Assignment - 1

1. Write a program for searching a given element. The program will display the presence and absence of the given element and also show the positional value if it is present.
Implement the above program by applying linear search and using dynamic memory allocation
technique.

2. Solve the above program with binary search.

Program:

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

// Binary and linear search on a dynamic array

int binary_search(int *ptr, int left, int right, int item) {
    /*
        binary_search function takes the pointer PTR to the first
        item in the array, size of the array, and the ITEM to be searched.
        Returns the position of the element of -1 if not found.
    */

      int i, mid;

      if (right >= left) {
        int mid = left + (right - left) / 2;

        if (ptr[mid] == item)
            return mid;

        if (ptr[mid] > item)
            return binary_search(ptr, left, mid - 1, item);

        return binary_search(ptr, mid + 1, right, item);
    }

    return -1;
}

int linear_search(int *ptr, int max, int item) {
    /*
        linear_search function takes the pointer PTR to the first
        item in the array, size of the array, and the ITEM to be searched.
        Returns the position of the element of -1 if not found.
```

```
    */

    int position_of_item, i;

      for (i = 0; i < max; ++i) {

            if (item == ptr[i]) {
                  position_of_item = i;
                  break;
            }

            else
                  position_of_item = -1;
      }

      return position_of_item;
}

void main() {
    int item, i, max, *ptr, return_value, choice;

    // max stores the numbers of elements to be stored in the array.

    printf("Enter number of elements you wish to store: ");
    scanf("%d", &max);

    ptr = (int*)malloc(max * sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation failed.\n");
        exit(0);
    }
    else {
        printf("Memory allocation sucessful.\n");
        printf("Enter the elements to be stored in an ascending form.\n");

        for (i = 0; i < max; ++i) {
            printf("Enter element number[%d]: ", i);
                  scanf("%d", &ptr[i]);
        }

            for (i = 0; i < max; ++i)
                  printf("| %d ", ptr[i]);
        printf(" |\n");
    }

    printf("Enter element to be searched: ");
    scanf("%d", &item);

      choice:
            printf("\n\n1.Use linear Search\n2.Use Binary Search\nEnter choice: ");
            scanf("%d", &choice);
```

```
	switch (choice)
	{
	case 1:
		// calling the linear search function
		return_value = linear_search(ptr, max, item);

		if (return_value >= 0)
			printf("Element found at: %d\n", return_value);
		else
			printf("Element not found\n");
		break;

	case 2:
		// calling the binary search function
		return_value = binary_search(ptr, 0, max-1, item);

		if (return_value >= 0)
			printf("Element found at: %d\n", return_value);
		else
			printf("Element not found\n");
		break;

	default:
		goto choice;
		break;
	}
}
```

Output:

```
Enter number of elements you wish to store: 8
Memory allocation successful.
Enter the elements to be stored in an ascending form.
Enter element number[0]: 1
Enter element number[1]: 2
Enter element number[2]: 3
Enter element number[3]: 4
Enter element number[4]: 5
Enter element number[5]: 6
Enter element number[6]: 7
Enter element number[7]: 8
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8  |
Enter element to be searched: 6


1.Use linear Search
2.Use Binary Search
Enter choice: 2
Element found at: 5
```

Assignment - 2

1. Write a program for sorting a giving array of elements, using the technique of merge sort. The array must be dynamically allocated.

Program:

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

void merge(int *ptr, int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = ptr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = ptr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            ptr[k] = L[i];
            i++;
        }
        else {
            ptr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
    are any */
    while (i < n1) {
        ptr[k] = L[i];
        i++;
        k++;
```

```
    }

    /* Copy the remaining elements of R[], if there
    are any */
    while (j < n2) {
        ptr[k] = R[j];
        j++;
        k++;
    }
}

void merge_sort(int *ptr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

            merge_sort(ptr, l, m);
            merge_sort(ptr, m + 1, r);

            merge(ptr, l, m, r);
    }
}

int main() {
      int i,  *ptr, max, start_t, end_t, l, r, m;
      double total;

    // max stores the numbers of elements to be stored in the array.

    printf("Enter number of elements: ");
    scanf("%d", &max);

    ptr = (int*)malloc(max * sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation harshil.\n");
        exit(0);
    }
    else {
        printf("Memory allocation sucessful.\n");
        printf("Enter the elements:\n");

        for (i = 0; i < max; ++i) {
            printf("Enter element number[%d]: ", i);
                  scanf("%d", &ptr[i]);
        }

            for (i = 0; i < max; ++i)
                  printf("| %d ", ptr[i]);
        printf(" |\n");
    }

    l = 0; r = max - 1; m = l+r/2;
```

```
    // call the merge function
      start_t = clock();
      {
            merge_sort(ptr, 0, max - 1);
      }
      end_t = clock();
      total = (double)(end_t - start_t);

      //    printf("Time taken by your desired function is: %f\n", total);

      printf("Sorted array:\n");
      for (i = 0; i < max; ++i)
                  printf("| %d ", ptr[i]);
    printf(" |\n");

      printf("Time taken by your desired function is: %f\n", total);
      return 0;
}
```

