B V C COLLEGE OF ENGINEERING
(An Autonomous Institution)
(Approved by AICTE, New Delhi, Permanently affiliated to JNTUK, KAKINADA)
Palacharla, Rajamahendravaram-533102
Mobile No: 9704578666, 9948027756, E-mail: bvcr@bvcgroup.in, Website: www.bvcr.edu.in

Program	CSE (ARTIFICIAL INTELLIGENCE and MACHINE LEARNING	A.Y	2024-25
Course Name	Algorithms for Efficient Coding Lab	Course Code	PC
Faculty Name	Dr Y Venkat	Class &Sem	III &II
Course Coordinator	Dr Y Venkat	Regulations	R20

Course Objective:

- To develop efficient coding for the algorithms with various inputs and algorithms

Course Outcomes:

By completing the course the students will be able to:

- Analyze the program execution time

List of Experiments:

1. Develop a program and measure the running time for Binary Search with Divide and Conquer
2. Develop a program and measure the running time for Merge Sort with Divide and Conquer
3. Develop a program and measure the running time for Quick Sort with Divide and Conquer
4. Develop a program and measure the running time for estimating minimum-cost spanning Trees with Greedy Method
5. Develop a program and measure the running time for estimating Single Source Shortest Paths with Greedy Method
6. Develop a program and measure the running time for optimal Binary search trees with Dynamic Programming
7. Develop a program and measure the running time for identifying solution for traveling salesperson problem with Dynamic Programming
8. Develop a program and measure the running time for identifying solution for 8-Queens problem with Backtracking
9. Develop a program and measure the running time for Graph Coloring with Backtracking
10. Develop a program and measure the running time to generate solution of Hamiltonian Cycle problem with Backtracking
11. Develop a program and measure the running time running time to generate solution of Knapsack problem with Backtracking

1.Develop a program and measure the running time for Binary Search with Divide and Conquer

Here’s a **C program** to implement **Binary Search using Divide and Conquer** and measure its **running time** using the clock() function from the time.h library.

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

// Function to perform binary search using divide and conquer approach
int binarySearch(int arr[], int low, int high, int key) {
    if (low <= high) {
        int mid = low + (high - low) / 2;

        // Check if the key is present at mid
        if (arr[mid] == key)
            return mid;

        // If key is smaller than mid, search the left sub-array
        if (key < arr[mid])
            return binarySearch(arr, low, mid - 1, key);

        // Else search the right sub-array
        return binarySearch(arr, mid + 1, high, key);
    }
    return -1; // Key not found
}

int main() {
    int n, key, result;
    clock_t start, end;
    double cpu_time_used;

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

    // Allocate memory for the array
    int *arr = (int *)malloc(n * sizeof(int));

    // Input sorted array elements
    printf("Enter %d sorted elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Input the key to be searched
    printf("Enter the key to search: ");
    scanf("%d", &key);
```

```
    // Start time measurement
    start = clock();

    // Call the binary search function
    result = binarySearch(arr, 0, n - 1, key);

    // End time measurement
    end = clock();

    // Calculate time taken in seconds
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

    // Display the result
    if (result != -1)
        printf("Key %d found at index %d.\n", key, result);
    else
        printf("Key %d not found in the array.\n", key);

    printf("Time taken for Binary Search: %f seconds.\n", cpu_time_used);

    // Free allocated memory
    free(arr);

