* \* Q \* \* \* \*

\* \* \* \* \* \* \* Q \* \*

\* \* \* \* \* \* \* \* \* Q

\* \* \* \* Q \* \* \* \* \*

\* \* \* \* \* \* \* \* Q \*

\* Q \* \* \* \* \* \* \* \*

\* \* \* Q \* \* \* \* \* \*

\* \* \* \* \* \* Q \* \* \*

11 - Queens problem soution

Total Number of solutions : 2680

solution : { 1,3,5,7,9,11,2,4,6,8,10 }

Q \* \* \* \* \* \* \* \* \* \*

\* \* Q \* \* \* \* \* \* \* \*

\* \* \* \* Q \* \* \* \* \* \*

\* \* \* \* \* \* Q \* \* \* \*

\* \* \* \* \* \* \* \* Q \* \*

\* \* \* \* \* \* \* \* \* \* Q

\* Q \* \* \* \* \* \* \* \* \*

\* \* \* Q \* \* \* \* \* \* \*

\* \* \* \* \* Q \* \* \* \* \*

\* \* \* \* \* \* \* Q \* \* \*

\* \* \* \* \* \* \* \* \* Q \*

12 - Queens problem soution

Total Number of solutions : 14200

solution : { 1,3,5,8,10,12,6,11,2,7,9,4 }

Q \* \* \* \* \* \* \* \* \* \* \*

\* \* Q \* \* \* \* \* \* \* \* \*

\* \* \* \* Q \* \* \* \* \* \* \*

\* \* \* \* \* \* \* Q \* \* \* \*

\* \* \* \* \* \* \* \* \* Q \* \*

\* \* \* \* \* \* \* \* \* \* \* Q

\* \* \* \* \* Q \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \* \* Q \*

\* Q \* \* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* Q \* \* \* \* \*

\* \* \* \* \* \* \* \* Q \* \* \*

\* \* \* Q \* \* \* \* \* \* \* \*

b) Sum of Subsets

#include<stdio.h>

#include<stdlib.h>

#include <time.h>

// Some data types and idk macro?

typedef enum boolean {true = 1, false = 0}bool;

#define non\_leaf\_node 0

#define leaf\_node 1

#define soloution\_node 2

typedef struct Node

{

    int level;

    int s,k,r;

    int\* set;

    bool type;

} node;

// Some global variables

int n, m, solCount = 0;

int x[100], w[100], temp\_w[100];

node \*stack[100]; int stackTop=0;

// creatas the nodes to be put on the stack representing the tree

node\* createNode(int s, int k, int r, int \*\_x, int \_level, int type)

{

    node \*\_node = malloc((int)1\*sizeof(node));

    \_node->s = s;

    \_node->k = k;

    \_node->r = r;

    \_node->level = \_level;

    \_node->type = type;

    \_node->set = malloc(n\*sizeof(int));

    for(int i=0; i<n; i++)

    {

        \_node->set[i] = 0;

    }

    for(int i=0; i<=k; i++)

    {

        \_node->set[i] = \_x[i];

    }

    return \_node;

}

void reverseInputaArray()

{

    for(int i=0 ; i<n; i++)

    {

        temp\_w[i] = w[i];

    }

    for(int i=0, j=n-1; i<n; i++, j--)

    {

        w[i] = temp\_w[j];

    }

}

void randomizeInputArray()

{

    for (int i = 0; i < n - 1; i++) {

        int j = i + rand() / (RAND\_MAX / (n - i) + 1);

        int temp = w[j];

        w[j] = w[i];

        w[i] = temp;

    }

}

void clearStack()

{

    for(int i=0; i<100; i++)

    {

        free(stack[i]);

    }

    stackTop=0;

}

// its just push

void pushOnStack(node \*\_node)

{

    stack[stackTop++] = \_node;

}

// drawing tree

void printTree()

{

    int \_stackTop = stackTop-1;

    while(\_stackTop != 0)

    {

        for(int i=0; i< stack[\_stackTop]->level; i++)

        {

            printf("    ");

        }

        printf("Node %d ( %d %d %d ) ",\_stackTop,stack[\_stackTop]->s+1,stack[\_stackTop]->k+1,stack[\_stackTop]->r);

        if(stack[\_stackTop]->type == leaf\_node)

        {

            printf("x[");

            for(int j=0; j<n; j++)

            {

                if(j!=n-1)

                    printf("%d,",stack[\_stackTop]->set[j]);

                else

                    printf("%d",stack[\_stackTop]->set[j]);

            }

            printf("]");

        }

        else if(stack[\_stackTop]->type == soloution\_node)

        {

            printf("x[");

            for(int j=0; j<n; j++)

            {

                if(j!=n-1)

                    printf("%d,",stack[\_stackTop]->set[j]);

                else

                    printf("%d",stack[\_stackTop]->set[j]);

            }

            printf("] w:[");

            for(int j=0; j<n; j++)

            {

                if(stack[\_stackTop]->set[j]){

                    if(j!=n-1)

                        printf("%d,",j+1);

                    else

                        printf("%d",j+1);

                }

            }

            printf("] Solution %d",solCount++);

        }

        \_stackTop--;

        printf("\n");

    }

}

void sumOfSub(int s, int k, int r) {

    x[k] = 1;

    int level = k+1;

    // Checks if this is the solution

    if (s + w[k] == m)

    {

        pushOnStack(createNode(s+w[k],k,r-w[k],x,level,soloution\_node));

        pushOnStack(createNode(s,k-1,r,x,level-1,non\_leaf\_node));

    }

    // possible solution still ahead

    else if (s + w[k] + w[k+1] <= m)

    {

        // goes to left child means there is a possible soloution with this combination of x

        sumOfSub(s + w[k], k + 1, r - w[k]);

        pushOnStack(createNode(s,k-1,r,x,level-1,non\_leaf\_node));

    }

    else

    {

        pushOnStack(createNode(s,k-1,r,x,level-1,non\_leaf\_node));

