CIA-1 SB COURSE

-PREPARED BY CSE-C

2 MARKS

1. Yes, the Knight's Tour problem can be solved on a standard 8x8 chessboard. The Knight's Tour is a classic chess problem that involves finding a sequence of moves for a knight piece on a chessboard in such a way that the knight visits every square exactly once. A standard 8x8 chessboard has 64 squares, and the Knight's Tour can indeed be solved on such a board. The Knight's Tour problem is a classic example of a mathematical puzzle and has been studied extensively. While finding a Knight's Tour is possible on an 8x8 chessboard, the problem becomes increasingly challenging on larger chessboards due to the greater number of squares and more complex configurations.

2. The Subset Sum problem involves determining whether a subset of a given set of positive integers can be chosen in such a way that their sum equals a specific target sum. It's a classic computational problem.
   Real-world scenarios where the Subset Sum problem arises include finance and investments (portfolio optimization), change making (determining the fewest coins/notes to make change), knapsack problems (resource allocation and logistics), cryptography (cryptanalysis), genetics and bioinformatics (gene identification), and manufacturing (component selection for cost reduction), among others. It's a fundamental problem in optimization and decision-making across various fields.

3. M-Colouring Problem in graph theory is about assigning colors to the vertices of a graph, ensuring that adjacent vertices have different colors, and using at most "M" colors. It's a decision problem, asking whether it's possible to color the graph with the given constraint of M colors.

The relationship to graph coloring is that the M-Colouring problem is a specific case within graph coloring. In graph coloring, you're typically concerned with finding the minimum number of colors (chromatic number) needed to color a graph. The M-Colouring problem, on the other hand, focuses on determining if a valid coloring exists using at most M colors. Both are important in graph theory and have practical applications in various areas.

4 .

Hamiltonian Cycle is a closed loop in a graph that visits each vertex exactly once, returning to the starting vertex.

Mathematically, for a graph G with vertices V, a Hamiltonian cycle is a sequence of vertices (v_1, v_2, ..., v_n) such that:

Each vertex in V appears exactly once in the sequence.

There is an edge between v_i and v_(i+1) for each i, where 1 ≤ i < n.

There is an edge between v_n and v_1, closing the cycle.

Hamiltonian Path is a path in a graph that visits every vertex exactly once but doesn't return to the starting vertex.

Mathematically, for a graph G with vertices V, a Hamiltonian path is a sequence of vertices (v_1, v_2, ..., v_n) such that:

Each vertex in V appears exactly once in the sequence.

There is an edge between v_i and v_(i+1) for each i, where 1 ≤ i < n.

5. A valid Sudoku puzzle should have a unique solution. If a Sudoku puzzle has multiple solutions, it is considered invalid because it violates the fundamental rules of Sudoku.

In a valid Sudoku puzzle, each number from 1 to 9 should be placed in such a way that every row, every column, and every 3x3 subgrid (also known as a "box") contains all the numbers from 1 to 9 exactly once. When there are multiple solutions, it means that there are different ways to fill in the grid while still satisfying these rules, which contradicts the concept of a well-constructed Sudoku puzzle.

6. The Sieve of Sundaram is a prime number sieve algorithm designed to generate a list of prime numbers. It was developed by the Indian mathematician S. P. Sundaram in the 1930s. The primary purpose of this algorithm is to efficiently and systematically identify prime numbers within a given range by eliminating composite (non-prime) numbers, making it a practical tool for generating prime numbers.

7. Yes, there can be multiple prime numbers following P with the same sum S due to Goldbach's Conjecture, which states that every even integer greater than 2 can be expressed as the sum of two prime numbers.

For example, if we take P = 5 and S = 12, there are several pairs of prime numbers that can satisfy this equation, such as (5, 7) and (11, 1), because:

5 + 7 = 12

11 + 1 = 12

8. Dijkstra's Algorithm:

Dijkstra's algorithm finds the minimum cost path from a source node to all other nodes in a weighted graph with non-negative edge weights.

It maintains a priority queue or set of visited nodes and iteratively selects the node with the minimum distance, updating distances to its neighbors.

Floyd-Warshall Algorithm:

The Floyd-Warshall algorithm finds the minimum cost path between all pairs of nodes in a weighted graph.

It uses a dynamic programming approach to iteratively update the minimum costs between all pairs of nodes.

9. Tracing the path of the minimum cost is crucial in scenarios such as GPS navigation, where it ensures efficient route planning, minimizes travel time, and provides accurate turn-by-turn directions. It is also vital in logistics and emergency services to optimize resource allocation and response times.

10. The objective of the minimum cost path problem in graph theory is to find the path between two nodes in a weighted graph that has the minimum sum of edge weights. This problem aims to identify the most economical or cost-effective route from a starting node to a destination node within the graph. It is often used in various applications, such as finding the shortest or cheapest route in transportation, minimizing travel time, or reducing resource consumption in network routing.

11. The common algorithmic approach to solving the Knight's Tour problem is to use a backtracking algorithm. This algorithm starts with an empty chessboard, places the knight on an arbitrary square, and recursively explores possible moves while maintaining a record of visited squares. If it successfully fills the entire board, a valid Knight's Tour is found. If not, it backtracks to explore alternative moves. The algorithm efficiently explores various move sequences, avoiding unnecessary paths and is adaptable to different board sizes and starting positions.

