Algorithm Lab by Tahmid Ahmed

Experiment No: 01

Experiment Name: Write a program to search an element from a given array using Binary Search algorithm.

Source Code:

```
#include <stdio.h>

int binarySearch(int array[], int x, int low, int high) {
  if (high >= low) {
    int mid = low + (high - low) / 2;

    // If found at mid, then return it
    if (array[mid] == x)
      return mid;

    // Search the left half
    if (array[mid] > x)
      return binarySearch(array, x, low, mid - 1);

    // Search the right half
    return binarySearch(array, x, mid + 1, high);
  }

  return -1;
}

int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 4;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
}
```

Experiment No: 02

Experiment Name: Write a program to find the maximum and minimum numbers from a given array using Divide and Conquer method.

Source Code:

```
#include<stdio.h>
#include<stdio.h>
int max, min;
```

```
int a[100];
void maxmin(int i, int j)
{
 int max1, min1, mid;
 if(i==j)
 {
  max = min = a[i];
 }
 else
 {
  if(i == j-1)
  {
   if(a[i] <a[j])
   {
    max = a[j];
    min = a[i];
   }
   else
   {
    max = a[i];
    min = a[j];
   }
  }
  else
  {
   mid = (i+j)/2;
   maxmin(i, mid);
   max1 = max; min1 = min;
   maxmin(mid+1, j);
   if(max <max1)
    max = max1;
   if(min > min1)
    min = min1;
  }
 }
}
int main ()
{
 int i, num;
 printf ("\nEnter the total number of numbers : ");
 scanf ("%d",&num);
 printf ("Enter the numbers : \n");
 for (i=1;i<=num;i++)
  scanf ("%d",&a[i]);
```

```
 max = a[0];
 min = a[0];
 maxmin(1, num);
 printf ("Minimum element in an array : %d\n", min);
 printf ("Maximum element in an array : %d\n", max);
 return 0;
}
```

Experiment No: 03

Experiment Name: Write a program to measure & compare the performances of Bubble sort & Quick sort algorithms using time function.

Source Code:

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

// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
            i++; // increment index of smaller element
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
```

```
            arr[j] = temp;
        }
    }
    // Swap arr[i+1] and arr[high] (or pivot)
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}

// Function to perform Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // pi is partitioning index, arr[pi] is now at right place
        int pi = partition(arr, low, high);

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

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

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

    // Measure time taken by Bubble Sort
    clock_t start_bubble = clock();
    bubbleSort(arr, n);
    clock_t end_bubble = clock();
    double time_bubble = ((double)(end_bubble - start_bubble)) / CLOCKS_PER_SEC;

    // Measure time taken by Quick Sort
    clock_t start_quick = clock();
    quickSort(arr, 0, n - 1);
    clock_t end_quick = clock();
    double time_quick = ((double)(end_quick - start_quick)) / CLOCKS_PER_SEC;
```

```
    // Display the time taken by both sorting algorithms
    printf("Time taken by Bubble Sort: %lf seconds\n", time_bubble);
    printf("Time taken by Quick Sort: %lf seconds\n", time_quick);

    return 0;
}
```

Experiment No: 04

Experiment Name: **Write a program to solve the Fractional Knapsack problem using Greedy method.**

Source Code:

```
#include<stdio.h>
int main()
{
    float weight[50],profit[50],ratio[50],Totalvalue,temp,capacity,amount;
    int n,i,j;
    printf("Enter the number of items :");
    scanf("%d",&n);
   for (i = 0; i < n; i++)
   {
      printf("Enter Weight and Profit for item[%d] :\n",i);
      scanf("%f %f", &weight[i], &profit[i]);
   }
   printf("Enter the capacity of knapsack :\n");
   scanf("%f",&capacity);

    for(i=0;i<n;i++)
        ratio[i]=profit[i]/weight[i];

   for (i = 0; i < n; i++)
     for (j = i + 1; j < n; j++)
        if (ratio[i] < ratio[j])
        {
           temp = ratio[j];
           ratio[j] = ratio[i];
           ratio[i] = temp;

           temp = weight[j];
           weight[j] = weight[i];
           weight[i] = temp;
```

