DESIGN AND ANALYSIS OF ALGORITHMS LAB

1. **Write a program to find the maximum and minimum element from the collection of elements using divide and conquer technique.**

DESCRIPTION:

The divide and conquer approach is a problem-solving technique that divides a problem into smaller subproblems, solves each subproblem recursively, and then combines the solutions to form the final result. This approach works well for problems that can be broken down into independent subproblems, such as sorting algorithms (like Merge Sort) and searching algorithms (like Binary Search).

1. **Divide: The problem is divided into smaller subproblems.**
2. **Conquer: Each subproblem is solved recursively.**
3. **Combine: The results from the subproblems are combined to get the final answer.**

Finding Min and Max using Divide and Conquer

To find the minimum and maximum in a collection of elements:

1. **Divide the array into two halves.**
2. **Conquer by recursively finding the minimum and maximum of each half, comparing each pair of elements when the array has two elements.**
3. **Combine the results by comparing the min and max of each half and selecting the overall min and max.**

PROGRAM:

```
#include <stdio.h>

void minmax(int A[], int i, int j, int *max, int *min) {
   int mid, max1, min1;

   // Base case: only one element
   if (i == j) {
      *max = *min = A[i];
   }
   // Base case: only two elements
   else if (j == i + 1) {
      if (A[i] < A[j]) {
         *max = A[j];
         *min = A[i];
      } else {
         *max = A[i];
         *min = A[j];
      }
```

```
    }
    // Recursive case
    else {
        mid = (i + j) / 2;
        minmax(A, i, mid, max, min);
        minmax(A, mid + 1, j, &max1, &min1);

        if (*max < max1) {
            *max = max1;
        }
        if (*min > min1) {
            *min = min1;
        }
    }
}

int main() {
    int n, i;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int A[n];
    printf("Enter the elements: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &A[i]);
    }

    int max, min;
    minmax(A, 0, n - 1, &max, &min);

    printf("Minimum element: %d\n", min);
    printf("Maximum element: %d\n", max);

    return 0;
}
```

OUTPUT:

```
Enter the number of elements in the array: 7
Enter the elements of the array: 34 15 12 5 125 78 1
Minimum element is 1
Maximum element is 125
```

2. **Write a program to find the optimal profit of a Knapsack using Greedy method.**

Description:
The greedy method is a problem-solving technique that builds a solution by choosing the best option available at each step. This approach is fast and efficient for problems where a locally optimal choice at each step leads to a globally optimal solution. One classic example where the greedy method works well is the fractional Knapsack problem.
Solving the Knapsack Problem using the Greedy Method
The Knapsack problem involves selecting items with given weights and profits to maximize profit within a fixed weight capacity of the knapsack. In the fractional Knapsack problem, where items can be taken in fractions, the greedy approach is effective:

1. **Calculate the Profit-to-Weight Ratio: For each item, calculate its profit-to-weight ratio (profit/weight), which indicates the item's value per unit of weight.**
2. **Sort Items by Ratio: Sort all items in descending order of their profit-to-weight ratio. This way, the items that offer the highest profit per unit weight are prioritized.**
3. **Select Items:**
    - **Start adding items to the knapsack by selecting them in this sorted order.**
    - **If the knapsack's remaining capacity can hold the entire item, add it fully and reduce the capacity by the item's weight.**
    - **If an item's weight exceeds the remaining capacity, take only the fraction that fits, adding the corresponding fraction of the profit.**
4. **Combine to Find Total Profit: The total profit is calculated by adding the profit of each fully or partially added item. This gives the optimal profit for the knapsack with the given capacity.**

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

void greedyKnapsack(int n, float weights[], float profits[], float capacity) {
   float x[n];  // Solution vector
   float totalProfit = 0;
   int i;
   float currentCapacity = capacity;

   // Initialize solution vector
   for (i = 0; i < n; i++) {
      x[i] = 0;
   }

   // Greedy approach
   for (i = 0; i < n; i++) {
      if (weights[i] > currentCapacity) {
         break;
      }
      x[i] = 1;
      currentCapacity -= weights[i];
```