12. The backtracking approach for solving the subset sum problem is a recursive method that explores the decision space systematically. It begins with an empty subset and, at each step, makes decisions about including or excluding elements from the original set to create potential solutions. The algorithm prunes branches of the decision tree that cannot lead to a valid solution by checking if the current subset's sum exceeds the target sum. It continues this process until it finds a valid subset or exhausts all possibilities. Backtracking is an efficient way to tackle this problem, as it avoids unnecessary exploration of the decision space and is capable of finding multiple valid subsets if they exist.

13. The common algorithmic approach to solving the M-Colouring problem is to use a backtracking algorithm. This algorithm starts with an empty assignment of colors to the vertices and proceeds by picking a vertex, assigning a color, and checking if it violates the coloring constraint. If a violation occurs, the algorithm backtracks and tries a different color until a valid coloring is found or it determines that no valid coloring is possible. This approach efficiently explores the solution space and is commonly used for graph coloring problems.

14. The backtracking algorithm is a common algorithmic approach for finding a Hamiltonian cycle in a graph. It starts at an arbitrary vertex, explores possible Hamiltonian cycles through recursive exploration, and backtracks when necessary. The algorithm systematically searches for a cycle that visits every vertex exactly once and ends where it started, reporting success if found or failure if no Hamiltonian cycle exists. This approach is widely used for Hamiltonian cycle problems but may not be efficient for large or complex graphs due to its NP-completeness.

15. Constraint propagation is vital in Sudoku-solving as it efficiently reduces possibilities, enabling logical deductions and early error detection. It benefits both human solvers and computer algorithms, improving efficiency and puzzle insight.

16. The Sieve of Sundaram is preferred in specific cases, such as generating odd prime numbers efficiently. It is particularly useful when you need a list of small prime numbers or when memory efficiency is a concern due to its simple implementation. However, it cannot generate even primes and is less effective for very large primes. For general-purpose prime generation or when even primes are needed, other methods like the Sieve of Eratosthenes are often preferred.

17. Using an optimized algorithm to solve the "Prime Number after P with the Sum S" problem offers several advantages:

Efficiency: Optimized algorithms can significantly reduce computation time, making it faster to find the prime numbers that follow "P" and sum to "S." This is especially beneficial for large values of "P" and "S."

Reduced Resource Usage: Optimized algorithms typically use fewer computational resources, such as memory and processing power, which can be important for efficient execution on resource-constrained systems.

Scalability: Optimized algorithms can handle a wide range of input sizes, ensuring that the problem can be solved efficiently even as the values of "P" and "S" increase.

Accuracy: Well-optimized algorithms are designed to minimize errors and provide accurate results, ensuring that the prime numbers found are indeed the correct ones following "P" with the given sum "S."

18. The objective of the minimum cost path problem in graph theory is to find the path between two nodes in a weighted graph that has the minimum sum of edge weights. This problem aims to determine the most cost-effective or efficient route from a starting node to a destination node while considering the weights assigned to the edges as a measure of cost, distance, or time.

19. A minimum cost path in a graph should not contain loops or cycles. The term "path" implies a sequence of distinct vertices and edges that connect them. Loops or cycles, involving revisiting the same vertex multiple times, contradict the concept of a path, making it inefficient and leading to higher costs. The objective of finding a minimum cost path is to identify the most cost-effective and efficient route without revisiting the same vertices.

20. Dynamic programming is key to solving the minimum cost path problem efficiently by breaking it down into subproblems and storing their solutions. This approach avoids redundant calculations and reduces time complexity.

Tracing the path of the minimum cost is essential because it provides the actual route, which is often the primary goal. Just knowing the minimum cost is insufficient; the path helps in practical applications such as navigation and decision-making.

16MARKS:

1.

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

const int N = 8; // Change N for different board sizes

int moveX[] = {2, 1, -1, -2, -2, -1, 1, 2};
int moveY[] = {1, 2, 2, 1, -1, -2, -2, -1};

bool isSafe(int x, int y, vector<vector<int>>& board) {
  return (x >= 0 && x < N && y >= 0 && y < N && board[x][y] == -1);
}

void printSolution(vector<vector<int>>& board) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      cout << board[i][j] << "\t";
    }
```

```
        cout << endl;
    }
}

bool solveKT(int x, int y, int moveCount, vector<vector<int>>& board) {
    if (moveCount == N * N) {
        return true;
    }

    for (int k = 0; k < 8; k++) {
        int newX = x + moveX[k];
        int newY = y + moveY[k];
        if (isSafe(newX, newY, board)) {
            board[newX][newY] = moveCount;
            if (solveKT(newX, newY, moveCount + 1, board)) {
                return true;
            }
            board[newX][newY] = -1; // Backtrack
        }
    }
    return false;
}

int main() {
    vector<vector<int>> board(N, vector<int>(N, -1));
    board[0][0] = 0; // Starting position

    if (solveKT(0, 0, 1, board)) {
        cout << "Solution:\n";
        printSolution(board);
    } else {
        cout << "No solution exists." << endl;
    }

    return 0;
}
```

2.
```
#include <iostream>
#include <vector>
using namespace std;

int countSubsetsWithSum(vector<int>& set, int n, int K) {
    vector<vector<int>> dp(n + 1, vector<int>(K + 1, 0));

    for (int i = 0; i <= n; i++) {
        dp[i][0] = 1; // There is always one way to make sum 0, i.e., by not selecting any element.
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= K; j++) {
            if (set[i - 1] <= j) {
                dp[i][j] = dp[i - 1][j] + dp[i - 1][j - set[i - 1]];
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }

    return dp[n][K];
}
```

```
int main() {
    int n;
    cout << "Enter the number of elements in the set: ";
    cin >> n;