```
            temp = profit[j];
            profit[j] = profit[i];
            profit[i] = temp;
          }

    printf("Knapsack problems using Greedy Algorithm:\n");
    for (i = 0; i < n; i++)
    {
     if (weight[i] > capacity)
        break;
      else
     {
        Totalvalue = Totalvalue + profit[i];
        capacity = capacity - weight[i];
      }
    }
      if (i < n)
      Totalvalue = Totalvalue + (ratio[i]*capacity);
    printf("\nThe maximum value is :%f\n",Totalvalue);
    return 0;
}
```

Experiment No: 05

Experiment Name: Write a program to find the minimum cost spanning tree using Prim's algorithm.

Source Code:

```
// A C program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph

#include <limits.h>
#include <stdbool.h>
#include <stdio.h>

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
```

```
	// Initialize min value
	int min = INT_MAX, min_index;

	for (int v = 0; v < V; v++)
		if (mstSet[v] == false && key[v] < min)
			min = key[v], min_index = v;

	return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
int printMST(int parent[], int graph[V][V])
{
	printf("Edge \tWeight\n");
	for (int i = 1; i < V; i++)
		printf("%d - %d \t%d \n", parent[i], i,
			graph[i][parent[i]]);
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
	// Array to store constructed MST
	int parent[V];
	// Key values used to pick minimum weight edge in cut
	int key[V];
	// To represent set of vertices included in MST
	bool mstSet[V];

	// Initialize all keys as INFINITE
	for (int i = 0; i < V; i++)
		key[i] = INT_MAX, mstSet[i] = false;

	// Always include first 1st vertex in MST.
	// Make key 0 so that this vertex is picked as first
	// vertex.
	key[0] = 0;

	// First node is always root of MST
	parent[0] = -1;
```

```
        // The MST will have V vertices
        for (int count = 0; count < V - 1; count++) {

                // Pick the minimum key vertex from the
                // set of vertices not yet included in MST
                int u = minKey(key, mstSet);

                // Add the picked vertex to the MST Set
                mstSet[u] = true;

                // Update key value and parent index of
                // the adjacent vertices of the picked vertex.
                // Consider only those vertices which are not
                // yet included in MST
                for (int v = 0; v < V; v++)

                        // graph[u][v] is non zero only for adjacent
                        // vertices of m mstSet[v] is false for vertices
                        // not yet included in MST Update the key only
                        // if graph[u][v] is smaller than key[v]
                        if (graph[u][v] && mstSet[v] == false
                                && graph[u][v] < key[v])
                                parent[v] = u, key[v] = graph[u][v];
        }

        // print the constructed MST
        printMST(parent, graph);
}

// Driver's code
int main()
{
        int graph[V][V] = { { 0, 2, 0, 6, 0 },
                                                { 2, 0, 3, 8, 5 },
                                                { 0, 3, 0, 0, 7 },
                                                { 6, 8, 0, 0, 9 },
                                                { 0, 5, 7, 9, 0 } };

        // Print the solution
        primMST(graph);

        return 0;
}
```

Experiment No: 06
Experiment Name: Write a program to solve the single source shortest path problem using Dijkstra's algorithm.
Source Code:

```
// C program for Dijkstra's single source shortest path
// algorithm. The program is for adjacency matrix
// representation of the graph

#include <limits.h>
#include <stdbool.h>
#include <stdio.h>

// Number of vertices in the graph
#define V 9

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

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

   return min_index;
}

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

// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// adjacency matrix representation
```

```
void dijkstra(int graph[V][V], int src)
{
   int dist[V]; // The output array.  dist[i] will hold the
                // shortest
   // distance from src to i

   bool sptSet[V]; // sptSet[i] will be true if vertex i is
                   // included in shortest
   // path tree or shortest distance from src to i is
   // finalized

   // Initialize all distances as INFINITE and stpSet[] as
   // false
   for (int i = 0; i < V; i++)
      dist[i] = INT_MAX, sptSet[i] = false;