```
        totalProfit += profits[i];
    }

    if (i < n) {
        x[i] = currentCapacity / weights[i];
        totalProfit += x[i] * profits[i];
    }

    // Print the solution vector and total profit
    printf("Solution vector: ");
    for (i = 0; i < n; i++) {
        printf("%f ", x[i]);
    }
    printf("\nTotal profit: %f\n", totalProfit);
}

int main() {
    int n;  // Number of objects
    float capacity;  // Capacity of knapsack

    // Input the number of objects
    printf("Enter the number of objects: ");
    scanf("%d", &n);

    float weights[n], profits[n];

    // Input weights and profits
    printf("Enter the weights of the objects: ");
    for (int i = 0; i < n; i++) {
        scanf("%f", &weights[i]);
    }

    printf("Enter the profits of the objects: ");
    for (int i = 0; i < n; i++) {
        scanf("%f", &profits[i]);
    }

    // Input the capacity of the knapsack
    printf("Enter the capacity of the knapsack: ");
    scanf("%f", &capacity);

    // Call the greedy knapsack function
    greedyKnapsack(n, weights, profits, capacity);

    return 0;
}
```

OUTPUT:

```
Enter the number of objects: 6
Enter the weights of the objects: 2 3 5 7 1 4
Enter the profits of the objects: 10 5 15 7 6 18
Enter the capacity of the knapsack: 15
Solution vector: 1.000000 1.000000 1.000000 0.714286 0.000000 0.000000
Total profit: 35.000000
```

3. Write a program for Optimal Merge Patterns problem using Greedy Method.

Description:

What is the Greedy Method?

The greedy method is a problem-solving technique that makes the best possible choice at each step, hoping these local optimizations lead to a globally optimal solution. It is generally faster and simpler than other methods, like dynamic programming. However, it doesn't guarantee an optimal solution for every problem. The greedy approach is particularly effective for problems like minimum spanning trees, the fractional knapsack problem, and the Optimal Merge Pattern.

What is the Optimal Merge Pattern Problem?

The Optimal Merge Pattern problem involves merging a collection of files with different sizes in such a way that minimizes the total cost of merging. In this context, the "cost" is defined as the sum of the sizes of the two files being merged.

The process typically involves:

1. **Selecting the Two Smallest Files: Always merge the two smallest files first to keep costs low.**
2. **Using a Min-Heap: A priority queue helps efficiently find and manage the smallest files.**
3. **Accumulating Costs: The merging cost is added each time two files are combined until only one file remains.**

This problem is especially relevant in data processing and storage, where minimizing merge costs can enhance efficiency and reduce resource usage.

PROGRAM:

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

// Define the structure for a binary tree node
typedef struct Node {
    int weight;
    struct Node *left, *right;
} Node;

// Function to create a new node
Node* getNode(int weight) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->weight = weight;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Function to find the index of the node with the least weight
int findLeast(Node** list, int n) {
    int minIndex = 0;
    for (int i = 1; i < n; i++) {
        if (list[i] && list[i]->weight < list[minIndex]->weight) {
            minIndex = i;
        }
    }
    return minIndex;
}

// Function to remove the node with the least weight from the list
Node* extractLeast(Node** list, int* n) {
    int minIndex = findLeast(list, *n);
    Node* leastNode = list[minIndex];
    for (int i = minIndex; i < *n - 1; i++) {
        list[i] = list[i + 1];
    }
    (*n)--;
    return leastNode;
}

// Function to insert a node into the list
void insert(Node** list, Node* node, int* n) {
    list[*n] = node;
    (*n)++;
}

// Function to calculate the total merge cost
int calculateMergeCost(Node* root) {
    if (root == NULL) {
        return 0;
    }
    return root->weight + calculateMergeCost(root->left) + calculateMergeCost(root->right);
}

int main() {
    int n;
    printf("Enter the number of files: ");
    scanf("%d", &n);

    Node* list[n];
    printf("Enter the weights of the files: ");
    for (int i = 0; i < n; i++) {
        int weight;
        scanf("%d", &weight);
        list[i] = getNode(weight);
    }

    int listSize = n;
    while (listSize > 1) {
        Node* node = getNode(0);

        // Extract two nodes with the least weight
        node->left = extractLeast(list, &listSize);
        node->right = extractLeast(list, &listSize);

        // Merge their weights and create a new node
        node->weight = node->left->weight + node->right->weight;

        // Insert the new node back into the list
        insert(list, node, &listSize);
    }

    // The final node in the list is the root of the optimal merge pattern tree
    Node* root = list[0];

    // Calculate the total cost of the optimal merge pattern
    int totalMergeCost = calculateMergeCost(root) - root->weight;
    printf("Total merge cost: %d\n", totalMergeCost);

    return 0;
}
```