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

    int K;
    cout << "Enter the target sum (K): ";
    cin >> K;

    int result = countSubsetsWithSum(set, n, K);
    cout << "Number of subsets with sum " << K << ": " << result << endl;

    return 0;
}
```

3.
```
#include <iostream>
#include <vector>
using namespace std;

const int V = 4; // Change the number of vertices as needed

// Function to check if assigning color 'c' to vertex 'v' is safe
bool isSafe(int v, vector<vector<int>>& graph, vector<int>& color, int c) {
for (int i = 0; i < V; i++) {
    if (graph[v][i] && color[i] == c) {
        return false;
    }
}
return true;
}

// Recursive function to color the graph
bool graphColoring(vector<vector<int>>& graph, int m, vector<int>& color, int v) {
if (v == V) {
    return true;
}

for (int c = 1; c <= m; c++) {
    if (isSafe(v, graph, color, c)) {
        color[v] = c;
        if (graphColoring(graph, m, color, v + 1)) {
            return true;
        }
        color[v] = 0; // Backtrack
    }
}

return false;
}

int main() {
vector<vector<int>> graph(V, vector<int>(V, 0)); // Initialize the graph matrix

// Input the graph matrix
cout << "Enter the adjacency matrix for the graph (4x4 matrix):" << endl;
for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
        cin >> graph[i][j];
    }
}

int m;
cout << "Enter the number of colors (m): ";
cin >> m;

vector<int> color(V, 0);

if (graphColoring(graph, m, color, 0)) {
    cout << "Solution exists. Assigned colors for vertices:" << endl;
    for (int i = 0; i < V; i++) {
        cout << "Vertex " << i << " -> Color " << color[i] << endl;
    }
} else {
    cout << "No solution. The graph cannot be colored with " << m << " colors." << endl;
}

return 0;
}
```

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

```
const int MAX_V = 10;

bool isHamiltonianCycle(vector<vector<int>>& graph, vector<int>& path, int pos, int V) {
    if (pos == V) {
        if (graph[path[pos - 1]][path[0]] == 1) {
            return true; // Hamiltonian Cycle found
        }
        return false;
    }

    for (int v = 1; v < V; v++) {
        if (graph[path[pos - 1]][v] == 1) {
            bool visited = false;
            for (int i = 0; i < pos; i++) {
                if (path[i] == v) {
                    visited = true;
                    break;
                }
            }

            if (!visited) {
                path[pos] = v;
                if (isHamiltonianCycle(graph, path, pos + 1, V)) {
                    return true; // Hamiltonian Cycle found
                }
                path[pos] = -1; // Backtrack
            }
        }
    }

    return false; // Hamiltonian Cycle not found
```

```
}

bool findHamiltonianCycle(vector<vector<int>>& graph, int V) {
    vector<int> path(V, -1);
    path[0] = 0;

    if (!isHamiltonianCycle(graph, path, 1, V)) {
        return false; // Hamiltonian path does not exist
    }

    cout << "Hamiltonian Cycle: ";
    for (int i = 0; i < V; i++) {
        cout << path[i] << " ";
    }
    cout << path[0] << endl;

    return true;
}

int main() {
    int V;
    cout << "Enter the number of vertices (V): ";
    cin >> V;

    vector<vector<int>> graph(V, vector<int>(V, 0));

    cout << "Enter the adjacency matrix for the graph:" << endl;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cin >> graph[i][j];
        }
    }

    if (!findHamiltonianCycle(graph, V)) {
        cout << "Hamiltonian path does not exist." << endl;
    }

    return 0;
}
```

5.
```
#include <iostream>
using namespace std;

const int N = 9;

bool findEmptyLocation(int grid[N][N], int &row, int &col) {
    for (row = 0; row < N; row++) {
        for (col = 0; col < N; col++) {
            if (grid[row][col] == 0) {
                return true;
            }
        }
    }
    return false;
}

bool isSafe(int grid[N][N], int row, int col, int num) {
    for (int x = 0; x < N; x++) {
        if (grid[row][x] == num || grid[x][col] == num) {
            return false;
        }
    }

    int startRow = row - row % 3;
    int startCol = col - col % 3;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (grid[i + startRow][j + startCol] == num) {
                return false;
            }
        }
    }

    return true;
}

bool solveSudoku(int grid[N][N]) {
    int row, col;
    if (!findEmptyLocation(grid, row, col)) {
        return true;
    }

    for (int num = 1; num <= 9; num++) {
        if (isSafe(grid, row, col, num)) {
            grid[row][col] = num;
            if (solveSudoku(grid)) {
                return true;
            }
            grid[row][col] = 0;
        }
    }
    return false;
}

void printGrid(int grid[N][N]) {
    for (int row = 0; row < N; row++) {
        for (int col = 0; col < N; col++) {
            cout << grid[row][col] << " ";
        }
        cout << endl;
    }
}

int main() {
    int grid[N][N];

    cout << "Enter the Sudoku grid (9x9), use 0 for empty cells:" << endl;
    for (int row = 0; row < N; row++) {
        for (int col = 0; col < N; col++) {
            cin >> grid[row][col];
        }
    }

    if (solveSudoku(grid)) {
        cout << "Sudoku solution:" << endl;
        printGrid(grid);
    } else {
        cout << "No solution exists for the given Sudoku." << endl;
    }

    return 0;
```

```
}
```

6.
```
#include <iostream>
#include <vector>
using namespace std;

void sieveSundaram(int n) {
    int nHalf = (n - 1) / 2;
    vector<bool> marked(nHalf + 1, false);

    for (int i = 1; i <= nHalf; i++) {
        int j = i;
        while (i + j + 2 * i * j <= nHalf) {
            marked[i + j + 2 * i * j] = true;
            j++;
        }
    }