Output:

```
Enter number of elements: 10
Memory allocation successful.
Enter the elements:
Enter element number[0]: 45
Enter element number[1]: 554
Enter element number[2]: 238
Enter element number[3]: 482
Enter element number[4]: 4873
Enter element number[5]: 28
Enter element number[6]: 477
Enter element number[7]: 0
Enter element number[8]: 037
Enter element number[9]: 47
| 45 | 554 | 238 | 482 | 4873 | 28 | 477 | 0 | 37 | 47  |
Sorted array:
| 0 | 28 | 37 | 45 | 47 | 238 | 477 | 482 | 554 | 4873  |
Time taken by your desired function is: 4.000000
```

2. Write a program for sorting a giving array of elements, using the technique of quick sort. The array must be dynamically allocated.

Program:

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

void print(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("| %d ", arr[i]);

      printf(" |\n");
}

int partition(int arr[], int low, int high) {
    int pivot, i, j, temp;
    pivot = arr[low];
    i = low + 1;
    j = high;
    do
    {

        while (arr[i] <= pivot)
        {
            i++;
        }
        while (arr[j] > pivot)
        {
            j--;
        }
        if (i < j)
        {
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    } while (i < j);
    temp = arr[low];
    arr[low] = arr[j];
    arr[j] = temp;

    return j;
}

void quicksort(int arr[], int low, int high) {
    int partition_index;
    if (low < high)
    {
```

```
        partition_index = partition(arr, low, high);
        quicksort(arr, low,  partition_index - 1);
        quicksort(arr,  partition_index + 1, high);
    }
}

int main() {
    int *arr, i = 0, n, start_t, end_t;
    double total;

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

    arr=(int*)malloc(n*sizeof(int));

    if(arr==NULL) {
        printf("Memory allocation failed.\n");
        exit(0);
    }

    else
        for(i=0;i<n;i++) {
                  printf("Enter element number[%d]: ", i);
                  scanf("%d",&arr[i]);
           }

    print(arr, n);

    start_t=clock();
    {
          quicksort(arr, 0, n - 1);
    }
    end_t=clock();
      total=(double)(end_t - start_t);

      printf("Sorted array:\n");
    print(arr, n);

      printf("Time taken by the function %f\n",total);

      return 0;
}
```

Output:

```
Enter number of elements: 10
Enter element number[0]: 45
Enter element number[1]: 554
Enter element number[2]: 238
Enter element number[3]: 482
```

```
Enter element number[4]: 4873
Enter element number[5]: 28
Enter element number[6]: 477
Enter element number[7]: 0
Enter element number[8]: 037
Enter element number[9]: 47
| 45 | 554 | 238 | 482 | 4873 | 28 | 477 | 0 | 37 | 47  |
Sorted array:
| 0 | 28 | 37 | 45 | 47 | 238 | 477 | 482 | 554 | 4873  |
Time taken by the function 4.000000
```

Assignment - 3

1. Write a program for executing the knapsack algorithm

weight = {2.0, 3.0, 5.0, 7.0, 1.0, 4.0, 1.0}
profit = {10.0, 5.0, 5.0, 7.0, 6.0, 18.0, 3.0}
Capacity = 15

Algorithm:

```
for i from 1 to n do:

    A[i, 0] := 0

for p from 1 to n*P do:

    if p == p[1] then:

        A[1, p] = p[1]

    else:

        A[1, p] = infty

for i from 2 to n do:

    for p from 1 to n*P do:

        if p[i] <= p then:

            A[i, p] = min( A[i-1, p], s[i] + A[i-1, p-p[i]] )

        else:

            A[i, p] = A[i-1, p]

Z = 0

for p from 1 to n*P do:

    if A[n, p] <= B then:

        Z = p

return Z
```

Program:

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

void merge(float ratio[], float weight[], float profit[], int mid, int low, int
high) {
    int i, j, k;
    float B[30],C[30], D[30];
    i = low;
    j = mid + 1;
    k = low;

    while (i <= mid && j <= high) {
        if (ratio[i] < ratio[j]) {
            B[k] = ratio[i];
            C[k] = weight[i];
            D[k] = profit[i];
            i++;
            k++;
        }
        else {
            B[k] = ratio[j];
            C[k] = weight[j];
            D[k] = profit[j];
            j++;
            k++;
        }
    }
    while (i <= mid) {
        B[k] = ratio[i];
        C[k] = weight[i];
        D[k] = profit[i];
        k++;
        i++;
    }
    while (j <= high) {
        B[k] = ratio[j];
        C[k] = weight[j];
        D[k] = profit[j];
        k++;
        j++;
    }
    for (int i = low; i <= high; i++) {
        ratio[i] = B[i];
        weight[i] = C[i];
        profit[i] = D[i];
    }
}

void mergesort(float ratio[], float weight[], float profit[], int low, int high) {
```

```
    int mid;

    if (low < high) {
        mid = (low + high) / 2;
        mergesort(ratio, weight, profit, low, mid);
        mergesort(ratio, weight, profit, mid + 1, high);
        merge(ratio, weight, profit, mid, low, high);
    }
}

void main() {
    int noofobject = 7, index, rem_cap = 15, i;
    float cost = 0.0;
    float ratio[7];
    float weight[7] = {2.0, 3.0, 5.0, 7.0, 1.0, 4.0, 1.0};
    float profit[7] = {10.0, 5.0, 5.0, 7.0, 6.0, 18.0, 3.0};

    printf("Ratio before sorting in ascending order:\n");
    for (index = 0; index < noofobject; index++) {
        ratio[index] = (float)(profit[index] / weight[index]);
        printf("%f ", ratio[index]);
    }
      printf("\n");

      mergesort(ratio, weight, profit, 0, noofobject - 1);
    printf("Ratios after sorting in ascending order:\n");
    for (i = 0; i < 7; i++) {
        printf("%f ", ratio[i]);
    }
    printf("\n");

      printf("Weights sorted with respect to ratios:\n");
    for (i = 0; i < 7; i++) {
        printf("%f ", weight[i]);
    }
    printf("\n");

      printf("Profits sorted with respect to ratios:\n");
    for (i = 0; i < 7; i++) {
        printf("%f ", profit[i]);
    }
    printf("\n");

      for (index = noofobject - 1; index >= 0; index--) {
        if (weight[index] <= rem_cap) {
            cost = cost + profit[index];
            rem_cap = rem_cap - weight[index];
        }
        else if (rem_cap > 0) {
            cost = cost + ((profit[index] / weight[index]) * rem_cap);
            rem_cap = 0;
            break;
        }
```

```
    }
    printf("Total profit: %f\n", cost);
}
```

Output:

```
Ratio before sorting in ascending order:
5.000000 1.666667 1.000000 1.000000 6.000000 4.500000 3.000000
Ratios after sorting in ascending order:
1.000000 1.000000 1.666667 3.000000 4.500000 5.000000 6.000000
Weights sorted with respect to ratios:
7.000000 5.000000 3.000000 1.000000 4.000000 2.000000 1.000000
Profits sorted with respect to ratios:
7.000000 5.000000 5.000000 3.000000 18.000000 10.000000 6.000000
Total profit: 46.000000
```

Assignment - 4

1. Minimal Spanning Tree by Prims Algorithm.

Algorithm.