OUTPUT:

```
Enter the number of files: 10
Enter the weights of the files: 8 2 9 1 12 10 18 15 14 17
Total merge cost: 332
```

4. **write a program for single source shortest path for general weights using dynamic programming**

description:
Dynamic Programming (DP) is an algorithmic technique used to solve complex problems by breaking them down into simpler, overlapping subproblems. It is particularly effective for optimization problems, where the solution can be constructed from optimal solutions to smaller subproblems. DP improves efficiency by storing the results of these subproblems in a table or array, which avoids redundant calculations and significantly reduces computation time. The technique often employs a bottom-up approach, starting from the simplest cases and building up to the desired solution, making it suitable for a variety of applications, including shortest path problems, resource allocation, and more.
The Bellman-Ford algorithm employs dynamic programming principles to find the shortest path from a single source vertex to all other vertices in a graph, including those with negative weights.
The algorithm starts by initializing the distance from the source to zero and all other vertices to infinity. In each iteration, it relaxes all edges by checking if the current distance to a vertex can be improved. This process updates distances based on previously computed results.
The relaxation step is repeated for V−1V-1V−1 iterations (where VVV is the number of vertices) to ensure that all shortest paths are found. After these iterations, an additional pass checks for negative weight cycles, confirming the optimality of the solutions.
Overall, the Bellman-Ford algorithm exemplifies dynamic programming by systematically updating shortest path estimates, making it effective for both positive and negative weight graphs in optimization problems.
4o mini

Program:
```
#include <stdio.h>
#include <limits.h>

// Define a structure to represent an edge in the graph
struct Edge {
   int u, v, weight;
};

// Function to print the distances to each vertex after each iteration, including vertex and distance
from source
void printIteration(int dist[], int V, int iteration) {
   printf("Iteration %d:\n", iteration);
   printf("Vertex  Distance from Source\n");
   for (int i = 0; i < V; i++) {
      if (dist[i] == INT_MAX) {
         printf("%d\tINF\n", i);
      } else {
         printf("%d\t%d\n", i, dist[i]);
      }
   }
   printf("\n");
}
```

```
// Function to perform the Bellman-Ford algorithm
void BellmanFord(int V, int E, struct Edge edges[], int src) {
   int dist[V];

   // Step 1: Initialize distances from src to all other vertices as INFINITE
   for (int i = 0; i < V; i++) {
      dist[i] = INT_MAX;
   }
   dist[src] = 0;

   // Step 2: Relax all edges |V| - 1 times
   for (int i = 1; i <= V - 1; i++) {
      for (int j = 0; j < E; j++) {
         int u = edges[j].u;
         int v = edges[j].v;
         int weight = edges[j].weight;

         if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            dist[v] = dist[u] + weight;
         }
      }
      printIteration(dist, V, i); // Print distances after each iteration
   }

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

   // Print the final distances
   printf("\nFinal Distances:\n");
   printIteration(dist, V, V); // Final result
}

int main() {
   int V, E, src;

   // Take input from user for the number of vertices and edges
   printf("Enter the number of vertices: ");
   scanf("%d", &V);
   printf("Enter the number of edges: ");
   scanf("%d", &E);

   struct Edge edges[E];
```

```
    // Input the edges
    for (int i = 0; i < E; i++) {
        printf("Enter edge %d (u v weight): ", i + 1);
        scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].weight);
    }

    // Input the source vertex
    printf("Enter the source vertex: ");
    scanf("%d", &src);

    // Run the Bellman-Ford algorithm
    BellmanFord(V, E, edges, src);

    return 0;
}
```
OUTPUT:
```
Enter the number of vertices: 4
Enter the number of edges: 5
Enter edge 1 (u v weight): 0 1 1
Enter edge 2 (u v weight): 0 2 4
Enter edge 3 (u v weight): 1 2 -2
Enter edge 4 (u v weight): 1 3 2
Enter edge 5 (u v weight): 2 3 3
Enter the source vertex: 0

Iteration 1:
Vertex  Distance from Source
0       0
1       1
2       4
3       INF

Iteration 2:
Vertex  Distance from Source
0       0
1       1
2       -1
3       3

Iteration 3:
Vertex  Distance from Source
0       0
1       1
2       -1
3       2

Final Distances:
Iteration 4:
Vertex  Distance from Source
0       0
1       1
```