    // Print 2 separately, as it's the only even prime number
    if (n >= 2) {
        cout << 2 << " ";
    }

    // Print other prime numbers
    for (int i = 1; i <= nHalf; i++) {
        if (!marked[i]) {
            cout << 2 * i + 1 << " ";
        }
    }
}
```

```
int main() {
    int n;
    cout << "Enter a value (n) to find all primes <= n: ";
    cin >> n;

    if (n >= 2) {
        cout << "Prime numbers <= " << n << " are:" << endl;
        sieveSundaram(n);
        cout << endl;
    } else {
        cout << "No prime numbers <= " << n << endl;
    }

    return 0;
}
```

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

bool isPrime(int num) {
    if (num <= 1) return false;
    for (int i = 2; i * i <= num; i++) {
        if (num % i == 0) return false;
    }
    return true;
}

int main() {
    int S, N, P;
    cin >> S >> N >> P;

    vector<int> primes;
    while (primes.size() < N) {
        if (isPrime(P)) {
            primes.push_back(P);
        }
        P++;
    }

    if (primes.size() == N) {
        int sum = 0;
        for (int prime : primes) {
            sum += prime;
        }
        if (sum == S) {
            for (int prime : primes) {
                cout << prime << " ";
            }
            cout << endl;
        } else {
            cout << "No N primes after P with a sum of S." << endl;
        }
    } else {
        cout << "Not enough prime numbers after P." << endl;
    }

    return 0;
}
```

8.
```
#include <iostream>

#include <vector>

using namespace std;


int minCostPath(vector<vector<int>>& cost) {

   int N = cost.size();

   int M = cost[0].size();


   // Create a 2D array to store the minimum cost

   vector<vector<int>> minCost(N, vector<int>(M, 0));


   // Initialize the first cell

   minCost[0][0] = cost[0][0];


   // Initialize the first row

   for (int col = 1; col < M; col++) {

      minCost[0][col] = minCost[0][col - 1] + cost[0][col];

   }


   // Initialize the first column

   for (int row = 1; row < N; row++) {

      minCost[row][0] = minCost[row - 1][0] + cost[row][0];

   }


   // Calculate the minimum cost for each cell

   for (int row = 1; row < N; row++) {

      for (int col = 1; col < M; col++) {

         minCost[row][col] = min(minCost[row - 1][col], minCost[row][col - 1]) + cost[row][col];

      }

   }
```

```
    // Return the minimum cost for the bottom-right cell
    return minCost[N - 1][M - 1];
}

int main() {
    int N, M;
    cin >> N >> M;

    vector<vector<int>> cost(N, vector<int>(M));

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            cin >> cost[i][j];
        }
    }

    int result = minCostPath(cost);
    cout << "Minimum cost to reach the bottom-right cell: " << result << endl;

    return 0;
}
```

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

// Function to find all possible combinations that sum up to the target
```

```
void findCombinations(vector<int>& candidates, int target, vector<int>& current,
vector<vector<int>>& result, int index) {
    if (target == 0) {
        result.push_back(current);
        return;
    }

    for (int i = index; i < candidates.size(); i++) {
        if (candidates[i] <= target) {
            current.push_back(candidates[i]);
            findCombinations(candidates, target - candidates[i], current, result, i);
            current.pop_back();
        }
    }
}

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

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

    cout << "Enter the target sum: ";
    cin >> target;

    vector<vector<int>> result;
    vector<int> current;

    findCombinations(nums, target, current, result, 0);

    if (result.empty()) {
        cout << "No combinations found for the target sum." << endl;
    } else {
        cout << "Possible combinations for the target sum:" << endl;
        for (const vector<int>& combination : result) {
            for (int num : combination) {
                cout << num << " ";
            }
            cout << endl;
        }
    }