1. Determine an arbitrary vertex as the starting vertex of the MST.
2. Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
3. Find edges connecting any tree vertex with the fringe vertices.
4. Find the minimum among these edges.
5. Add the chosen edge to the MST if it does not form any cycle.
6. Return the MST and exit

Program:

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

#define V 5

int minKey(int key[], bool mstSet[])
{
	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;
}

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]]);
}

void primMST(int graph[V][V])
{
	int parent[V];
	int key[V];
```

```
	bool mstSet[V];

	for (int i = 0; i < V; i++)
		key[i] = INT_MAX, mstSet[i] = false;

	key[0] = 0;

	parent[0] = -1;

	for (int count = 0; count < V - 1; count++) {

		int u = minKey(key, mstSet);

		mstSet[u] = true;

		for (int v = 0; v < V; v++)

			if (graph[u][v] && mstSet[v] == false
				&& graph[u][v] < key[v])
				parent[v] = u, key[v] = graph[u][v];
	}

	printMST(parent, graph);
}

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 } };

	primMST(graph);

	return 0;
}
```

Output:

```
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

2. Minimal Spanning Kruskal's Algorithm.

Algorithm:

1. First, add the edge AB with weight 1 to the MST.
2. Add the edge DE with weight 2 to the MST as it is not creating the cycle.
3. Add the edge BC with weight 3 to the MST, as it is not creating any cycle or loop.
4. Now, pick the edge CD with weight 4 to the MST, as it is not forming the cycle.
5. After that, pick the edge AE with weight 5. Including this edge will create the cycle, so discard it.
6. Pick the edge AC with weight 7. Including this edge will create the cycle, so discard it.
7. Pick the edge AD with weight 10. Including this edge will also create the cycle, so discard it.

Program.

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

int comparator(const void* p1, const void* p2)
{
	const int(*x)[3] = p1;
	const int(*y)[3] = p2;

	return (*x)[2] - (*y)[2];
}

void makeSet(int parent[], int rank[], int n)
{
	for (int i = 0; i < n; i++) {
		parent[i] = i;
		rank[i] = 0;
	}
}

int findParent(int parent[], int component)
{
	if (parent[component] == component)
		return component;

	return parent[component]
		= findParent(parent, parent[component]);
}

void unionSet(int u, int v, int parent[], int rank[], int n)
{
	u = findParent(parent, u);
	v = findParent(parent, v);
```

```
	if (rank[u] < rank[v]) {
		parent[u] = v;
	}
	else if (rank[u] > rank[v]) {
		parent[v] = u;
	}
	else {
		parent[v] = u;

		rank[u]++;
	}
}

void kruskalAlgo(int n, int edge[n][3])
{
	qsort(edge, n, sizeof(edge[0]), comparator);

	int parent[n];
	int rank[n];

	makeSet(parent, rank, n);

	int minCost = 0;

	printf(
		"Following are the edges in the constructed MST\n");
	for (int i = 0; i < n; i++) {
		int v1 = findParent(parent, edge[i][0]);
		int v2 = findParent(parent, edge[i][1]);
		int wt = edge[i][2];

		if (v1 != v2) {
			unionSet(v1, v2, parent, rank, n);
			minCost += wt;
			printf("%d -- %d == %d\n", edge[i][0],
				edge[i][1], wt);
		}
	}

	printf("Minimum Cost Spanning Tree: %d\n", minCost);
}

int main()
{
	int edge[5][3] = { { 0, 1, 10 },
				{ 0, 2, 6 },
				{ 0, 3, 5 },
				{ 1, 3, 15 },
				{ 2, 3, 4 } };

	kruskalAlgo(5, edge);
```

```
      return 0;
}
```

Output:

```
Ratio before sorting in ascending order:
5.000000 1.666667 1.000000 1.000000 6.000000 4.500000 3.000000
Ratios after sorting in ascending order:
1.000000 1.000000 1.666667 3.000000 4.500000 5.000000 6.000000
Weights sorted with respect to ratios:
7.000000 5.000000 3.000000 1.000000 4.000000 2.000000 1.000000
Profits sorted with respect to ratios:
7.000000 5.000000 5.000000 3.000000 18.000000 10.000000 6.000000
Total profit: 46.000000
```

Assignment - 5

1. Dijkstra Algorithm

Algorithm:

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

Program:

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

#define V 9

int minDistance(int dist[], bool sptSet[])
{
      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;
}

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]);
}

void dijkstra(int graph[V][V], int src)
{
      int dist[V];

      bool sptSet[V];
      for (int i = 0; i < V; i++)
            dist[i] = INT_MAX, sptSet[i] = false;

      dist[src] = 0;

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

            sptSet[u] = true;

            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);
}

int main()
{
      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 } };

      dijkstra(graph, 0);

      return 0;
}
```