    }

    if(k == n-1)

    {

        x[k] = 0;

        pushOnStack(createNode(s,k,r-w[k],x,level,leaf\_node));   // leaf node without a solution

    }

    // checks if there is a possible solution without this combination of x

    if ((s + r - w[k] >= m) && (s + w[k+1] <= m))

    {

        x[k] = 0;

        // goes to next possible combination of x without the current node

        sumOfSub(s, k+1, r - w[k]);

    }

}

int main() {

    int sum=0;

    printf("Enter the number of elements in set : ");

    scanf("%d",&n);

    printf("Enter the weights of the elements of the set : ");

    for(int i = 0; i < n; i++)

    {

        scanf("%d",&w[i]);

        sum += w[i];

    }

    printf("Enter the desired sum : ");

    scanf("%d",&m);

    for (int i = 0; i < n; i++) {

        x[i] = 0;

    }

    printf("\nSubsets using asscending input : CASE 1\n\n\n");

    pushOnStack(createNode(0,0,sum,x,0,non\_leaf\_node));

    sumOfSub(0,0,sum);

    printTree();

    printf("\n\n\nSubsets using descending input : CASE 2\n\n\n");

    reverseInputaArray();

    solCount=0;

    clearStack();

    pushOnStack(createNode(0,0,sum,x,0,non\_leaf\_node));

    sumOfSub(0,0,sum);

    printTree();

    printf("\n\n\nSubsets using jumbled input : CASE 3\n\n\n");

    srand(time(NULL));

    randomizeInputArray();

    printf("Randomized array\n");

    for(int i=0; i<n; i++)

    {

        if(i ==n-1)

        printf("%d",w[i]);

        else

        printf("%d,",w[i]);

    }

    printf("\n");

    solCount=0;

    clearStack();

    pushOnStack(createNode(0,0,sum,x,0,non\_leaf\_node));

    sumOfSub(0,0,sum);

    printTree();

    return 0;

}

**Output:**

Subsets using asscending input : CASE 1

Node 36 ( 1 7 0 ) x[0,0,0,0,0,0,0]

Node 35 ( 1 6 29 )

Node 34 ( 30 7 0 ) x[0,0,0,0,0,0,1] w:[7] Solution 0

Node 33 ( 1 5 53 )

Node 32 ( 1 4 72 )

Node 31 ( 1 3 81 )

Node 30 ( 10 4 72 )

Node 29 ( 1 2 85 )

Node 28 ( 5 5 53 )

Node 27 ( 5 4 72 )

Node 26 ( 5 3 81 )

Node 25 ( 1 1 87 )

Node 24 ( 3 5 53 )

Node 23 ( 3 4 72 )

Node 22 ( 3 3 81 )

Node 21 ( 3 2 85 )

Node 20 ( 7 4 72 )

Node 19 ( 7 3 81 )

Node 18 ( 1 0 88 )

Node 17 ( 2 5 53 )

Node 16 ( 2 4 72 )

Node 15 ( 2 3 81 )

Node 14 ( 11 4 72 )

Node 13 ( 30 5 53 ) x[1,0,0,1,1,0,0] w:[1,4,5,] Solution 1

Node 12 ( 2 2 85 )

Node 11 ( 6 5 53 )

Node 10 ( 30 6 29 ) x[1,0,1,0,0,1,0] w:[1,3,6,] Solution 2

Node 9 ( 6 4 72 )

Node 8 ( 6 3 81 )

Node 7 ( 2 1 87 )

Node 6 ( 4 5 53 )

Node 5 ( 4 4 72 )

Node 4 ( 4 3 81 )

Node 3 ( 4 2 85 )

Node 2 ( 8 4 72 )

Node 1 ( 8 3 81 )

Subsets using descending input : CASE 2

Node 5 ( 1 2 35 )

Node 4 ( 20 3 16 )

Node 3 ( 1 1 59 )

Node 2 ( 1 0 88 )

Node 1 ( 30 1 59 ) x[1,0,0,0,0,0,0] w:[1,] Solution 0

Subsets using jumbled input : CASE 3

Randomized array

1,2,19,24,29,9,4

Node 13 ( 1 4 42 )

Node 12 ( 30 5 13 ) x[0,0,0,0,1,0,0] w:[5,] Solution 0

Node 11 ( 1 3 66 )

Node 10 ( 1 2 85 )

Node 9 ( 1 1 87 )

Node 8 ( 3 3 66 )

Node 7 ( 3 2 85 )

Node 6 ( 1 0 88 )

Node 5 ( 2 3 66 )

Node 4 ( 2 2 85 )

Node 3 ( 2 1 87 )

Node 2 ( 4 3 66 )

Node 1 ( 4 2 85 )

**c) Graph Colouring**

#include <stdio.h>

#define TRUE 1

#define FALSE 0

#define MAX 20

int n, solncnt = 0;

int G[MAX][MAX], x[MAX];

void nextValue(int k, int m, int printNode)

{

    int j;

    do

    {

        x[k] = (x[k] + 1) % (m + 1);

        if (x[k] == 0)

            return;

        for (j = 1; j <= n; j++)

        {

            if (G[k][j] == 1 && x[k] == x[j])

                break;

        }

        if (j == n + 1)

        {

            if (printNode)

            {

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

                    printf(" %d\t", x[i]);

                printf("\n");

            }

            break;

        }

    } while (TRUE);

}

void graphColoring(int k, int m, int printSolution)

{

    int i, checked = 0;

    do

    {

        nextValue(k, m, printSolution);

        if (x[k] == 0)

            return;

        if (k == n)

        {

            if (!checked)

            {

                ++solncnt;

                if (printSolution)

                {

                    for (int i = 0; i <= 55; i++)

                        printf(" ");

                    printf("\n\n\e[1mSolution %3d : \e[m", solncnt);

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

                        printf("\e[1mx[%d] = %d \e[m", i, x[i]);

                    printf("\t\e[1mSolution Vector found\e[m");

                    printf("\n");

                    for (int i = 0; i <= 55; i++)