    return 0;
}
```

10. Backtracking is a powerful algorithmic technique used to solve problems by incrementally building a solution and abandoning it as soon as it's determined that it cannot lead to a valid solution. It systematically explores all possible solutions by trying different choices at each step, and it backtracks when a choice leads to a dead-end.

Here's a step-by-step explanation of the backtracking approach:

Initialization: Start with an empty or partial solution.

Construction: Incrementally build a solution by making choices at each step. These choices should lead to a valid solution.

Validation: At each step, check if the current partial solution is valid. If not, backtrack.

Completion: If the solution is complete (e.g., all elements placed, all paths explored), add it to the list of solutions.

Backtracking: If the current partial solution cannot be extended to a valid solution, backtrack to the previous step and make different choices.

Termination: Continue the process until all possibilities have been explored or a valid solution has been found.

CODING:

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

bool isSafe(vector<vector<int>>& board, int row, int col, int N) {
  for (int i = 0; i < col; i++) {
    if (board[row][i]) return false;  // Check the left side
    for (int j = 0; j < N; j++) {
      if (board[j][i] && abs(row - j) == abs(col - i)) return false;  // Check diagonals
    }
  }
  return true;
}
```

```
void solveNQueens(vector<vector<int>>& board, int col, int N, vector<vector<vector<int>>&
solutions) {
    if (col == N) {
        solutions.push_back(board);
        return;
    }
    for (int row = 0; row < N; row++) {
        if (isSafe(board, row, col, N)) {
            board[row][col] = 1;
            solveNQueens(board, col + 1, N, solutions);
            board[row][col] = 0;  // Backtrack
        }
    }
}

vector<vector<vector<int>> solveNQueens(int N) {
    vector<vector<int>> board(N, vector<int>(N, 0));
    vector<vector<vector<int>> solutions;
    solveNQueens(board, 0, N, solutions);
    return solutions;
}

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

    vector<vector<vector<int>> solutions = solveNQueens(N);

    if (solutions.empty()) {
        cout << "No solutions found." << endl;
    } else {
        for (const auto& solution : solutions) {
            for (const auto& row : solution) {
                for (int val : row) {
                    cout << (val == 1 ? "Q" : ".") << " ";
                }
                cout << endl;
            }
            cout << endl;
        }
    }

    return 0;
}
```

11. Dynamic Programming is an algorithmic technique used to solve complex problems by breaking them down into smaller overlapping subproblems. It stores the solutions to subproblems and reuses them when needed, which reduces redundant calculations. Dynamic programming is often employed in optimization and decision-making problems, where you want to find the best solution among many possible choices.

Here's how the dynamic programming approach works:

Initialization: Define the problem in terms of smaller subproblems.

Recurrence Relation: Express the solution of a problem in terms of solutions to smaller subproblems.

Memoization (Top-Down) or Tabulation (Bottom-Up): Store solutions to subproblems in a data structure (like a table or array) to avoid recomputation.

Optimal Solution: Use the solutions to subproblems to find the optimal solution to the original problem.

Applications of the dynamic programming approach to real-world problems:

Fibonacci Sequence: Calculate Fibonacci numbers efficiently using dynamic programming.

Shortest Path Algorithms: Find the shortest path in a graph (e.g., Dijkstra's or Floyd-Warshall algorithms).

Knapsack Problem: Solve the knapsack problem to maximize value within a weight limit.

String Matching: Implement string matching algorithms like Levenshtein distance and longest common subsequence.

Matrix Chain Multiplication: Optimize the order of matrix multiplication to minimize cost.

Time Complexity: The time complexity of dynamic programming depends on the problem and the specific approach used (top-down or bottom-up). In general, dynamic programming aims to reduce time complexity by reusing solutions to subproblems. However, the time complexity is often proportional to the product of the number of subproblems and the time it takes to solve each subproblem.

Space Complexity: Space complexity in dynamic programming depends on the method used (top-down or bottom-up) and how solutions to subproblems are stored. Tabulation (bottom-up) approaches often use less space than memoization (top-down) as they avoid function call stack overhead. Space complexity is proportional to the size of the table or array used to store solutions to subproblems.

CODING:

```
#include <iostream>

#include <vector>

using namespace std;


int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }

    vector<int> dp(n + 1, 0);  // Initialize a table to store Fibonacci numbers

    dp[0] = 0;
    dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];  // Use previously calculated values
    }

    return dp[n];
}

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

    int result = fibonacci(n);

    cout << "Fibonacci(" << n << ") = " << result << endl;

    return 0;
}
```

12. #include <iostream>

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

```
int countSubsetsWithSum(vector<int>& nums, int n, int target) {
    vector<vector<int>> dp(n + 1, vector<int>(target + 1, 0));

    // If the sum is 0, there is one empty subset.
    for (int i = 0; i <= n; i++) {
        dp[i][0] = 1;
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= target; j++) {
            if (nums[i - 1] <= j) {
                dp[i][j] = dp[i - 1][j] + dp[i - 1][j - nums[i - 1]];
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }

    return dp[n][target];
}

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

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

    int target;
    cout << "Enter the target sum: ";
    cin >> target;

    int count = countSubsetsWithSum(nums, n, target);

    cout << "Number of subsets with the sum " << target << " is: " << count << endl;

    return 0;
}
```

13. Backtracking is an algorithmic technique used to solve problems by incrementally building a solution and abandoning it as soon as it's determined that it cannot lead to a valid solution. Backtracking is often implemented using recursive functions. Here, I'll explain how to use backtracking to solve two specific tasks:

i. Recursion function to print the factors of the given number:

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

void printFactors(int n, int currentFactor) {
    if (currentFactor > n)
        return;

    if (n % currentFactor == 0)
        cout << currentFactor << " ";

    printFactors(n, currentFactor + 1);
}

int main() {
    int num;
    cout << "Enter a number: ";
    cin >> num;
    cout << "Factors of " << num << " are: ";
    printFactors(num, 1);
    return 0;
}
```

ii. Recursion function to print whether given integers are amicable numbers or not:

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

int sumOfDivisors(int n, int currentDivisor) {
    if (currentDivisor > n / 2)
        return 0;

    if (n % currentDivisor == 0)
        return currentDivisor + sumOfDivisors(n, currentDivisor + 1);

    return sumOfDivisors(n, currentDivisor + 1);
}

bool areAmicable(int num1, int num2) {
    int sum1 = sumOfDivisors(num1, 1);
    int sum2 = sumOfDivisors(num2, 1);
    return (sum1 == num2 && sum2 == num1);
}

int main() {
    int num1, num2;
    cout << "Enter two numbers: ";
    cin >> num1 >> num2;

    if (areAmicable(num1, num2))
        cout << "Amicable numbers!" << endl;
    else
        cout << "Not amicable numbers." << endl;

    return 0;
}
```

14. Initialize a 2D boolean table dp, where dp[i][j] indicates whether there is a subset of the first i elements that adds up to j. Also, initialize a 2D vector selected to track the selected elements.