2	-1
3	2

5.write a program to find all pair shortest path from any node to any other node within a graph
Description:
Dynamic Programming (DP) is an algorithmic technique used to solve complex problems by breaking them down into simpler overlapping subproblems. It is particularly effective for optimization problems where the overall solution can be constructed from optimal solutions to smaller subproblems. DP improves efficiency by storing the results of these subproblems in a table or array, thereby avoiding redundant calculations. This approach is widely used in various fields, including operations research, economics, and computer science.
The Floyd-Warshall algorithm is a classic example of dynamic programming applied to finding the shortest paths between all pairs of vertices in a graph. It begins by initializing a distance matrix, where the distance from a vertex to itself is set to zero and the distances between other vertices are initialized to infinity, except for existing edges.
In each iteration, the algorithm updates the distance matrix by checking if a path through an intermediate vertex provides a shorter path between two vertices. This process is repeated for all vertices as intermediates. After processing, the distance matrix reflects the shortest path lengths between all pairs, making the Floyd-Warshall algorithm effective for dense graphs, including those with negative weights.

Program:
```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>

#define INF_STR "INF" // String representation of INF

// Utility function to print the matrix
void print_matrix(int **dist, int V, int iteration) {
  printf("\nMatrix A^%d:\n", iteration);
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
      if (dist[i][j] == INT_MAX)
        printf("%7s", "INF");
      else
        printf("%7d", dist[i][j]);
    }
    printf("\n");
  }
}

// Function to implement the all-pairs shortest path algorithm
void all_pairs_shortest_path(int **graph, int V) {
  int **dist;
  int i, j, k;

  // Step 1: Dynamically allocate memory for the solution matrix
  dist = (int **)malloc(V * sizeof(int *));
  for (i = 0; i < V; i++) {
    dist[i] = (int *)malloc(V * sizeof(int));
    }

    // Initialize the solution matrix the same as the input graph matrix
    for (i = 0; i < V; i++) {
        for (j = 0; j < V; j++) {
            dist[i][j] = graph[i][j];
        }
    }

    // Print initial matrix A^0 (before any iterations)
    print_matrix(dist, V, 0);

    // Step 2 and Step 3: Update the solution matrix iteratively using the formula
    for (k = 0; k < V; k++) {
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                // Update dist[i][j] considering vertex k as an intermediate vertex
                if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX && dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
        // Print the matrix after each iteration A^k
        print_matrix(dist, V, k + 1);
    }

    // Free allocated memory
    for (i = 0; i < V; i++) {
        free(dist[i]);
    }
    free(dist);
}

int main() {
    int V;
    int **graph;
    char input[100]; // Buffer to store input (whole row as a string)

    // Take input from user for the number of vertices
    printf("Enter the number of vertices: ");
    scanf("%d", &V);

    // Dynamically allocate memory for the adjacency matrix
    graph = (int **)malloc(V * sizeof(int *));
    for (int i = 0; i < V; i++) {
        graph[i] = (int *)malloc(V * sizeof(int));
    }

    // Clear buffer for newline character after previous input
    getchar();
    // Take row-wise input for the adjacency matrix
    printf("Enter the adjacency matrix row-wise (use 'INF' for infinity):\n");
    for (int i = 0; i < V; i++) {
        printf("Enter row %d: ", i + 1);
        fgets(input, sizeof(input), stdin); // Read the entire row as a string

        // Tokenize the row and process each element
        char *token = strtok(input, " ");
        for (int j = 0; j < V && token != NULL; j++) {
            // Check if token is "INF"
            if (strcmp(token, INF_STR) == 0 || strcmp(token, "INF\n") == 0) {
                graph[i][j] = INT_MAX; // Treat "INF" as INT_MAX
            } else {
                graph[i][j] = atoi(token); // Convert numeric string to integer
            }
            token = strtok(NULL, " "); // Move to the next token
        }
    }

    // Execute the APSP algorithm
    all_pairs_shortest_path(graph, V);

    // Free allocated memory
    for (int i = 0; i < V; i++) {
        free(graph[i]);
    }
    free(graph);

    return 0;
}
```