                        printf(" ");

                    printf("\n\n\n");

                    for (int v = 1; v <= 8; v++)

                        printf("x[%d]   ", v);

                    printf("\n");

                }

                checked = 1;

            }

        }

        else

            graphColoring(k + 1, m, printSolution);

    } while (TRUE);

}

int main()

{

    int i, j, max = 0, m;

    printf("Enter the number of vertices in the graph : ");

    scanf("%d", &n);

    for (i = 1; i <= n; i++)

        x[i] = 0;

    printf("Enter the adjacency matrix :\n");

    m = 0;

    for (i = 1; i <= n; i++)

    {

        max = 0;

        for (j = 1; j <= n; j++)

        {

            scanf("%d", &G[i][j]);

            if (G[i][j] == 1)

                max++;

        }

        if (max > m)

            m = max;

    }

    for (int v = 1; v <= 8; v++)

        printf("x[%d]   ", v);

    for (i = 1; i <= m; i++)

    {

        graphColoring(1, i, 0);

        if (solncnt != 0)

        {

            solncnt = 0;

            graphColoring(1, i, 1);

            printf("\nDegree of the graph : %d\n", ++m - 1);

            printf("Total number of solutions : %d\n", solncnt);

            printf("Chromatic number of the graph : %d\n", i);

            break;

        }

        for (j = 1; j <= n; j++)

            x[j] = 0;

    }

    return 0;

}

Output:

Enter the number of vertices in the graph : 8

Enter the adjacency matrix :

0 1 0 1 0 0 0 0

1 0 0 1 1 1 0 0

0 0 0 0 0 0 1 1

0 1 0 0 1 0 0 1

0 1 0 1 0 1 0 0

0 1 1 0 1 0 1 1

0 0 1 0 0 1 0 0

0 0 0 1 0 1 0 0

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

1

1 2

1 2 1

1 2 1 1

1 2 1

1 2 1 3

1 2 1 3 1

1 2 1 3 1 3

1 2 1 3 1 3 2

Solution 1 : x[1] = 1 x[2] = 2 x[3] = 1 x[4] = 3 x[5] = 1 x[6] = 3 x[7] = 2 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

1 2 1 3 1 3 2

1 2

1 2 2

1 2 2 1

1 2 2 1 3

1 2 2 1 3 1

1 2 2 1 3 1 3

Solution 2 : x[1] = 1 x[2] = 2 x[3] = 2 x[4] = 1 x[5] = 3 x[6] = 1 x[7] = 3 x[8] = 2 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

1 2 2 1 3 1 3

1 2 2

1 2 2 3

1 2 2 3 1

1 2 2 3 1 3

1 2 2 3 1 3 1

Solution 3 : x[1] = 1 x[2] = 2 x[3] = 2 x[4] = 3 x[5] = 1 x[6] = 3 x[7] = 1 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

1 2 2 3 1 3 1

1 2

1 2 3

1 2 3 1

1 2 3 1 3

1 2 3 1 3 1

1 2 3 1 3 1 2

Solution 4 : x[1] = 1 x[2] = 2 x[3] = 3 x[4] = 1 x[5] = 3 x[6] = 1 x[7] = 2 x[8] = 2 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

1 2 3 1 3 1 2

1 2 3

1 2 3 3

1

1 3

1 3 1

1 3 1 1

1 3 1

1 3 1 2

1 3 1 2 1

1 3 1 2 1 2

1 3 1 2 1 2 3

Solution 5 : x[1] = 1 x[2] = 3 x[3] = 1 x[4] = 2 x[5] = 1 x[6] = 2 x[7] = 3 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

1 3 1 2 1 2 3

1 3

1 3 2

1 3 2 1

1 3 2 1 2

1 3 2 1 2 1

1 3 2 1 2 1 3

Solution 6 : x[1] = 1 x[2] = 3 x[3] = 2 x[4] = 1 x[5] = 2 x[6] = 1 x[7] = 3 x[8] = 2 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

1 3 2 1 2 1 3

1 3 2

1 3 2 2

1 3

1 3 3

1 3 3 1

1 3 3 1 2

1 3 3 1 2 1

1 3 3 1 2 1 2

Solution 7 : x[1] = 1 x[2] = 3 x[3] = 3 x[4] = 1 x[5] = 2 x[6] = 1 x[7] = 2 x[8] = 2 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

1 3 3 1 2 1 2

1 3 3

1 3 3 2

1 3 3 2 1

1 3 3 2 1 2

1 3 3 2 1 2 1

Solution 8 : x[1] = 1 x[2] = 3 x[3] = 3 x[4] = 2 x[5] = 1 x[6] = 2 x[7] = 1 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

1 3 3 2 1 2 1

2

2 1

2 1 1

2 1 1 2

2 1 1 2 3

2 1 1 2 3 2

2 1 1 2 3 2 3

Solution 9 : x[1] = 2 x[2] = 1 x[3] = 1 x[4] = 2 x[5] = 3 x[6] = 2 x[7] = 3 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

2 1 1 2 3 2 3

2 1 1

2 1 1 3

2 1 1 3 2

2 1 1 3 2 3

2 1 1 3 2 3 2

Solution 10 : x[1] = 2 x[2] = 1 x[3] = 1 x[4] = 3 x[5] = 2 x[6] = 3 x[7] = 2 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

2 1 1 3 2 3 2

2 1

2 1 2

2 1 2 2

2 1 2

2 1 2 3

2 1 2 3 2

2 1 2 3 2 3

2 1 2 3 2 3 1

Solution 11 : x[1] = 2 x[2] = 1 x[3] = 2 x[4] = 3 x[5] = 2 x[6] = 3 x[7] = 1 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

2 1 2 3 2 3 1

2 1

2 1 3

2 1 3 2

2 1 3 2 3

2 1 3 2 3 2

2 1 3 2 3 2 1

Solution 12 : x[1] = 2 x[2] = 1 x[3] = 3 x[4] = 2 x[5] = 3 x[6] = 2 x[7] = 1 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