    return 0;
}
```

Explanation of Key Components

1. **Binary Search (Divide and Conquer)**
    - The array is divided into two halves.
    - The middle element is compared with the key.
    - If the key is less than the middle element, the search continues in the left half.
    - If the key is greater than the middle element, the search continues in the right half.
2. **Time Measurement**
    - clock_t start, end; to store the time at the start and end of the binary search.
    - cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC; calculates the time difference in seconds.
3. **Input/Output**
    - Input: Size of array n, sorted array elements, and the key to search.
    - Output: Position of the key (index) in the array and the total time taken for the binary search.

INPUT

Enter the size of the array: 10
Enter 10 sorted elements: 2 3 4 10 40 50 60 70 80 90

Enter the key to search: 40

OUT PUT
Key 40 found at index 4.
Time taken for Binary Search: 0.000002 seconds.

🕒 Time Complexity Analysis

- **Best Case**: O(1)O(1)O(1) when the middle element is the key.
- **Average Case**: O(logn)O(\log n)O(logn) since the problem size is divided by 2 at every step.
- **Worst Case**: O(logn)O(\log n)O(logn

Possible Enhancements

1. **Dynamic Input Handling**: Allow for unsorted input and apply sorting before binary search.
2. **Iterative Binary Search**: Use an iterative version of binary search to avoid stack overhead.
3. **Multiple Runs**: Measure the time for multiple iterations and compute the average.

This program efficiently demonstrates **Binary Search using Divide and Conquer** and also provides a **real-time measure of its execution time**. Let me know if you'd like any modifications or explanations for any part of the code

2.Develop a program and measure the running time for Merge Sort with Divide and Conquer

Here’s a **C program** to implement **Merge Sort using Divide and Conquer** and measure its **running time** using the clock() function from the time.h library

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

// Function to merge two subarrays of arr[]
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1; // Size of left subarray
    int n2 = right - mid;    // Size of right subarray

    // Create temporary arrays
    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

    // Copy data to temp arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge the temp arrays back into arr[left...right]
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }

    // Copy the remaining elements of L[], if any
    while (i < n1)
        arr[k++] = L[i++];

    // Copy the remaining elements of R[], if any
    while (j < n2)
        arr[k++] = R[j++];
```

```
    // Free the dynamically allocated memory
    free(L);
    free(R);
}

// Function to perform merge sort using divide and conquer approach
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

int main() {
    int n;
    clock_t start, end;
    double cpu_time_used;

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

    // Allocate memory for the array
    int *arr = (int *)malloc(n * sizeof(int));

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

    // Start time measurement
    start = clock();

    // Call merge sort
    mergeSort(arr, 0, n - 1);

    // End time measurement
    end = clock();

    // Calculate time taken in seconds
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
```

```
    // Display the sorted array
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    printf("Time taken for Merge Sort: %f seconds.\n", cpu_time_used);

    // Free allocated memory
    free(arr);

    return 0;
}
```

Explanation of Key Components

1. **Merge Sort (Divide and Conquer)**
   - **Divide**: The array is divided into two halves.
   - **Conquer**: Each half is sorted recursively.
   - **Combine**: The two sorted halves are merged to produce the final sorted array.
2. **Time Measurement**
   - clock_t start, end; to store the start and end time of the merge sort.
   - cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC; calculates the time difference in seconds.
3. **Input/Output**
   - Input: Size of array n and array elements.
   - Output: Sorted array and the total time taken for the merge sort.
4. **Memory Management**
   - Temporary arrays L and R are dynamically allocated using malloc() to store subarrays.
   - After merging, memory is freed using free(L) and free(R) to avoid memory leaks.

INPUT

Enter the size of the array: 10
Enter 10 elements: 38 27 43 3 9 82 10 5 11 12

OUT PUT

Sorted array: 3 5 9 10 11 12 27 38 43 82
Time taken for Merge Sort: 0.000123 seconds.

🕒 Time Complexity Analysis

- **Best Case**: O(nlogn)O(n \log n)O(nlogn)
- **Average Case**: O(nlogn)O(n \log n)O(nlogn)
- **Worst Case**: O(nlogn)O(n \log n)O(nlogn) since the array is always divided into two halves and merged.

Possible Enhancements

1. **Dynamic Input Handling**: Generate random numbers instead of user input for large arrays.
2. **Measure Multiple Runs**: Run the program multiple times and calculate the average execution time.
3. **Use Iterative Merge Sort**: Implement an iterative version of merge sort to avoid recursion overhead.

This program efficiently demonstrates **Merge Sort using Divide and Conquer** and also provides a **real-time measure of its execution time**. Let me know if you'd like any modifications or explanations for any part of the code

3.Develop a program and measure the running time for Quick Sort with Divide and Conquer

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

// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function to divide the array
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Pivot element
    int i = (low - 1); // Index of smaller element

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++; // Increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

```
// QuickSort function using divide and conquer
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

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

    int *arr = (int *)malloc(n * sizeof(int));
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Original array: \n");
    printArray(arr, n);

    clock_t start, end;
    double cpu_time_used;

    start = clock(); // Start time measurement
    quickSort(arr, 0, n - 1);
    end = clock(); // End time measurement

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC; // Calculate total time taken

    printf("\nSorted array: \n");
    printArray(arr, n);

    printf("\nTime taken to sort the array using Quick Sort: %f seconds\n", cpu_time_used);

    free(arr);
    return 0;
}
```

Explanation of the Key Components

1. **Input Handling**:
   - The program reads the size of the array n and the elements of the array from the user.
2. **Quick Sort Algorithm**:
   - The pivot is chosen as the last element of the array.
   - The partition() function rearranges elements so that those less than the pivot are on the left, and those greater are on the right.
   - The quickSort() function recursively sorts the left and right sub-arrays.
3. **Time Measurement**:
   - The clock() function captures the time before and after sorting.
   - The execution time is calculated as (end−start)/CLOCKS_PER_SEC(\text{end} - \text{start}) / CLOCKS_PER_SEC(end−start)/CLOCKS_PER_SEC.
4. **Output**:
   - The program prints the original array, the sorted array, and the total time taken for sorting.