Output:

```
Vertex           Distance from Source
0                                0
1                                4
2                                12
```

3	19
4	21
5	11
6	9
7	8
8	14

2. Bellman Ford Algorithm

Algorithm:

1. Initialize distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2. Calculate shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph. Do following for each edge u-v
3. If dist[v] > dist[u] + weight of edge uv, then update dist[v] to
dist[v] = dist[u] + weight of edge uv
4. Report if there is a negative weight cycle in the graph. Again traverse every edge and do following for each edge u-v
If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn’t contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

Program.

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

struct Edge
{
    int source, destination, weight;
};

struct Graph
{
    int V, E;

    struct Edge* edge;
};
```

```
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( graph->E * sizeof( struct Edge ) );

    return graph;
}

void FinalSolution(int dist[], int n)
{
    printf("\nVertex\tDistance from Source Vertex\n");
    int i;

    for (i = 0; i < n; ++i){
printf("%d \t\t %d\n", i, dist[i]);
}
}

void BellmanFord(struct Graph* graph, int source)
{
    int V = graph->V;
    int E = graph->E;
    int StoreDistance[V];
    int i,j;


    for (i = 0; i < V; i++)
        StoreDistance[i] = INT_MAX;

    StoreDistance[source] = 0;

    for (i = 1; i <= V-1; i++)
    {
        for (j = 0; j < E; j++)
        {
            int u = graph->edge[j].source;

            int v = graph->edge[j].destination;

            int weight = graph->edge[j].weight;

            if (StoreDistance[u] + weight < StoreDistance[v])
                StoreDistance[v] = StoreDistance[u] + weight;
        }
    }

    for (i = 0; i < E; i++)
    {
```

```
        int u = graph->edge[i].source;

        int v = graph->edge[i].destination;

        int weight = graph->edge[i].weight;

        if (StoreDistance[u] + weight < StoreDistance[v])
            printf("This graph contains negative edge cycle\n");
    }

    FinalSolution(StoreDistance, V);

    return;
}

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

      printf("Enter number of vertices in graph\n");
    scanf("%d",&V);

      printf("Enter number of edges in graph\n");
    scanf("%d",&E);

      printf("Enter your source vertex number\n");
      scanf("%d",&S);

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

    int i;
    for(i=0;i<E;i++){
        printf("\nEnter edge %d properties Source, destination, weight
respectively\n",i+1);
        scanf("%d",&graph->edge[i].source);
        scanf("%d",&graph->edge[i].destination);
        scanf("%d",&graph->edge[i].weight);
    }

    BellmanFord(graph, S);

    return 0;
}
```

Output:

```
Enter number of vertices in graph: 5
Enter number of edges in graph: 3
Enter your source vertex number: 1 1 2

Enter edge 1 properties Source, destination, weight respectively: 1 1 2
```

```
Enter edge 2 properties Source, destination, weight respectively: 1 1 1

Enter edge 3 properties Source, destination, weight respectively: 1 1 1

Vertex  Distance from Source Vertex: 0           2147483647
1                0
2                1
3                2147483647
4                2147483647
```

Assignment - 6

1. Floyed Warshall Algorithm

Algorithm:

1. n ← rows [W].
2. D^0 ← W
3. for k ← 1 to n
4. do for i ← 1 to n
5. do for j ← 1 to n
6. do $d_{ij}^{(k)}$ ← min ($d_{ij}^{(k-1)}$,$d_{ik}^{(k-1)}$+$d_{kj}^{(k-1)}$)
7. return $D^{(n)}$

Program:

```
#include <stdio.h>

#define V 4

#define INF 99999

void printSolution(int dist[][V]);

void floydWarshall(int dist[][V])
{
    int i, j, k;

    for (k = 0; k < V; k++) {
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    printSolution(dist);
}

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");
    }
}

int main()
{
    int graph[V][V] = { { 0, 5, INF, 10 },
                        { INF, 0, 3, INF },
                        { INF, INF, 0, 1 },
                        { INF, INF, INF, 0 } };

    floydWarshall(graph);
    return 0;
}
```

Output:

```
The following matrix shows the shortest distances between every pair of vertices
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0
```

Assignment – 7

1. Travel Salesman Algorithm

Algorithm:

```
1. C ({1}, 1) = 0
2. for s = 2 to n do
3. for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
4. C (S, 1) = ∞
5. for all j Є S and j ≠ 1
6. C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
7. Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

Program:

```
#include <stdio.h>

int tsp_g[10][10] = {
  {12, 30, 33, 10, 45},
  {56, 22, 9, 15, 18},
  {29, 13, 8, 5, 12},
  {33, 28, 16, 10, 3},
  {1, 4, 30, 24, 20}
};

int visited[10], n, cost = 0;
void travellingsalesman (int c) {
  int k, adj_vertex = 999;
  int min = 999;

  visited[c] = 1;

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

  for (k = 0; k < n; k++)
    {
      if ((tsp_g[c][k] != 0) && (visited[k] == 0))
      {
        if (tsp_g[c][k] < min)
          {
            min = tsp_g[c][k];
          }
        adj_vertex = k;
      }
    }
  if (min != 999)
    {
```

```
      cost = cost + min;
    }
  if (adj_vertex == 999)
    {
      adj_vertex = 0;
      printf ("%d", adj_vertex + 1);
      cost = cost + tsp_g[c][adj_vertex];
      return;
    }
  travellingsalesman (adj_vertex);
}

int main () {
  int i, j;
  n = 5;
  for (i = 0; i < n; i++)
    {
      visited[i] = 0;
    }
  printf ("\n\nShortest Path:\t");
  travellingsalesman (0);
  printf ("\n\nMinimum Cost: \t");
  printf ("%d\n", cost);
  return 0;
}
```

Output:

```
Shortest Path: 1 5 4 3 2 1
Minimum Cost: 99
```

Assignment – 8

1. Implementation of BFS

Algorithm:

1. Initially queue and visited arrays are empty.
2. Push node 0 into queue and mark it visited.
3. Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.
4. Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.
5. Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.
6. Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.
7. Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

Program:

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

#define MAX_VERTICES 50

typedef struct Graph_t {

	int V;
	bool adj[MAX_VERTICES][MAX_VERTICES];
} Graph;

Graph* Graph_create(int V)
{
	Graph* g = malloc(sizeof(Graph));
	g->V = V;

	for (int i = 0; i < V; i++) {
		for (int j = 0; j < V; j++) {
			g->adj[i][j] = false;
		}
	}

	return g;
}

void Graph_destroy(Graph* g) {
	free(g);
```

```
}

void Graph_addEdge(Graph* g, int v, int w)
{
      g->adj[v][w] = true;
}

void Graph_BFS(Graph* g, int s)
{
      bool visited[MAX_VERTICES];
      for (int i = 0; i < g->V; i++) {
            visited[i] = false;
      }

      int queue[MAX_VERTICES];
      int front = 0, rear = 0;

      visited[s] = true;
      queue[rear++] = s;

      while (front != rear) {
            s = queue[front++];
            printf("%d ", s);

            for (int adjacent = 0; adjacent < g->V;
                  adjacent++) {
                  if (g->adj[s][adjacent] && !visited[adjacent]) {
                        visited[adjacent] = true;
                        queue[rear++] = adjacent;
                  }
            }
      }
}

int main()
{
      Graph* g = Graph_create(4);
      Graph_addEdge(g, 0, 1);
      Graph_addEdge(g, 0, 2);
      Graph_addEdge(g, 1, 2);
      Graph_addEdge(g, 2, 0);
      Graph_addEdge(g, 2, 3);
      Graph_addEdge(g, 3, 3);

      printf("Following is Breadth First Traversal "
            "(starting from vertex 2) \n");
      Graph_BFS(g, 2);
      Graph_destroy(g);

      return 0;
}
```

Output:

```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

2. Implementation of DFS

Algorithm:

1. Push the root node in the Stack.
2. Loop until stack is empty.
3. Peek the node of the stack.
4. If the node has unvisited child nodes, get the unvisited child node, mark it as traversed and push it on stack.
5. If the node does not have any unvisited child nodes, pop the node from the stack.