2 1 3 2 3 2 1

2 1 3

2 1 3 3

2

2 3

2 3 1

2 3 1 1

2 3 1

2 3 1 2

2 3 1 2 1

2 3 1 2 1 2

2 3 1 2 1 2 3

Solution 13 : x[1] = 2 x[2] = 3 x[3] = 1 x[4] = 2 x[5] = 1 x[6] = 2 x[7] = 3 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

2 3 1 2 1 2 3

2 3

2 3 2

2 3 2 1

2 3 2 1 2

2 3 2 1 2 1

2 3 2 1 2 1 3

Solution 14 : x[1] = 2 x[2] = 3 x[3] = 2 x[4] = 1 x[5] = 2 x[6] = 1 x[7] = 3 x[8] = 2 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

2 3 2 1 2 1 3

2 3 2

2 3 2 2

2 3

2 3 3

2 3 3 1

2 3 3 1 2

2 3 3 1 2 1

2 3 3 1 2 1 2

Solution 15 : x[1] = 2 x[2] = 3 x[3] = 3 x[4] = 1 x[5] = 2 x[6] = 1 x[7] = 2 x[8] = 2 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

2 3 3 1 2 1 2

2 3 3

2 3 3 2

2 3 3 2 1

2 3 3 2 1 2

2 3 3 2 1 2 1

Solution 16 : x[1] = 2 x[2] = 3 x[3] = 3 x[4] = 2 x[5] = 1 x[6] = 2 x[7] = 1 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

2 3 3 2 1 2 1

3

3 1

3 1 1

3 1 1 2

3 1 1 2 3

3 1 1 2 3 2

3 1 1 2 3 2 3

Solution 17 : x[1] = 3 x[2] = 1 x[3] = 1 x[4] = 2 x[5] = 3 x[6] = 2 x[7] = 3 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

3 1 1 2 3 2 3

3 1 1

3 1 1 3

3 1 1 3 2

3 1 1 3 2 3

3 1 1 3 2 3 2

Solution 18 : x[1] = 3 x[2] = 1 x[3] = 1 x[4] = 3 x[5] = 2 x[6] = 3 x[7] = 2 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

3 1 1 3 2 3 2

3 1

3 1 2

3 1 2 2

3 1 2

3 1 2 3

3 1 2 3 2

3 1 2 3 2 3

3 1 2 3 2 3 1

Solution 19 : x[1] = 3 x[2] = 1 x[3] = 2 x[4] = 3 x[5] = 2 x[6] = 3 x[7] = 1 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

3 1 2 3 2 3 1

3 1

3 1 3

3 1 3 2

3 1 3 2 3

3 1 3 2 3 2

3 1 3 2 3 2 1

Solution 20 : x[1] = 3 x[2] = 1 x[3] = 3 x[4] = 2 x[5] = 3 x[6] = 2 x[7] = 1 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

3 1 3 2 3 2 1

3 1 3

3 1 3 3

3

3 2

3 2 1

3 2 1 1

3 2 1

3 2 1 3

3 2 1 3 1

3 2 1 3 1 3

3 2 1 3 1 3 2

Solution 21 : x[1] = 3 x[2] = 2 x[3] = 1 x[4] = 3 x[5] = 1 x[6] = 3 x[7] = 2 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

3 2 1 3 1 3 2

3 2

3 2 2

3 2 2 1

3 2 2 1 3

3 2 2 1 3 1

3 2 2 1 3 1 3

Solution 22 : x[1] = 3 x[2] = 2 x[3] = 2 x[4] = 1 x[5] = 3 x[6] = 1 x[7] = 3 x[8] = 2 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

3 2 2 1 3 1 3

3 2 2

3 2 2 3

3 2 2 3 1

3 2 2 3 1 3

3 2 2 3 1 3 1

Solution 23 : x[1] = 3 x[2] = 2 x[3] = 2 x[4] = 3 x[5] = 1 x[6] = 3 x[7] = 1 x[8] = 1 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

3 2 2 3 1 3 1

3 2

3 2 3

3 2 3 1

3 2 3 1 3

3 2 3 1 3 1

3 2 3 1 3 1 2

Solution 24 : x[1] = 3 x[2] = 2 x[3] = 3 x[4] = 1 x[5] = 3 x[6] = 1 x[7] = 2 x[8] = 2 Solution Vector found

x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8]

3 2 3 1 3 1 2

3 2 3

3 2 3 3

Degree of the graph : 5

Total number of solutions : 24

Chromatic number of the graph : 3

d) Hamiltonian Cycle

#include <stdio.h>

#define MAX 100

int cnt = 0;

typedef struct

{

    int path[MAX];

    int length;

} Solution;

int graph[MAX][MAX];

int nol = 0, result\_count = 0;

Solution solutions[MAX];

void print\_path(int path[MAX], int n, int F)

{

    if (F == 1)

    {

        printf("\e[1m%d\e[m", path[1]);

    }

    else

    {

        printf("%d", path[1]);

    }

    for (int i = 2; i < n; i++)

    {

        if (F == 1)

        {

            printf("\e[1m -> %d\e[m", path[i]);

        }

        else

        {

            printf(" -> %d", path[i]);

        }

    }

    if (F == 1)

    {

        printf("\e[1m -> %d -> %d\e[m", path[n], path[1]);

    }

    else

    {

        printf("-> %d", path[1]);

    }

}

void next\_val(int path[MAX], int nodes, int k)

{

    do

    {

        path[k] = (path[k] + 1) % (nodes + 1);

        if (path[k] == 0)

        {

            return;

        }

        if (graph[path[k - 1]][path[k]] != 0)

        {

            int j;

            for (j = 1; j < k; j++)

            {

                if (path[j] == path[k])

                {

                    break;

                }

            }

            if (j == k)

            {

                if ((k < nodes) || (k == nodes && graph[path[nodes]][path[1]] != 0))