Initialize the base cases:

dp[0][0] is true because you can have an empty subset that sums to 0.

For other dp[0][j], set them to false since there are no elements to choose from.

Iterate through the elements of the set:

For each element arr[i], update the dp table by considering two cases:

Case 1: Include arr[i]. Check if dp[i - 1][j - arr[i]] is true. If it is, set dp[i][j] to true and add arr[i] to the selected[i][j] list.

Case 2: Exclude arr[i]. If dp[i - 1][j] is true, copy the selected[i - 1][j] list to selected[i][j].

After the dynamic programming table is filled, you can check if there is a subset with the target sum (dp[n][targetSum]). If it's true, you can print the elements in the selected[n][targetSum] list, which represents the actual items in the subset.

Time Complexity:

The time complexity for the modified solution is O(n * targetSum), where 'n' is the number of elements in the set and 'targetSum' is the target sum. This is because we fill a 2D table of size (n+1) x (targetSum+1).

Space Complexity:

The space complexity is O(n * targetSum) as we use a 2D table and additional space for tracking selected elements.

CODING:

```
#include <iostream>

#include <vector>

using namespace std;


bool subsetSum(vector<int>& nums, int targetSum, vector<int>& selectedItems) {

   int n = nums.size();

   vector<vector<bool>> dp(n + 1, vector<bool>(targetSum + 1, false));


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

      dp[i][0] = true;

   }


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

      for (int j = 1; j <= targetSum; j++) {

         dp[i][j] = dp[i - 1][j];
```

```
            if (j >= nums[i - 1] && dp[i - 1][j - nums[i - 1]]) {
                dp[i][j] = true;
                selectedItems.push_back(nums[i - 1]);
            }
        }
    }

    return dp[n][targetSum];
}

int main() {
    vector<int> nums = {2, 4, 6, 8, 10};
    int targetSum = 18;
    vector<int> selectedItems;

    if (subsetSum(nums, targetSum, selectedItems)) {
        cout << "Subset with sum " << targetSum << " exists: ";
        for (int item : selectedItems) {
            cout << item << " ";
        }
        cout << endl;
    } else {
        cout << "No subset with sum " << targetSum << " exists." << endl;
    }

    return 0;
}
```

15. The Knight's Tour problem is a classic puzzle in the field of recreational mathematics and computer science. It involves finding a sequence of moves for a knight on a chessboard, such that the knight visits every square exactly once and then returns to its starting position. The knight moves in an L-shape, either two squares horizontally and one square vertically or two squares vertically and one square horizontally. The problem is often framed on an 8x8 chessboard, but it can be generalized to other board sizes.

Key components of the code:

is_valid_move: Checks if a move is within the board boundaries and the destination square has not been visited before.

knights_tour: Initializes the chessboard and starting position, then calls the solve_knights_tour function.

solve_knights_tour: Recursively explores all possible moves and backtracks when no valid moves are available.

print_solution: Prints the final solution.

Practical applications of solving the Knight's Tour problem:

Recreational Mathematics: It's a classic puzzle that enthusiasts enjoy solving for its mathematical and logical challenges.

Chess Programming: Understanding knight moves is essential for chess engines and move generation.

Optimization Problems: Similar techniques are used in routing, network design, and logistics to find efficient paths.

Game Development: Pathfinding for characters or game pieces with restricted movement patterns.

Education: Teaching problem-solving skills and algorithms in computer science and mathematics.

//CODING:

```
 // You are using GCC
#include<iostream>
using namespace std;
int chess[100][100];
int n;
bool isSafe(int n,int a,int b,int chess[][100])
```

```
{
    if(a>=0&& a<n && b>=0 &&b<n && chess[a][b]==-1)
    {
        return true;
    }
    else
    {
        return false;

    }
}
bool knight(int n,int a,int b,int chess[][100],int x[],int y[],int count)
{
    int xcoordinate,ycoordinate;
    if(count==n*n)
    {
        return true;
    }
    for(int i=0;i<8;i++)
    {
        xcoordinate=a+x[i];
        ycoordinate=b+y[i];

        if(isSafe(n,xcoordinate,ycoordinate,chess))
        {
            chess[xcoordinate][ycoordinate]=count;
            if(knight(n,xcoordinate,ycoordinate,chess,x,y,count+1))
            return true;
            else
            chess[xcoordinate][ycoordinate]=-1;
        }
    }
    return false;
}
void print()
{
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            cout<<chess[i][j]<<" ";
        }
        cout<<endl;
    }
}
int main()
{
    cin>>n;
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            chess[i][j]=-1;
        }
    }
    chess[0][0]=0;
    int x[]={2,1,-1,-2,-2,-1,1,2};
    int y[]={1,2,2,1,-1,-2,-2,-1};
    if(knight(n,0,0,chess,x,y,1))
    {
        cout<<"Solution exists: "<<endl;
        print();
    }
    else
    {
        cout<<"No solution exists."<<endl;
    }
}
```

16.