OUTPUT:

```
Enter the number of vertices: 5
Enter the adjacency matrix row-wise (use 'INF' for infinity):
Enter row 1: 0 3 INF 7 4
Enter row 2: 3 0 1 INF 5
Enter row 3: INF 1 0 2 7
Enter row 4: 7 INF 2 0 9
Enter row 5: INF 2 6 7 0

Matrix A^0:
   0    3  INF    7    4
   3    0    1  INF    5
 INF    1    0    2    7
   7  INF    2    0    9
 INF    2    6    7    0

Matrix A^1:
   0    3  INF    7    4
   3    0    1   10    5
 INF    1    0    2    7
   7   10    2    0    9
 INF     2     6     7     0

Matrix A^2:
   0     3     4     7     4
   3     0     1    10     5
   4     1     0     2     6
   7    10     2     0     9
   5     2     3     7     0

Matrix A^3:
   0     3     4     6     4
   3     0     1     3     5
   4     1     0     2     6
   6     3     2     0     8
   5     2     3     5     0

Matrix A^4:
   0     3     4     6     4
   3     0     1     3     5
   4     1     0     2     6
   6     3     2     0     8
   5     2     3     5     0

Matrix A^5:
   0     3     4     6     4
   3     0     1     3     5
   4     1     0     2     6
   6     3     2     0     8
   5     2     3     5     0
```

6. Write a program to find the non-attacking positions of Queens in a given chess board using backtracking.

Description:

Backtracking is an algorithmic technique used for solving problems incrementally by trying partial solutions and abandoning them if they are determined to be invalid or not optimal. It systematically explores all possible configurations in a search space, making choices one at a time and backtracking whenever it encounters a conflict or when a potential solution cannot be completed. This method is particularly useful in combinatorial problems, where it helps find all possible solutions or the best solution by efficiently pruning the search space.

The N-Queens problem is a classic example where backtracking is applied. It involves placing N queens on an N×N chessboard such that no two queens threaten each other. Specifically, this means ensuring that no two queens share the same row, column, or diagonal.

To solve the N-Queens problem using backtracking, the algorithm starts by attempting to place a queen in the first row and proceeds to subsequent rows. For each row, it checks each column for a safe position and places a queen if the position is valid. If placing a queen leads to a configuration where no further queens can be placed safely, the algorithm backtracks by removing the last placed queen and trying the next column. This process continues until all valid configurations are explored, yielding all non-attacking arrangements of queens on the chessboard.