                {

                    return;

                }

            }

        }

    } while (1);

}

void Hamiltonian(int path[MAX], int nodes, int k)

{

    int c = 0;

    do

    {

        next\_val(path, nodes, k);

        if (path[k] != 0)

        {

            c = 1;

        }

        if (path[k] == 0)

        {

            if (c == 0)

            {

                printf(" %3d : ", ++nol);

                print\_path(path, k, 0);

                printf("\r\t\t\t\t\t\t\tBound\n");

            }

            return;

        }

        if (k == nodes)

        {

            printf("\e[1m %3d : \e[m", ++nol);

            print\_path(path, nodes, 1);

            printf("\r\t\t\t\t\t\t\t\e[1mSOLUTION\e[m\n");

            solutions[result\_count].length = nodes;

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

                solutions[result\_count].path[i] = path[i];

            result\_count++;

        }

        else

            Hamiltonian(path, nodes, k + 1);

    } while (1);

}

int main()

{

    int num\_nodes, path[MAX], k;

    printf("Enter the number of nodes: ");

    scanf("%d", &num\_nodes);

    for (int i = 1; i <= num\_nodes; i++)

    {

        for (int j = 1; j <= num\_nodes; j++)

        {

            graph[i][j] = 0;

        }

    }

    int edges[][2] = {

        {1, 2}, {1, 3}, {1, 7}, {2, 3}, {2, 4}, {2, 5}, {3, 4}, {3, 7}, {4, 6}, {5, 6}, {6, 7}};

    int num\_edges = sizeof(edges) / sizeof(edges[0]);

    for (int i = 0; i < num\_edges; i++)

    {

        graph[edges[i][0]][edges[i][1]] = 1;

        graph[edges[i][1]][edges[i][0]] = 1;

    }

    for (k = 1; k <= num\_nodes; k++)

    {

        path[1] = k;

        for (int i = 2; i <= num\_nodes; i++)

            path[i] = 0;

        printf("\nStarting vertex %d\nLeaf |\nNode | \n", k);

        Hamiltonian(path, num\_nodes, 2);

        nol = 0;

    }

    printf("\nAll possible solutions :\n");

    for (int i = 0; i < result\_count; i++)

    {

        cnt++;

        for (int j = 1; j <= solutions[i].length; j++)

        {

            if (j == solutions[i].length)

                printf("%d -> %d", solutions[i].path[j], solutions[i].path[1]);

            else

                printf("%d -> ", solutions[i].path[j]);

        }

            printf("\n");

    }

    printf("\nTotal number of solutions: %d\n", result\_count);

    return 0;