The Coin Change problem is a classic dynamic programming problem in which you are given a set of coin denominations and a target sum. The goal is to find the number of ways to make the target sum using the given denominations. It's a commonly used problem in the field of algorithms and is often solved using dynamic programming.

#CODING

```
#include <iostream>
using namespace std;
int count(int coins[], int n,int sum)
{
    if (sum == 0)
    {
        return 1;
    }
    if (sum < 0 )
    {
        return 0;
    }
    if(n<=0)
    {
        return 0;
    }
    return count(coins,n,sum-coins[n-1])+count(coins,n-1,sum);
}
int main()
{
    int n,i,sum;
    cin>>n;
    int coins[n];
    for(i=0;i<n;i++)
    {
        cin>>coins[i];
    }
    cin>>sum;
    cout<<count(coins,n,sum);
}
```

17.

M-coloring is a graph coloring problem in graph theory. The goal is to assign colors to the vertices of a graph in such a way that no two adjacent vertices have the same color, and the fewest number of colors are used. The minimum number of colors required for such a coloring is called the chromatic number of the graph, often denoted as χ(G).

The M-coloring algorithm involves recursively assigning colors to the vertices of a graph while ensuring that adjacent vertices do not have the same color. Here are the key steps:

Start with an arbitrary vertex and assign it a color, usually the first available color.

Move to the next uncolored vertex.

For this vertex, check the colors used by its adjacent vertices. Assign the lowest available color that has not been used by its neighbors.

Repeat steps 2 and 3 until all vertices are colored.

The chromatic number of the graph is the highest color used

```
//coding:
#include <iostream>
using namespace std;

const int V = 5; // Number of vertices

// Function to print the solution (coloring)
void printSolution(int color[]) {
   cout << "Solution exists: Following are the assigned colors:" << endl;
   for (int i = 0; i < V; i++)
      cout << color[i] << " ";
   cout << endl;
}

// Check if it's safe to color the vertex with the given color
bool isSafe(int v, int graph[V][V], int color[], int c) {
   for (int i = 0; i < V; i++) {
      if (graph[v][i] && color[i] == c)
         return false;
   }
   return true;
}

// Recursive function to perform graph coloring
bool graphColoringUtil(int graph[V][V], int m, int color[], int v) {
    if (v == V)
        return true;

    for (int c = 1; c <= m; c++) {
        if (isSafe(v, graph, color, c)) {
            color[v] = c;
            if (graphColoringUtil(graph, m, color, v + 1))
                return true;
            color[v] = 0; // Backtrack
        }
    }
    return false;
}

// Main function to perform graph coloring
bool graphColoring(int graph[V][V], int m) {
    int color[V] = {0}; // Initialize colors to 0

    if (!graphColoringUtil(graph, m, color, 0)) {
        cout << "No solution exists" << endl;
        return false;
    }

    printSolution(color);
    return true;
}
```

```
int main() {
   int graph[V][V] = {
       {0, 1, 1, 1, 0},
       {1, 0, 1, 0, 1},
       {1, 1, 0, 1, 0},
       {1, 0, 1, 0, 1},
       {0, 1, 0, 1, 0}
   };

   int m = 3; // Number of colors
   graphColoring(graph, m);

   return 0;
}
```

18. The Hamiltonian cycle problem is a classic NP-complete problem in graph theory. It involves finding a cycle that visits each vertex in a graph exactly once and returns to the starting vertex. This cycle is called a Hamiltonian cycle, and a graph that contains a Hamiltonian cycle is called a Hamiltonian graph.

One common algorithm for finding a Hamiltonian cycle is the Backtracking Algorithm. It explores all possible permutations of vertices and checks if a Hamiltonian cycle exists.

Here's a step-by-step explanation of the Backtracking Algorithm for finding a Hamiltonian cycle in a graph:

Start at an arbitrary vertex (usually vertex 0).

Mark the current vertex as visited.

Choose an unvisited adjacent vertex. If there is one, move to that vertex, mark it as visited, and add it to the cycle.

If all vertices have been visited and the last vertex has an edge to the starting vertex, return the cycle as a Hamiltonian cycle.

If the current vertex has no unvisited neighbors, backtrack to the previous vertex and mark the current vertex as unvisited.

Continue backtracking until you find an unexplored neighbor or all possibilities are exhausted.

If no Hamiltonian cycle is found, backtrack to the previous vertex and explore other paths.

Repeat the process for all vertices as a starting point.

If no Hamiltonian cycle is found for any starting vertex, conclude that the graph does not contain a Hamiltonian cycle.

#CODING:

```
#include <iostream>

#include <vector>

using namespace std;

const int V = 5; // Number of vertices

bool isSafe(int v, int pos, const vector<int>& path, const vector<vector<int>>& graph) {

    if (!graph[path[pos - 1]][v])

        return false;
    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;
    return true;
}

bool hamiltonianCycleUtil(const vector<vector<int>>& graph, vector<int>& path, int pos) {
    if (pos == V) {
        if (graph[path[pos - 1]][path[0]]) // Check if there's an edge from the last to the first vertex
            return true;
        return false;
    }

    for (int v = 1; v < V; v++) {
        if (isSafe(v, pos, path, graph)) {
            path[pos] = v;
            if (hamiltonianCycleUtil(graph, path, pos + 1))
                return true;
            path[pos] = -1; // Backtrack
        }
    }

    return false;
}

bool findHamiltonianCycle(const vector<vector<int>>& graph) {
    vector<int> path(V, -1);
    path[0] = 0; // Start from the first vertex

    if (!hamiltonianCycleUtil(graph, path, 1)) {
        cout << "No Hamiltonian cycle exists" << endl;
        return false;
    }

    cout << "Hamiltonian Cycle: ";
    for (int v : path)
        cout << v << " ";
    cout << path[0] << endl;

    return true;
}

int main() {
    vector<vector<int>> graph = {
        {0, 1, 1, 1, 0},
        {1, 0, 1, 0, 1},
        {1, 1, 0, 1, 0},
        {1, 0, 1, 0, 1},
        {0, 1, 0, 1, 0}
    };

    findHamiltonianCycle(graph);

    return 0;
}
```