Program:

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

struct node
{
    int vertexNumber;
    struct node *pointerToNextVertex;
};

struct Graph
{
    int numberOfVertices;

    int *visitedRecord;

    struct node **adjacencyLists;
};

struct node *createNodeForList(int v)
{
    struct node *newNode = malloc(sizeof(struct node));

    newNode->vertexNumber = v;

    newNode->pointerToNextVertex = NULL;
    return newNode;
}

void addEdgeToGraph(struct Graph *graph, int source, int destination)
{
    struct node *newNode = createNodeForList(destination);
```

```
    newNode->pointerToNextVertex = graph->adjacencyLists[source];

    graph->adjacencyLists[source] = newNode;

    newNode = createNodeForList(source);
    newNode->pointerToNextVertex = graph->adjacencyLists[destination];
    graph->adjacencyLists[destination] = newNode;
}

struct Graph *createGraph(int vertices)
{
    int i;

    struct Graph *graph = malloc(sizeof(struct Graph));

    graph->numberOfVertices = vertices;

    graph->adjacencyLists = malloc(vertices * sizeof(struct node *));

    graph->visitedRecord = malloc(vertices * sizeof(int));

    for (i = 0; i < vertices; i++)
    {
        graph->adjacencyLists[i] = NULL;
        graph->visitedRecord[i] = 0;
    }

    return graph;
}

void depthFirstSearch(struct Graph *graph, int vertexNumber)
{
    struct node *adjList = graph->adjacencyLists[vertexNumber];
    struct node *temp = adjList;

    graph->visitedRecord[vertexNumber] = 1;
    printf("%d ", vertexNumber);

    while (temp != NULL)
    {
        int connectedVertex = temp->vertexNumber;

        if (graph->visitedRecord[connectedVertex] == 0)
        {
            depthFirstSearch(graph, connectedVertex);
        }
        temp = temp->pointerToNextVertex;
    }
}

int main()
{
```

```
    int numberOfVertices, numberOfEdges, i;
    int source, destination;
    int startingVertex;

    printf("Enter Number of Vertices and Edges in the Graph: ");
    scanf("%d%d", &numberOfVertices, &numberOfEdges);
    struct Graph *graph = createGraph(numberOfVertices);

    printf("Add %d Edges of the Graph(Vertex numbering should be from 0 to %d)\n",
numberOfEdges, numberOfVertices - 1);
    for (i = 0; i < numberOfEdges; i++)
    {
        scanf("%d%d", &source, &destination);
        addEdgeToGraph(graph, source, destination);
    }

    printf("Enter Starting Vertex for DFS Traversal: ");
    scanf("%d", &startingVertex);

    if (startingVertex < numberOfVertices)
    {
        printf("DFS Traversal: ");
        depthFirstSearch(graph, startingVertex);
    }
    return 0;
}
```

Output:

```
Enter Number of Vertices and Edges in the Graph: 7 5
Add 5 Edges of the Graph(Vertex numbering should be from 0 to 6)
0 1 2 3 4 5 6 3 2 2
Enter Starting Vertex for DFS Traversal: 4
DFS Traversal: 4 5
```

Assignment – 9

1. Matrix Chain multiplication

Algorithm:

Algorithm to calculate optimal cost

```
1. n length[p]-1
2. for i ← 1 to n
3. do m [i, i] ← 0
4. for l ← 2 to n // l is the chain length
5. do for i ← 1 to n-l + 1
6. do j ← i+ l -1
7. m[i,j] ← ∞
8. for k ← i to j-1
9. do q ← m [i, k] + m [k + 1, j] + pi-1 pk pj
10. If q < m [i,j]
11. then m [i,j] ← q
12. s [i,j] ← k
13. return m and s.
```

Algorithm to parenthesize the matrix sequence