}

**Output:**

Enter the number of nodes: 7

Starting vertex 1

Leaf |

Node |

1 : 1 -> 2 -> 3 -> 4 -> 6 -> 5-> 1 Bound

2 : 1 -> 2 -> 3 -> 4 -> 6 -> 7-> 1 Bound

3 : 1 -> 2 -> 3 -> 7 -> 6 -> 4-> 1 Bound

4 : 1 -> 2 -> 3 -> 7 -> 6 -> 5-> 1 Bound

5 : 1 -> 2 -> 4 -> 3 -> 7 -> 6-> 1 Bound

6 : 1 -> 2 -> 4 -> 6 -> 5-> 1 Bound

7 : 1 -> 2 -> 4 -> 6 -> 7 -> 3-> 1 Bound

8 : 1 -> 2 -> 5 -> 6 -> 4 -> 3 -> 7 -> 1 SOLUTION

9 : 1 -> 2 -> 5 -> 6 -> 7 -> 3-> 1 Bound

10 : 1 -> 3 -> 2 -> 4 -> 6 -> 5-> 1 Bound

11 : 1 -> 3 -> 2 -> 4 -> 6 -> 7-> 1 Bound

12 : 1 -> 3 -> 2 -> 5 -> 6 -> 4-> 1 Bound

13 : 1 -> 3 -> 2 -> 5 -> 6 -> 7-> 1 Bound

14 : 1 -> 3 -> 4 -> 2 -> 5 -> 6 -> 7 -> 1 SOLUTION

15 : 1 -> 3 -> 4 -> 6 -> 5 -> 2-> 1 Bound

16 : 1 -> 3 -> 4 -> 6 -> 7-> 1 Bound

17 : 1 -> 3 -> 7 -> 6 -> 4 -> 2-> 1 Bound

18 : 1 -> 3 -> 7 -> 6 -> 5 -> 2-> 1 Bound

19 : 1 -> 7 -> 3 -> 2 -> 4 -> 6-> 1 Bound

20 : 1 -> 7 -> 3 -> 2 -> 5 -> 6-> 1 Bound

21 : 1 -> 7 -> 3 -> 4 -> 2 -> 5-> 1 Bound

22 : 1 -> 7 -> 3 -> 4 -> 6 -> 5 -> 2 -> 1 SOLUTION

23 : 1 -> 7 -> 6 -> 4 -> 2 -> 3-> 1 Bound

24 : 1 -> 7 -> 6 -> 4 -> 2 -> 5-> 1 Bound

25 : 1 -> 7 -> 6 -> 4 -> 3 -> 2-> 1 Bound

26 : 1 -> 7 -> 6 -> 5 -> 2 -> 3-> 1 Bound

27 : 1 -> 7 -> 6 -> 5 -> 2 -> 4 -> 3 -> 1 SOLUTION

Starting vertex 2

Leaf |

Node |

1 : 2 -> 1 -> 3 -> 4 -> 6 -> 5-> 2 Bound

2 : 2 -> 1 -> 3 -> 4 -> 6 -> 7-> 2 Bound

3 : 2 -> 1 -> 3 -> 7 -> 6 -> 4-> 2 Bound

4 : 2 -> 1 -> 3 -> 7 -> 6 -> 5-> 2 Bound

5 : 2 -> 1 -> 7 -> 3 -> 4 -> 6 -> 5 -> 2 SOLUTION

6 : 2 -> 1 -> 7 -> 6 -> 4 -> 3-> 2 Bound

7 : 2 -> 1 -> 7 -> 6 -> 5-> 2 Bound

8 : 2 -> 3 -> 1 -> 7 -> 6 -> 4-> 2 Bound

9 : 2 -> 3 -> 1 -> 7 -> 6 -> 5-> 2 Bound

10 : 2 -> 3 -> 4 -> 6 -> 5-> 2 Bound

11 : 2 -> 3 -> 4 -> 6 -> 7 -> 1-> 2 Bound

12 : 2 -> 3 -> 7 -> 1-> 2 Bound

13 : 2 -> 3 -> 7 -> 6 -> 4-> 2 Bound

14 : 2 -> 3 -> 7 -> 6 -> 5-> 2 Bound

15 : 2 -> 4 -> 3 -> 1 -> 7 -> 6 -> 5 -> 2 SOLUTION

16 : 2 -> 4 -> 3 -> 7 -> 1-> 2 Bound

17 : 2 -> 4 -> 3 -> 7 -> 6 -> 5-> 2 Bound

18 : 2 -> 4 -> 6 -> 5-> 2 Bound

19 : 2 -> 4 -> 6 -> 7 -> 1 -> 3-> 2 Bound

20 : 2 -> 4 -> 6 -> 7 -> 3 -> 1-> 2 Bound

21 : 2 -> 5 -> 6 -> 4 -> 3 -> 1-> 2 Bound

22 : 2 -> 5 -> 6 -> 4 -> 3 -> 7 -> 1 -> 2 SOLUTION

23 : 2 -> 5 -> 6 -> 7 -> 1 -> 3 -> 4 -> 2 SOLUTION

24 : 2 -> 5 -> 6 -> 7 -> 3 -> 1-> 2 Bound

25 : 2 -> 5 -> 6 -> 7 -> 3 -> 4-> 2 Bound

Starting vertex 3

Leaf |

Node |

1 : 3 -> 1 -> 2 -> 4 -> 6 -> 5-> 3 Bound

2 : 3 -> 1 -> 2 -> 4 -> 6 -> 7-> 3 Bound

3 : 3 -> 1 -> 2 -> 5 -> 6 -> 4-> 3 Bound

4 : 3 -> 1 -> 2 -> 5 -> 6 -> 7-> 3 Bound

5 : 3 -> 1 -> 7 -> 6 -> 4 -> 2-> 3 Bound

6 : 3 -> 1 -> 7 -> 6 -> 5 -> 2 -> 4 -> 3 SOLUTION

7 : 3 -> 2 -> 1 -> 7 -> 6 -> 4-> 3 Bound

8 : 3 -> 2 -> 1 -> 7 -> 6 -> 5-> 3 Bound

9 : 3 -> 2 -> 4 -> 6 -> 5-> 3 Bound

10 : 3 -> 2 -> 4 -> 6 -> 7 -> 1-> 3 Bound

11 : 3 -> 2 -> 5 -> 6 -> 4-> 3 Bound

12 : 3 -> 2 -> 5 -> 6 -> 7 -> 1-> 3 Bound

13 : 3 -> 4 -> 2 -> 1 -> 7 -> 6-> 3 Bound

14 : 3 -> 4 -> 2 -> 5 -> 6 -> 7 -> 1 -> 3 SOLUTION

15 : 3 -> 4 -> 6 -> 5 -> 2 -> 1 -> 7 -> 3 SOLUTION

16 : 3 -> 4 -> 6 -> 7 -> 1 -> 2-> 3 Bound

17 : 3 -> 7 -> 1 -> 2 -> 4 -> 6-> 3 Bound