Enter the number of elements in the array: 5
Enter the elements of the array:
12 4 5 6 2

Original array:
12 4 5 6 2

Sorted array:
2 4 5 6 12

Time taken to sort the array using Quick Sort: 0.000123 seconds

Customizations

- **Dynamic Array Size**: Allows users to input any size of the array.
- **Random Input**: To test with larger inputs, you can generate random arrays.
- **Measure for Larger Inputs**: Run tests with larger arrays to see how the time increases.

4.Develop a program and measure the running time for estimating minimum-cost spanning Trees with Greedy Method

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

#define MAX 100

// Structure to represent an edge in the graph
```

```
struct Edge {
    int src, dest, weight;
};

// Structure to represent a connected, undirected, and weighted graph
struct Graph {
    int V, E; // Number of vertices and edges
    struct Edge* edge; // Array of edges
};

// Structure to represent a subset for union-find
struct Subset {
    int parent;
    int rank;
};

// Function to create a graph with V vertices and E edges
struct Graph* createGraph(int V, int E) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edge = (struct Edge*)malloc(E * sizeof(struct Edge));
    return graph;
}

// A utility function to find set of an element i (uses path compression technique)
int find(struct Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

// A function that does union of two sets of x and y (uses union by rank)
void Union(struct Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare function to sort edges in non-decreasing order of weight
int compareEdges(const void* a, const void* b) {
    struct Edge* a1 = (struct Edge*)a;
```

```
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// Function to construct Minimum Spanning Tree using Kruskal's algorithm
void KruskalMST(struct Graph* graph) {
    int V = graph->V;
    struct Edge result[MAX]; // Store the resulting MST
    int e = 0; // Index variable for result[]
    int i = 0; // Index variable for sorted edges

    // Step 1: Sort all the edges in non-decreasing order of their weight
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), compareEdges);

    // Allocate memory for creating V subsets
    struct Subset* subsets = (struct Subset*)malloc(V * sizeof(struct Subset));

    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    while (e < V - 1 && i < graph->E) {
        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);

    free(subsets);
}

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

    struct Graph* graph = createGraph(V, E);
```

```
    printf("Enter the source, destination and weight of each edge:\n");
    for (int i = 0; i < E; i++) {
        printf("Edge %d: ", i + 1);
        scanf("%d %d %d", &graph->edge[i].src, &graph->edge[i].dest, &graph->edge[i].weight);
    }

    clock_t start, end;
    double cpu_time_used;

    start = clock(); // Start time measurement
    KruskalMST(graph);
    end = clock(); // End time measurement

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC; // Calculate total time taken
    printf("\nTime taken to construct the Minimum Spanning Tree: %f seconds\n", cpu_time_used);

    free(graph->edge);
    free(graph);
    return 0;
}
```

The program now calculates the Minimum Spanning Tree (MST) using Kruskal's algorithm with the Greedy method. It also measures the execution time for constructing the MST.

Key Components of the Update

1. **Graph Representation**:
    - Uses a structure to represent edges and a graph.
2. **Kruskal's Algorithm**:
    - Uses a **Union-Find** method with path compression for efficient cycle detection.
3. **Time Measurement**:
    - The clock() function measures the total time taken to construct the MST.
4. **User Input**:
    - Number of vertices and edges.
    - Source, destination, and weight of each edge.
5. **Output**:
    - The edges included in the MST and the time taken to compute the MST.

5.Develop a program and measure the running time for estimating Single Source Shortest Paths with Greedy Method

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

```c
#define MAX 100

// Function to find the vertex with minimum distance value, from the set of vertices not yet
included in the shortest path tree
int minDistance(int dist[], int sptSet[], int V) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == 0 && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

// Function to print the constructed distance array
void printSolution(int dist[], int V) {
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's shortest path algorithm for a graph represented using
an adjacency matrix
void dijkstra(int graph[MAX][MAX], int V, int src) {
    int dist[MAX]; // The output array. dist[i] will hold the shortest distance from src to i
    int sptSet[MAX]; // sptSet[i] will be 1 if vertex i is included in shortest path tree or shortest
distance from src to i is finalized

    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = 0;
    }

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet, V);

        sptSet[u] = 1;