   // Distance of source vertex from itself is always 0
   dist[src] = 0;

   // Find shortest path for all vertices
   for (int count = 0; count < V - 1; count++) {
      // Pick the minimum distance vertex from the set of
      // vertices not yet processed. u is always equal to
      // src in the first iteration.
      int u = minDistance(dist, sptSet);

      // Mark the picked vertex as processed
      sptSet[u] = true;

      // Update dist value of the adjacent vertices of the
      // picked vertex.
      for (int v = 0; v < V; v++)

         // Update dist[v] only if is not in sptSet,
         // there is an edge from u to v, and total
         // weight of path from src to  v through u is
         // smaller than current value of dist[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];
   }

   // print the constructed distance array
```

```
    printSolution(dist);
}

// driver's code
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    // Function call
    dijkstra(graph, 0);

    return 0;
}
```

Experiment No: 07

Experiment Name: Write a program to solve the All Pairs Shortest Path problem using Floyd's algorithm.

Source Code:

```
// C Program for Floyd Warshall Algorithm
#include <stdio.h>

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough
  value. This value will be used
  for vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);
```

```
// Solves the all-pairs shortest path
// problem using Floyd Warshall algorithm
void floydWarshall(int dist[][V])
{
   int i, j, k;

   /* Add all vertices one by one to
     the set of intermediate vertices.
     ---> Before start of an iteration, we
     have shortest distances between all
     pairs of vertices such that the shortest
     distances consider only the
     vertices in set {0, 1, 2, .. k-1} as
     intermediate vertices.
     ----> After the end of an iteration,
     vertex no. k is added to the set of
     intermediate vertices and the set
     becomes {0, 1, 2, .. k} */
   for (k = 0; k < V; k++) {
      // Pick all vertices as source one by one
      for (i = 0; i < V; i++) {
         // Pick all vertices as destination for the
         // above picked source
         for (j = 0; j < V; j++) {
            // If vertex k is on the shortest path from
            // i to j, then update the value of
            // dist[i][j]
            if (dist[i][k] + dist[k][j] < dist[i][j])
               dist[i][j] = dist[i][k] + dist[k][j];
         }
      }
   }

   // Print the shortest distance matrix
   printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
   printf(
      "The following matrix shows the shortest distances"
      " between every pair of vertices \n");
   for (int i = 0; i < V; i++) {
```

```
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

// driver's code
int main()
{
    /* Let us create the following weighted graph
            10
       (0)------->(3)
        |        /|\
      5 |         |
        |         | 1
       \|/        |
       (1)------->(2)
            3           */
    int graph[V][V] = { { 0, 5, INF, 10 },
                        { INF, 0, 3, INF },
                        { INF, INF, 0, 1 },
                        { INF, INF, INF, 0 } };

    // Function call
    floydWarshall(graph);
    return 0;
}
```

Experiment No: 08

Experiment Name: Write a program to solve the 4-queens problem using Backtracking Method.

Source Code:

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

#define N 4 // Number of Queens

// Function to print the solution
```

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

    // Check this row on left side
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

// Recursive function to solve N-queens problem
bool solveNQUtil(int board[N][N], int col) {
    // Base case: If all queens are placed, return true
    if (col >= N)
        return true;

    // Consider this column and try placing this queen in all rows one by one
    for (int i = 0; i < N; i++) {
        // Check if the queen can be placed on board[i][col]
        if (isSafe(board, i, col)) {
            // Place this queen in board[i][col]
            board[i][col] = 1;

            // Recur to place rest of the queens
```

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

            // If placing queen in board[i][col] doesn't lead to a solution,
            // then remove the queen from board[i][col]
            board[i][col] = 0; // BACKTRACK
        }
    }

    // If the queen cannot be placed in any row in this column col, then return false
    return false;
}

// Function to solve N-queens problem
void solveNQ() {
    int board[N][N] = {0};

    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return;
    }

    printSolution(board);
}

// Main function
int main() {
    solveNQ();
    return 0;
}
```

Experiment No: 09

Experiment Name: Write a program to solve the Sum of Subsets problem using Backtracking Method.