18 : 3 -> 7 -> 1 -> 2 -> 5 -> 6 -> 4 -> 3 SOLUTION

19 : 3 -> 7 -> 6 -> 4 -> 2 -> 1-> 3 Bound

20 : 3 -> 7 -> 6 -> 4 -> 2 -> 5-> 3 Bound

21 : 3 -> 7 -> 6 -> 5 -> 2 -> 1-> 3 Bound

22 : 3 -> 7 -> 6 -> 5 -> 2 -> 4-> 3 Bound

Starting vertex 4

Leaf |

Node |

1 : 4 -> 2 -> 1 -> 3 -> 7 -> 6-> 4 Bound

2 : 4 -> 2 -> 1 -> 7 -> 3-> 4 Bound

3 : 4 -> 2 -> 1 -> 7 -> 6 -> 5-> 4 Bound

4 : 4 -> 2 -> 3 -> 1 -> 7 -> 6-> 4 Bound

5 : 4 -> 2 -> 3 -> 7 -> 1-> 4 Bound

6 : 4 -> 2 -> 3 -> 7 -> 6 -> 5-> 4 Bound

7 : 4 -> 2 -> 5 -> 6 -> 7 -> 1 -> 3 -> 4 SOLUTION

8 : 4 -> 2 -> 5 -> 6 -> 7 -> 3-> 4 Bound

9 : 4 -> 3 -> 1 -> 2 -> 5 -> 6-> 4 Bound

10 : 4 -> 3 -> 1 -> 7 -> 6 -> 5 -> 2 -> 4 SOLUTION

11 : 4 -> 3 -> 2 -> 1 -> 7 -> 6-> 4 Bound

12 : 4 -> 3 -> 2 -> 5 -> 6 -> 7-> 4 Bound

13 : 4 -> 3 -> 7 -> 1 -> 2 -> 5 -> 6 -> 4 SOLUTION

14 : 4 -> 3 -> 7 -> 6 -> 5 -> 2-> 4 Bound

15 : 4 -> 6 -> 5 -> 2 -> 1 -> 3-> 4 Bound

16 : 4 -> 6 -> 5 -> 2 -> 1 -> 7 -> 3 -> 4 SOLUTION

17 : 4 -> 6 -> 5 -> 2 -> 3 -> 1-> 4 Bound

18 : 4 -> 6 -> 5 -> 2 -> 3 -> 7-> 4 Bound

19 : 4 -> 6 -> 7 -> 1 -> 2 -> 3-> 4 Bound

20 : 4 -> 6 -> 7 -> 1 -> 2 -> 5-> 4 Bound

21 : 4 -> 6 -> 7 -> 1 -> 3 -> 2-> 4 Bound

22 : 4 -> 6 -> 7 -> 3 -> 1 -> 2-> 4 Bound

23 : 4 -> 6 -> 7 -> 3 -> 2 -> 1-> 4 Bound

24 : 4 -> 6 -> 7 -> 3 -> 2 -> 5-> 4 Bound

Starting vertex 5

Leaf |

Node |

1 : 5 -> 2 -> 1 -> 3 -> 4 -> 6-> 5 Bound

2 : 5 -> 2 -> 1 -> 3 -> 7 -> 6-> 5 Bound

3 : 5 -> 2 -> 1 -> 7 -> 3 -> 4 -> 6 -> 5 SOLUTION

4 : 5 -> 2 -> 1 -> 7 -> 6 -> 4-> 5 Bound

5 : 5 -> 2 -> 3 -> 1 -> 7 -> 6-> 5 Bound

6 : 5 -> 2 -> 3 -> 4 -> 6 -> 7-> 5 Bound

7 : 5 -> 2 -> 3 -> 7 -> 1-> 5 Bound

8 : 5 -> 2 -> 3 -> 7 -> 6 -> 4-> 5 Bound

9 : 5 -> 2 -> 4 -> 3 -> 1 -> 7 -> 6 -> 5 SOLUTION

10 : 5 -> 2 -> 4 -> 3 -> 7 -> 1-> 5 Bound

11 : 5 -> 2 -> 4 -> 3 -> 7 -> 6-> 5 Bound

12 : 5 -> 2 -> 4 -> 6 -> 7 -> 1-> 5 Bound

13 : 5 -> 2 -> 4 -> 6 -> 7 -> 3-> 5 Bound

14 : 5 -> 6 -> 4 -> 2 -> 1 -> 3-> 5 Bound

15 : 5 -> 6 -> 4 -> 2 -> 1 -> 7-> 5 Bound

16 : 5 -> 6 -> 4 -> 2 -> 3 -> 1-> 5 Bound

17 : 5 -> 6 -> 4 -> 2 -> 3 -> 7-> 5 Bound

18 : 5 -> 6 -> 4 -> 3 -> 1 -> 2-> 5 Bound

19 : 5 -> 6 -> 4 -> 3 -> 1 -> 7-> 5 Bound

20 : 5 -> 6 -> 4 -> 3 -> 2 -> 1-> 5 Bound

21 : 5 -> 6 -> 4 -> 3 -> 7 -> 1 -> 2 -> 5 SOLUTION

22 : 5 -> 6 -> 7 -> 1 -> 2 -> 3-> 5 Bound

23 : 5 -> 6 -> 7 -> 1 -> 2 -> 4-> 5 Bound

24 : 5 -> 6 -> 7 -> 1 -> 3 -> 2-> 5 Bound

25 : 5 -> 6 -> 7 -> 1 -> 3 -> 4 -> 2 -> 5 SOLUTION

26 : 5 -> 6 -> 7 -> 3 -> 1 -> 2-> 5 Bound

27 : 5 -> 6 -> 7 -> 3 -> 2 -> 1-> 5 Bound

28 : 5 -> 6 -> 7 -> 3 -> 2 -> 4-> 5 Bound

29 : 5 -> 6 -> 7 -> 3 -> 4 -> 2-> 5 Bound

Starting vertex 6

Leaf |

Node |

1 : 6 -> 4 -> 2 -> 1 -> 3 -> 7-> 6 Bound

2 : 6 -> 4 -> 2 -> 1 -> 7 -> 3-> 6 Bound

3 : 6 -> 4 -> 2 -> 3 -> 1 -> 7-> 6 Bound

4 : 6 -> 4 -> 2 -> 3 -> 7 -> 1-> 6 Bound

5 : 6 -> 4 -> 2 -> 5-> 6 Bound

6 : 6 -> 4 -> 3 -> 1 -> 2 -> 5-> 6 Bound

7 : 6 -> 4 -> 3 -> 1 -> 7-> 6 Bound

8 : 6 -> 4 -> 3 -> 2 -> 1 -> 7-> 6 Bound

9 : 6 -> 4 -> 3 -> 2 -> 5-> 6 Bound

10 : 6 -> 4 -> 3 -> 7 -> 1 -> 2 -> 5 -> 6 SOLUTION

11 : 6 -> 5 -> 2 -> 1 -> 3 -> 4-> 6 Bound