4o mini

PROGRAM:

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

int *x;  // Array to store the column positions of queens

// Function to check if placing a queen in row k and column i is safe
int place(int k, int i) {
    for (int j = 1; j < k; j++) {
        // Check if another queen is in the same column or diagonal
        if (x[j] == i || abs(x[j] - i) == abs(j - k))
            return 0;
    }
    return 1;
}

// Function to solve the N-Queens problem using backtracking
void NQUEENS(int k, int n) {
    for (int i = 1; i <= n; i++) {
        if (place(k, i)) {
            x[k] = i;  // Place the queen in row k and column i
            if (k == n) {
                // Print the solution
                for (int j = 1; j <= n; j++) {
```

```
                printf("%d ", x[j]);
            }
            printf("\n");
        } else {
            NQUEENS(k + 1, n);  // Recur for the next row
        }
      }
    }
}

int main() {
    int n;

    // Take input from the user
    printf("Enter the number of queens (N): ");
    scanf("%d", &n);

    // Allocate memory for the array to store queen positions
    x = (int *)malloc((n + 1) * sizeof(int));

    // Solve the N-Queens problem starting from the first row
    printf("The possible solutions are:\n");
    NQUEENS(1, n);

    // Free the dynamically allocated memory
    free(x);

    return 0;
}
```

OUTPUT:

```
Enter the number of queens (N): 4
The possible solutions are:
2 4 1 3
3 1 4 2

Enter the number of queens (N): 8
The possible solutions are:
1 5 8 6 3 7 2 4
1 6 8 3 7 4 2 5
2 5 7 4 1 8 6 3
2 6 1 7 4 8 3 5
3 1 7 5 8 2 4 6
3 5 2 8 1 7 4 6
4 1 5 8 2 7 3 6
4 1 5 8 6 3 7 2
5 1 4 6 8 2 7 3
5 1 8 4 2 7 3 6
```

7. Find a subset of a given set S = {Sl, S2, ..., Sn} of n positive integers, whose sum is equal to a given positive integer d. For example, if S= {1, 2, 5, 6, 8} and d = 9 there are two solutions {1, 2, 6} and {1, 8}. A suitable message is to be displayed if the given problem instance doesn't have a solution.

Description:

Backtracking is an algorithmic technique that involves incrementally building candidate solutions and abandoning those that fail to satisfy the problem's constraints. It systematically explores all possible configurations in a search space, making choices step by step. If a candidate solution cannot be completed or is found to be invalid, the algorithm backtracks to the previous step and tries a different path. This method is particularly effective for solving combinatorial problems and optimization issues, where it helps identify all potential solutions or the optimal one by efficiently reducing the search space.

The Sum of Subsets problem is a classic example of using backtracking. The problem requires finding all subsets of a given set of integers that sum up to a specified target value. For instance, given a set of numbers and a target sum, the goal is to identify which combinations of the numbers can be added together to equal that target.

To solve this problem using backtracking, the algorithm starts with an empty subset and explores the inclusion of each element from the set. It recursively adds elements to the current subset while keeping track of the current sum. If the current sum equals the target, the subset is recorded as a valid solution. If the sum exceeds the target, the algorithm backtracks by removing the last added element and exploring other combinations. This process continues until all possible subsets are examined, effectively identifying all combinations that meet the target sum condition.

4o mini

PROGRAM:

```
#include <stdio.h>

void findSubsets(int arr[], int n, int target, int index, int currentSum, int subset[], int subsetSize) {
    // Check for success: if currentSum equals the target
    if (currentSum == target) {
        printf("{ ");
        for (int i = 0; i < subsetSize; i++) {
            printf("%d ", subset[i]);
        }
        printf("}\n");
        return;
    }

    // Check for dead ends: if currentSum exceeds target or no more elements left
    if (currentSum > target || index >= n) {
        return;
    }

    // Include the current element
    subset[subsetSize] = arr[index]; // Add current element to the subset
    findSubsets(arr, n, target, index + 1, currentSum + arr[index], subset, subsetSize + 1);

    // Exclude the current element
    findSubsets(arr, n, target, index + 1, currentSum, subset, subsetSize);
}

int main() {
    int n, target;

    // Get the number of elements in the set from the user
    printf("Enter the number of elements in the set: ");
    scanf("%d", &n);

    int arr[n]; // Declare array of size n

    // Get the elements of the set from the user
    printf("Enter the elements of the set: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Get the target sum from the user
    printf("Enter the target sum: ");
    scanf("%d", &target);

    printf("Subsets with sum %d are:\n", target);
    int subset[n]; // Temporary array to hold current subset