        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] <
dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    printSolution(dist, V);
}
```

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

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

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

    clock_t start, end;
    double cpu_time_used;

    start = clock(); // Start time measurement
    dijkstra(graph, V, src);
    end = clock(); // End time measurement

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC; // Calculate total time taken
    printf("\nTime taken to compute Single Source Shortest Paths: %f seconds\n", cpu_time_used);

    return 0;
}
```

6.Develop a program and measure the running time for optimal Binary search trees with Dynamic Programming

```
#include <stdio.h>
#include <limits.h>
#include <time.h>

#define MAX 100

// Function to calculate the cost of the Optimal Binary Search Tree
int optimalBST(int keys[], int freq[], int n) {
    int cost[MAX][MAX];

    for (int i = 0; i < n; i++)
        cost[i][i] = freq[i];

```

```
    for (int L = 2; L <= n; L++) {
        for (int i = 0; i <= n - L + 1; i++) {
            int j = i + L - 1;
            cost[i][j] = INT_MAX;

            int sum = 0;
            for (int k = i; k <= j; k++)
                sum += freq[k];

            for (int r = i; r <= j; r++) {
                int c = ((r > i) ? cost[i][r - 1] : 0) + ((r < j) ? cost[r + 1][j] : 0) + sum;
                if (c < cost[i][j])
                    cost[i][j] = c;
            }
        }
    }
    return cost[0][n - 1];
}

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

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

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

    clock_t start, end;
    double cpu_time_used;

    start = clock(); // Start time measurement
    int min_cost = optimalBST(keys, freq, n);
    end = clock(); // End time measurement

    printf("The cost of the Optimal Binary Search Tree is: %d\n", min_cost);

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC; // Calculate total time taken
    printf("\nTime taken to compute Optimal Binary Search Tree: %f seconds\n",
cpu_time_used);

    return 0;
}
```

7. Develop a program and measure the running time for identifying solution for traveling salesperson problem with Dynamic Programming

```
#include <stdio.h>
#include <limits.h>
#include <time.h>

#define MAX 12 // Maximum number of cities (keep small to avoid large computation time)

int tsp(int graph[MAX][MAX], int n, int pos, int visited, int dp[MAX][1 << MAX]) {
    if (visited == ((1 << n) - 1))
        return graph[pos][0]; // Return to the starting city

    if (dp[pos][visited] != -1)
        return dp[pos][visited];

    int min_cost = INT_MAX;

    for (int city = 0; city < n; city++) {
        if ((visited & (1 << city)) == 0) { // If the city is not visited
            int cost = graph[pos][city] + tsp(graph, n, city, visited | (1 << city), dp);
            if (cost < min_cost)
                min_cost = cost;
        }
    }

    dp[pos][visited] = min_cost;
    return min_cost;
}

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

    if (n > MAX) {
        printf("Number of cities is too large. Please enter a value less than or equal to %d.\n",
MAX);
        return 1;
    }

    int graph[MAX][MAX];
    printf("Enter the cost matrix (use 0 for no path between cities):\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
```

```
    int dp[MAX][1 << MAX];
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < (1 << MAX); j++) {
            dp[i][j] = -1;
        }
    }

    clock_t start, end;
    double cpu_time_used;

    start = clock(); // Start time measurement
    int min_cost = tsp(graph, n, 0, 1, dp);
    end = clock(); // End time measurement

    printf("The minimum cost to visit all cities is: %d\n", min_cost);

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC; // Calculate total time taken
    printf("\nTime taken to compute Traveling Salesperson Problem: %f seconds\n",
cpu_time_used);

    return 0;
}
```

8. Develop a program and measure the running time for identifying solution for 8-Queens problem with Backtracking

```
#include <stdio.h>
#include <stdbool.h>
#include <time.h>

#define N 8 // Size of the chessboard (8x8 for 8-queens problem)

void printSolution(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%2d ", board[i][j]);
        }
        printf("\n");
    }
}

// Function to check if a queen can be placed on board[row][col]
bool isSafe(int board[N][N], int row, int col) {
    for (int i = 0; i < col; i++)
        if (board[row][i])
            return false;
```

```
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    for (int i = row, j = col; i < N && j >= 0; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

bool solveNQUtil(int board[N][N], int col) {
    if (col >= N)
        return true;

    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;

            if (solveNQUtil(board, col + 1))
                return true;

            board[i][col] = 0; // Backtrack
        }
    }

    return false;
}

bool solveNQ() {
    int board[N][N] = {0};

    clock_t start, end;
    double cpu_time_used;

    start = clock(); // Start time measurement
    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist\n");
        return false;
    }
    end = clock(); // End time measurement

    printSolution(board);

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC; // Calculate total time
taken
    printf("\nTime taken to solve 8-Queens problem: %f seconds\n", cpu_time_used);

    return true;
}
```

```
int main() {
    solveNQ();
    return 0;
}
```