12 : 6 -> 5 -> 2 -> 1 -> 3 -> 7-> 6 Bound

13 : 6 -> 5 -> 2 -> 1 -> 7 -> 3 -> 4 -> 6 SOLUTION

14 : 6 -> 5 -> 2 -> 3 -> 1 -> 7-> 6 Bound

15 : 6 -> 5 -> 2 -> 3 -> 4-> 6 Bound

16 : 6 -> 5 -> 2 -> 3 -> 7 -> 1-> 6 Bound

17 : 6 -> 5 -> 2 -> 4 -> 3 -> 1 -> 7 -> 6 SOLUTION

18 : 6 -> 5 -> 2 -> 4 -> 3 -> 7-> 6 Bound

19 : 6 -> 7 -> 1 -> 2 -> 3 -> 4-> 6 Bound

20 : 6 -> 7 -> 1 -> 2 -> 4 -> 3-> 6 Bound

21 : 6 -> 7 -> 1 -> 2 -> 5-> 6 Bound

22 : 6 -> 7 -> 1 -> 3 -> 2 -> 4-> 6 Bound

23 : 6 -> 7 -> 1 -> 3 -> 2 -> 5-> 6 Bound

24 : 6 -> 7 -> 1 -> 3 -> 4 -> 2 -> 5 -> 6 SOLUTION

25 : 6 -> 7 -> 3 -> 1 -> 2 -> 4-> 6 Bound

26 : 6 -> 7 -> 3 -> 1 -> 2 -> 5-> 6 Bound

27 : 6 -> 7 -> 3 -> 2 -> 1-> 6 Bound

28 : 6 -> 7 -> 3 -> 2 -> 4-> 6 Bound

29 : 6 -> 7 -> 3 -> 2 -> 5-> 6 Bound

30 : 6 -> 7 -> 3 -> 4 -> 2 -> 1-> 6 Bound

31 : 6 -> 7 -> 3 -> 4 -> 2 -> 5-> 6 Bound

Starting vertex 7

Leaf |

Node |

1 : 7 -> 1 -> 2 -> 3 -> 4 -> 6-> 7 Bound

2 : 7 -> 1 -> 2 -> 4 -> 3-> 7 Bound

3 : 7 -> 1 -> 2 -> 4 -> 6 -> 5-> 7 Bound

4 : 7 -> 1 -> 2 -> 5 -> 6 -> 4 -> 3 -> 7 SOLUTION

5 : 7 -> 1 -> 3 -> 2 -> 4 -> 6-> 7 Bound

6 : 7 -> 1 -> 3 -> 2 -> 5 -> 6-> 7 Bound

7 : 7 -> 1 -> 3 -> 4 -> 2 -> 5 -> 6 -> 7 SOLUTION

8 : 7 -> 1 -> 3 -> 4 -> 6 -> 5-> 7 Bound

9 : 7 -> 3 -> 1 -> 2 -> 4 -> 6-> 7 Bound

10 : 7 -> 3 -> 1 -> 2 -> 5 -> 6-> 7 Bound

11 : 7 -> 3 -> 2 -> 1-> 7 Bound

12 : 7 -> 3 -> 2 -> 4 -> 6 -> 5-> 7 Bound

13 : 7 -> 3 -> 2 -> 5 -> 6 -> 4-> 7 Bound

14 : 7 -> 3 -> 4 -> 2 -> 1-> 7 Bound

15 : 7 -> 3 -> 4 -> 2 -> 5 -> 6-> 7 Bound

16 : 7 -> 3 -> 4 -> 6 -> 5 -> 2 -> 1 -> 7 SOLUTION

17 : 7 -> 6 -> 4 -> 2 -> 1 -> 3-> 7 Bound

18 : 7 -> 6 -> 4 -> 2 -> 3 -> 1-> 7 Bound

19 : 7 -> 6 -> 4 -> 2 -> 5-> 7 Bound

20 : 7 -> 6 -> 4 -> 3 -> 1 -> 2-> 7 Bound

21 : 7 -> 6 -> 4 -> 3 -> 2 -> 1-> 7 Bound

22 : 7 -> 6 -> 4 -> 3 -> 2 -> 5-> 7 Bound

23 : 7 -> 6 -> 5 -> 2 -> 1 -> 3-> 7 Bound

24 : 7 -> 6 -> 5 -> 2 -> 3 -> 1-> 7 Bound

25 : 7 -> 6 -> 5 -> 2 -> 3 -> 4-> 7 Bound

26 : 7 -> 6 -> 5 -> 2 -> 4 -> 3 -> 1 -> 7 SOLUTION

All possible solutions :

1 -> 2 -> 5 -> 6 -> 4 -> 3 -> 7 -> 1

1 -> 3 -> 4 -> 2 -> 5 -> 6 -> 7 -> 1

1 -> 7 -> 3 -> 4 -> 6 -> 5 -> 2 -> 1

1 -> 7 -> 6 -> 5 -> 2 -> 4 -> 3 -> 1

2 -> 1 -> 7 -> 3 -> 4 -> 6 -> 5 -> 2

2 -> 4 -> 3 -> 1 -> 7 -> 6 -> 5 -> 2

2 -> 5 -> 6 -> 4 -> 3 -> 7 -> 1 -> 2

2 -> 5 -> 6 -> 7 -> 1 -> 3 -> 4 -> 2

3 -> 1 -> 7 -> 6 -> 5 -> 2 -> 4 -> 3

3 -> 4 -> 2 -> 5 -> 6 -> 7 -> 1 -> 3

3 -> 4 -> 6 -> 5 -> 2 -> 1 -> 7 -> 3

3 -> 7 -> 1 -> 2 -> 5 -> 6 -> 4 -> 3

4 -> 2 -> 5 -> 6 -> 7 -> 1 -> 3 -> 4

4 -> 3 -> 1 -> 7 -> 6 -> 5 -> 2 -> 4

4 -> 3 -> 7 -> 1 -> 2 -> 5 -> 6 -> 4

4 -> 6 -> 5 -> 2 -> 1 -> 7 -> 3 -> 4

5 -> 2 -> 1 -> 7 -> 3 -> 4 -> 6 -> 5

5 -> 2 -> 4 -> 3 -> 1 -> 7 -> 6 -> 5

5 -> 6 -> 4 -> 3 -> 7 -> 1 -> 2 -> 5

5 -> 6 -> 7 -> 1 -> 3 -> 4 -> 2 -> 5

6 -> 4 -> 3 -> 7 -> 1 -> 2 -> 5 -> 6

6 -> 5 -> 2 -> 1 -> 7 -> 3 -> 4 -> 6

6 -> 5 -> 2 -> 4 -> 3 -> 1 -> 7 -> 6

6 -> 7 -> 1 -> 3 -> 4 -> 2 -> 5 -> 6

7 -> 1 -> 2 -> 5 -> 6 -> 4 -> 3 -> 7

7 -> 1 -> 3 -> 4 -> 2 -> 5 -> 6 -> 7

7 -> 3 -> 4 -> 6 -> 5 -> 2 -> 1 -> 7

7 -> 6 -> 5 -> 2 -> 4 -> 3 -> 1 -> 7

Total number of solutions: 28