    // Call the function to find subsets
    findSubsets(arr, n, target, 0, 0, subset, 0);

    // Check if no subset found and display a suitable message
    if (target == 0) { // If target is 0, there are always subsets (the empty subset)
        printf("{ }\n"); // The empty subset is a valid solution
    }

    return 0;
}
```

OUTPUT:

```
Enter the number of elements in the set: 7
Enter the elements of the set: 15 7 20 5 18 10 12
Enter the target sum: 35
Subsets with sum 35 are:
{ 15 20 }
{ 7 18 10 }
{ 20 5 10 }
{ 5 18 12 }
```

8. Write a program to color the nodes in a given graph such that no two adjacent can have the same color using backtracking.

Description:

Backtracking

Backtracking is an algorithmic technique used to solve problems by exploring all possible solutions and eliminating those that fail to meet certain criteria. It involves a recursive approach where the algorithm incrementally builds a solution and backtracks when it detects that a particular path cannot lead to a valid solution. This method is particularly effective for problems involving combinatorial search, such as puzzles and optimization challenges, where the solution space can be large. By pruning invalid paths early, backtracking can often find solutions more efficiently than brute-force methods.

Graph Coloring

Graph coloring is a problem in computer science where the objective is to assign colors to the vertices of a graph such that no two adjacent vertices share the same color. This is represented mathematically by a graph composed of vertices (nodes) and edges (connections). The minimum number of colors needed to achieve this is called the graph's chromatic number. Graph coloring has practical applications in various fields, including scheduling, map coloring, and resource allocation, where it helps prevent conflicts and optimize the use of limited resources. Solving graph coloring problems often involves techniques like backtracking, as finding an optimal solution can be computationally complex.

Program:

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

int n, m;      // n = number of vertices, m = number of colors
int **graph;   // Dynamically allocated adjacency matrix
int *colors;   // Dynamically allocated array for storing colors of vertices

void nextValue(int k) {
   do {
      colors[k] = (colors[k] + 1) % (m + 1);  // Assign next color

      if (colors[k] == 0) return;  // No valid color available

      // Check for conflicts with adjacent vertices
      int conflict = 0;
      for (int j = 1; j <= n; j++) {
         if (graph[k][j] && colors[k] == colors[j]) {  // Conflict found
            conflict = 1;
            break;
         }
      }
      if (!conflict) return;  // Valid color found
   } while (1);
}

void m_coloring(int k) {
   while (1) {
      nextValue(k);
```

```
        if (colors[k] == 0) return;  // No valid color found for this vertex

        if (k == n) {  // If all vertices are colored
            printf("Solution: ");
            for (int i = 1; i <= n; i++) {
                printf("%d ", colors[i]);  // Output the colors assigned to each vertex
            }
            printf("\n");
        } else {
            m_coloring(k + 1);  // Move to the next vertex
        }
    }
}

int main() {
    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    graph = (int **)malloc((n + 1) * sizeof(int *));
    colors = (int *)malloc((n + 1) * sizeof(int));

    printf("Enter the adjacency matrix (1 for edge, 0 for no edge):\n");
    for (int i = 1; i <= n; i++) {
        graph[i] = (int *)malloc((n + 1) * sizeof(int));
        for (int j = 1; j <= n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    printf("Enter the number of colors: ");
    scanf("%d", &m);

    // Check if the number of colors is less than 3
    if (m < 3) {
        printf("Not possible to color the graph with less than 3 colors.\n");
        goto cleanup;  // Jump to cleanup code
    }

    for (int i = 1; i <= n; i++) {
        colors[i] = 0;  // Initialize the color array with 0 (no color assigned)
    }

    m_coloring(1);  // Start the coloring from the first vertex

cleanup:
    for (int i = 1; i <= n; i++) {
        free(graph[i]);  // Free allocated memory for each row
    }
    free(graph);  // Free the graph array
    free(colors); // Free the colors array
    return 0;
}
```