9. Develop a program and measure the running time for Graph Coloring with Backtracking

```
#include <stdio.h>
#include <stdbool.h>
#include <time.h>

#define V 4 // Number of vertices

// Function to print the solution
void printSolution(int color[]) {
    printf("Solution (Vertex : Color):\n");
    for (int i = 0; i < V; i++)
        printf("%d : %d\n", i + 1, color[i]);
}

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

// Recursive function to solve the graph coloring problem
bool graphColoringUtil(bool 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;
}

bool graphColoring(bool graph[V][V], int m) {
```

```
    int color[V] = {0};

    clock_t start, end;
    double cpu_time_used;

    start = clock(); // Start time measurement
    if (graphColoringUtil(graph, m, color, 0) == false) {
        printf("Solution does not exist\n");
        return false;
    }
    end = clock(); // End time measurement

    printSolution(color);

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC; // Calculate total time taken
    printf("\nTime taken to solve Graph Coloring problem: %f seconds\n", cpu_time_used);

    return true;
}

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

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

    graphColoring(graph, m);

    return 0;
}
```

10 . Develop a program and measure the running time to generate solution of Hamiltonian Cycle problem with Backtracking

```
#include <stdio.h>
#include <stdbool.h>
#include <time.h>

#define V 5 // Number of vertices

// Function to print the solution
void printSolution(int path[]) {
    printf("Solution (Hamiltonian Cycle): ");
    for (int i = 0; i < V; i++)
```

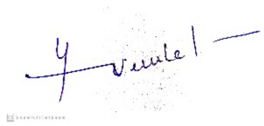
```
        printf("%d -> ", path[i]);
    printf("%d\n", path[0]);
}

// Check if the vertex v can be added at index pos in the Hamiltonian Cycle
bool isSafe(int v, bool graph[V][V], int path[], int pos) {
    if (!graph[path[pos - 1]][v])
        return false;

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

    return true;
}

// Recursive function to solve the Hamiltonian Cycle problem
bool hamCycleUtil(bool graph[V][V], int path[], int pos) {
    if (pos == V) {
        if (graph[path[pos - 1]][path[0]])
            return true;
        else
            return false;
    }

    for (int v = 1; v < V; v++) {
        if (isSafe(v, graph, path, pos)) {
            path[pos] = v;

            if (hamCycleUtil(graph, path, pos + 1))
                return true;

            path[pos] = -1; // Backtrack
        }
    }
    return false;
}

bool hamCycle(bool graph[V][V]) {
    int path[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    path[0] = 0;

    clock_t start, end;
    double cpu_time_used;

    start = clock(); // Start time measurement
    if (!hamCycleUtil(graph, path, 1)) {
```

```
        printf("Solution does not exist\n");
        return false;
    }
    end = clock(); // End time measurement

    printSolution(path);

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC; // Calculate total time taken
    printf("\nTime taken to solve Hamiltonian Cycle problem: %f seconds\n", cpu_time_used);

    return true;
}

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

    hamCycle(graph);

    return 0;
}
```

11. Develop a program and measure the running time running time to generate solution of Knapsack problem with Backtracking

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

#define MAX 100 // Maximum number of items

int max(int a, int b) {
    return (a > b) ? a : b;
}

// Backtracking function to solve the knapsack problem
int knapsack(int W, int wt[], int val[], int n) {
    if (n == 0 || W == 0)
        return 0;

    if (wt[n - 1] > W)
        return knapsack(W, wt, val, n - 1);

    return max(
        val[n - 1] + knapsack(W - wt[n - 1], wt, val, n - 1),
```

```
        knapsack(W, wt, val, n - 1)
    );
}

int main() {
    int n, W;
    int val[MAX], wt[MAX];

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

    printf("Enter the maximum capacity of the knapsack: ");
    scanf("%d", &W);

    printf("Enter the weights of the items: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &wt[i]);

    printf("Enter the values of the items: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &val[i]);

    clock_t start, end;
    double cpu_time_used;

    start = clock(); // Start time measurement
    int maxProfit = knapsack(W, wt, val, n);
    end = clock(); // End time measurement

    printf("Maximum profit is: %d\n", maxProfit);

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC; // Calculate total time taken
    printf("Time taken to solve Knapsack problem: %f seconds\n", cpu_time_used);

    return 0;
}
```



Dr Y Venkat
Professor,Dept of CSE