Source Code:

```
#include <stdio.h>

#define MAX 10

int set[MAX];
int include[MAX];

void subsetSum(int, int, int, int);
void display(int, int);
```

```
int main() {
    int n, sum;

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

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

    printf("Enter the sum to find subsets: ");
    scanf("%d", &sum);

    printf("The subsets with sum %d are:\n", sum);
    subsetSum(0, 0, sum, n);

    return 0;
}

void subsetSum(int i, int sum, int targetSum, int n) {
    if (sum == targetSum) {
        display(i, n);
        return;
    }
    if (i == n)
        return;

    include[i] = 1;
    subsetSum(i + 1, sum + set[i], targetSum, n);
    include[i] = 0;
    subsetSum(i + 1, sum, targetSum, n);
}

void display(int i, int n) {
    printf("{ ");
    for (int j = 0; j < i; j++) {
        if (include[j] == 1)
            printf("%d ", set[j]);
    }
    printf("}\n");
}
```

Experiment No: 10

Experiment Name: Write a program to solve the Sum of Subsets problem using Backtracking Method.

Source Code:

```
// C program for solution of M
// Coloring problem using backtracking

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

// Number of vertices in the graph
#define V 4

void printSolution(int color[]);

/* A utility function to check if
the current color assignment
is safe for vertex v i.e. checks
whether the edge exists or not
(i.e, graph[v][i]==1). If exist
then checks whether the color to
be filled in the new vertex(c is
sent in the parameter) is already
used by its adjacent
vertices(i-->adj vertices) or
not (i.e, color[i]==c) */
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;
}

/* A recursive utility function
to solve m coloring problem */
bool graphColoringUtil(bool graph[V][V], int m, int color[],
                                                int v)
{
	/* base case: If all vertices are
	assigned a color then return true */
	if (v == V)
		return true;
```

```
        /* Consider this vertex v and
        try different colors */
        for (int c = 1; c <= m; c++) {
                /* Check if assignment of color
                c to v is fine*/
                if (isSafe(v, graph, color, c)) {
                        color[v] = c;

                        /* recur to assign colors to
                        rest of the vertices */
                        if (graphColoringUtil(graph, m, color, v + 1)
                                == true)
                                return true;

                        /* If assigning color c doesn't
                        lead to a solution then remove it */
                        color[v] = 0;
                }
        }

        /* If no color can be assigned to
        this vertex then return false */
        return false;
}

/* This function solves the m Coloring
problem using Backtracking. It mainly
uses graphColoringUtil() to solve the
problem. It returns false if the m
colors cannot be assigned, otherwise
return true and prints assignments of
colors to all vertices. Please note
that there may be more than one solutions,
this function prints one of the
feasible solutions.*/
bool graphColoring(bool graph[V][V], int m)
{
        // Initialize all color values as 0.
        // This initialization is needed
        // correct functioning of isSafe()
        int color[V];
        for (int i = 0; i < V; i++)
                color[i] = 0;
```

```
	// Call graphColoringUtil() for vertex 0
	if (graphColoringUtil(graph, m, color, 0) == false) {
		printf("Solution does not exist");
		return false;
	}

	// Print the solution
	printSolution(color);
	return true;
}

/* A utility function to print solution */
void printSolution(int color[])
{
	printf("Solution Exists:"
		" Following are the assigned colors \n");
	for (int i = 0; i < V; i++)
		printf(" %d ", color[i]);
	printf("\n");
}

// Driver code
int main()
{
	/* Create following graph and test
	whether it is 3 colorable
	(3)---(2)
	| / |
	| / |
	| / |
	(0)---(1)
	*/
	bool graph[V][V] = {
		{ 0, 1, 1, 1 },
		{ 1, 0, 1, 0 },
		{ 1, 1, 0, 1 },
		{ 1, 0, 1, 0 },
	};
	int m = 3; // Number of colors

	// Function call
	graphColoring(graph, m);
	return 0;
}
```

Algorithm Lab by Tahmid Ahmed