19.

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

bool unassignedcell(int grid[][9], int &row, int &col)
```

```
{
    for (row = 0; row < 9; row++)
    {
        for (col = 0; col < 9; col++)
        {
            if (grid[row][col] == 0)
            {
                return true;
            }
        }
    }
    return false;
}

bool checkrow(int grid[][9], int row, int num)
{
    for (int j = 0; j < 9; j++)
    {
        if (grid[row][j] == num)
            return false;
    }
    return true;
}

bool checkcol(int grid[][9], int col, int num)
{
    for (int i = 0; i < 9; i++)
    {
        if (grid[i][col] == num)
            return false;
    }
    return true;
}

bool checksub(int grid[][9], int row, int col, int num)
{
    for (int i = row; i < (row + 3); i++)
    {
        for (int j = col; j < (col + 3); j++)
        {
            if (grid[i][j] == num)
            {
                return false;
            }
        }
    }
    return true;
}

bool isSafe(int grid[][9], int row, int col, int num)
{
    return checkrow(grid, row, num) &&
           checkcol(grid, col, num) &&
           checksub(grid, row - row % 3, col - col % 3, num) &&
           (grid[row][col] == 0);
}

bool sudo(int grid[][9])
{
    int row, col;
    if (!unassignedcell(grid, row, col))
        return true;
    for (int num = 1; num <= 9; num++)
    {
        if (isSafe(grid, row, col, num))
        {
            grid[row][col] = num;
            if (sudo(grid))
            {
                return true;
            }
            grid[row][col] = 0;
        }
    }
    return false;
}

int main()
{
    int n = 9;
    int grid[9][9];

    for (int i = 0; i < 9; i++)
    {
        for (int j = 0; j < 9; j++)
        {
            cin >> grid[i][j];
        }
    }
    if (sudo(grid))
    {
        for (int i = 0; i < 9; i++)
        {
            for (int j = 0; j < 9; j++)
            {
                cout << grid[i][j] << " ";
            }
            cout << endl;
        }
    }
    else
    {
        cout << "No solution exists." << endl;
    }
}
```

20.

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

bool is_prime(int i) {
    if (i <= 1) {
        return false;
    }
    for (int j = 2; j * j <= i; j++) {
        if (i % j == 0) {
            return false;
        }
    }
    return true;
}
```

```
vector<int> primes;

void find_primes_with_sum(int s, int n, int p, int current_sum, int current_index) {
    if (current_sum == s && primes.size() == n) {
        for (int i = 0; i < primes.size(); i++) {
            cout << primes[i];
            if (i != primes.size() - 1) {
                cout << " ";
            }
        }
        cout << endl;
        return;
    }
    if (current_sum > s || current_index > p + s || primes.size() >= n) {
        return;
    }
    if (is_prime(current_index)) {
        primes.push_back(current_index);
        find_primes_with_sum(s, n, p, current_sum + current_index, current_index + 1);
        primes.pop_back();
    }
    find_primes_with_sum(s, n, p, current_sum, current_index + 1);
}

int main() {
    int s, n, p;
    cout << "Enter the values of S, N, and P: ";
    cin >> s >> n >> p;

    find_primes_with_sum(s, n, p, 0, p + 1);
    return 0;
```

}

21.

The Minimum Cost Path Problem is a well-known optimization problem in computer science and mathematics. It involves finding the path through a grid, graph, or any network from a starting point to a destination while minimizing the sum of costs associated with traversing the path. This problem is commonly solved using dynamic programming.

Example: Consider a grid representing a terrain with each cell having a cost value associated with it. You need to find the path from the top-left corner to the bottom-right corner with the minimum cumulative cost. You can move either right or down in each step.

Grid with costs:

3 2 3

4 1 1

5 2 2

Using Dynamic Programming:

Dynamic programming can be applied to solve the Minimum Cost Path Problem efficiently. The key insight is that the minimum cost path from the start point to any cell (i, j) can be expressed as the minimum of the cost of coming from the cell above (i-1, j) and the cell to the left (i, j-1) plus the cost of the current cell (i, j).

Time and Space Complexity:

Time Complexity: The dynamic programming approach runs in O(m*n) time, where m and n are the dimensions of the grid. It needs to compute the minimum cost for each cell in the grid, and each computation takes constant time.

Space Complexity: The space complexity is also O(m*n) because it uses a 2D array of the same size to store the intermediate results.

Comparison to Alternative Methods:

Alternative methods, such as depth-first search (DFS) or breadth-first search (BFS), can also be used to find the minimum cost path in a grid. However, these methods are less efficient for larger grids because they explore a much larger search space and may require backtracking.