OUTPUT:

```
Enter the number of vertices: 6
Enter the adjacency matrix (1 for edge, 0 for no edge):
0 1 1 0 1 0
1 0 1 1 0 1
1 1 0 1 1 0
0 1 1 0 0 1
1 0 1 0 0 1
0 1 0 1 1 0
Enter the number of colors: 3
Solution: 1 2 3 1 2 3
Solution: 1 3 2 1 3 2
Solution: 2 1 3 2 1 3
Solution: 2 3 1 2 3 1
Solution: 3 1 2 3 1 2
Solution: 3 2 1 3 2 1

Enter the number of vertices: 4
Enter the adjacency matrix (1 for edge, 0 for no edge):
0 1 1 0
1 0 1 1
1 1 0 1
0 1 1 0
Enter the number of colors: 2
Not possible to color the graph with less than 3 colors.
```

9. Design and implement to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using Backtracking principle.

Description:

Backtracking

Backtracking is an algorithmic technique used for solving complex problems by incrementally exploring potential solutions and discarding those that fail to meet specific criteria. It works recursively, building a solution step by step and reverting when it encounters a conflict or dead end. This method is particularly effective for problems involving combinatorial searches, such as puzzles and optimization challenges, where the solution space can be vast. By pruning invalid paths early in the process, backtracking can significantly enhance efficiency compared to brute-force approaches.

Hamiltonian Cycle

A Hamiltonian cycle is a specific path in a graph that visits each vertex exactly once before returning to the starting point. Determining the existence of such a cycle is known as the Hamiltonian cycle problem, which is classified as NP-complete. This means that no known polynomial-time algorithm can solve it for all graphs efficiently. Hamiltonian cycles are applicable in fields such as operations research and network routing. Backtracking is often employed to address this problem, allowing for a systematic exploration of potential cycles while effectively managing the complexities of the search space by retracing steps when necessary.

Program:

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

int n;            // Number of vertices
int **graph;      // Adjacency matrix for the graph
int *path;        // Array to store the current path for the Hamiltonian cycle

void printCycle() {
   printf("Hamiltonian Cycle: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", path[i]);
   }
   printf("%d\n", path[0]); // Close the cycle
}

int isValidVertex(int k) {
   // Check if there is an edge and if the vertex is not visited
   if (graph[path[k - 1]][path[k]] &&
      !memchr(path, path[k], k * sizeof(int))) {
      return 1; // Valid vertex
   }
   return 0; // Invalid vertex
}

void hamiltonianCycle(int k) {
   if (k == n) {
      // Check if there is an edge back to the starting vertex
      if (graph[path[n - 1]][path[0]]) {
         printCycle();
```

```
        }
        return;
    }

    for (path[k] = 1; path[k] <= n; path[k]++) {
        if (isValidVertex(k)) {
            hamiltonianCycle(k + 1); // Move to the next vertex
        }
    }
}

int main() {
    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    graph = (int **)malloc((n + 1) * sizeof(int *));
    for (int i = 0; i <= n; i++) {
        graph[i] = (int *)malloc((n + 1) * sizeof(int));
    }

    printf("Enter the adjacency matrix (1 for edge, 0 for no edge):\n");
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    path = (int *)malloc(n * sizeof(int));
    path[0] = 1; // Start from vertex 1

    hamiltonianCycle(1); // Start finding the Hamiltonian cycle

    // Free allocated memory
    for (int i = 0; i <= n; i++) {
        free(graph[i]);
    }
    free(graph);
    free(path);

    return 0;
}
```

OUTPUT:

```
Enter the number of vertices: 7
Enter the adjacency matrix (1 for edge, 0 for no edge):
0 1 1 1 0 0 0
1 0 0 1 1 0 1
1 0 0 0 0 1 0
1 1 0 0 0 1 1
0 1 0 0 0 0 1
0 0 1 1 0 0 1
```

```
0 1 0 1 1 1 0
Hamiltonian Cycle: 1 2 5 7 4 6 3 1
Hamiltonian Cycle: 1 3 6 4 7 5 2 1
Hamiltonian Cycle: 1 3 6 7 5 2 4 1
Hamiltonian Cycle: 1 4 2 5 7 6 3 1
```