```
PRINT-OPTIMAL-PARENS (s, i, j)
1. if i=j
2. then print "A"
3. else print "("
4. PRINT-OPTIMAL-PARENS (s, i, s [i, j])
5. PRINT-OPTIMAL-PARENS (s, s [i, j] + 1, j)
6. print ")"
```

Program:

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

int MatrixChainMultiplication (int p[], int n) {
  int m[n][n];
  int i, j, k, L, q;
  for (i = 1; i < n; i++)
    m[i][i] = 0;
  for (L = 2; L < n; L++)
    {
      for (i = 1; i < n - L + 1; i++)
      {
        j = i + L - 1;
        m[i][j] = INT_MAX;
```

```
        for (k = i; k <= j - 1; k++)
          {
            q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
            if (q < m[i][j])
                  {
                  m[i][j] = q;
                  }
          }
      }
    }
  return m[1][n - 1];
}

int main () {
  int n, i;
  printf ("Enter number of matrices\n");
  scanf ("%d", &n);
  n++;
  int arr[n];
  printf ("Enter dimensions \n");
  for (i = 0; i < n; i++)
    {
      printf ("Enter d%d :: ", i);
      scanf ("%d", &arr[i]);
    }
  int size = sizeof (arr) / sizeof (arr[0]);
  printf ("Minimum number of multiplications is %d ",
        MatrixChainMultiplication (arr, size));
  return 0;
}
```

Output:

```
Enter number of matrices
4
Enter dimensions
Enter d0 :: 10
Enter d1 :: 100
Enter d2 :: 20
Enter d3 :: 5
Enter d4 :: 80
Minimum number of multiplications is 19000
```

Assignment – 10

1. Implementation N queen problem

Algorithm:

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x1, x2,....xk-1 and whether there is no other queen on the same diagonal. Using place, we give a precise solution to then n- queens problem.

```
1. Place (k, i)
2. {
3. For j ← 1 to k - 1
4. do if (x [j] = i)
5. or (Abs x [j]) - i) = (Abs (j - k))
6. then return false;
7. return true;
8. }
9. Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise
return is false. x [] is a global array whose final k - 1 values have been set. Abs (r) returns the
absolute value of r.
1. N - Queens (k, n)
2. {
3. For i ← 1 to n
4. do if Place (k, i) then
5. {
6. x [k] ← i;
7. if (k ==n) then
8. write (x [1....n));
9. else
10. N - Queens (k + 1, n);
11. }
12. }
```

Program:

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

int a[30], count = 0;
int place (int pos) {
  int i;
  for (i = 1; i < pos; i++)
```

```
    {
      if ((a[i] == a[pos]) || ((abs (a[i] - a[pos]) == abs (i - pos))))
      return 0;
    }
  return 1;
}

void print_sol (int n) {
  int i, j;
  count++;
  printf ("\n\nSolution #%d:\n", count);
  for (i = 1; i <= n; i++)
    {
      for (j = 1; j <= n; j++)
      {
        if (a[i] == j)
          printf ("Q\t");
        else
          printf ("*\t");
      }
      printf ("\n");
    }
}

void queen (int n) {
  int k = 1;
  a[k] = 0;
  while (k != 0)
    {
      a[k] = a[k] + 1;
      while ((a[k] <= n) && !place (k))
      a[k]++;
      if (a[k] <= n)
      {
        if (k == n)
          print_sol (n);
        else
          {
            k++;
            a[k] = 0;
          }
      }
      else
      k--;
    }
}

void main () {
  int i, n;
  printf ("Enter the number of Queens\n");
  scanf ("%d", &n);
  queen (n);
  printf ("\nTotal solutions=%d", count);
```

```
}
```

Output:

```
Enter the number of Queens
4
Solution #1:
* Q * *
* * * Q
Q * * *
* * Q *
Solution #2:
* * Q *
Q * * *
* * * Q
* Q * *
Total solutions=2
```

Assignment – 11

1. Graph Coloring Problem

Algorithm:

1. Arrange the vertices of the graph in some order.
2. Choose the first vertex and color it with the first color.
3. Choose the next vertex and color it with the lowest numbered color that has not been
colored on any vertices adjacent to it. If all the adjacent vertices are colored with this color,
assign a new color to it. Repeat this step until all the vertices are colored.

Program:

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

#define V 4

void printSolution (int color[]);

bool isSafe (bool graph[V][V], int color[]) {
  for (int i = 0; i < V; i++)
    for (int j = i + 1; j < V; j++)
      if (graph[i][j] && color[j] == color[i])
      return false;
  return true;
}

bool graphColoring (bool graph[V][V], int m, int i, int color[V]) {
  if (i == V)
    {
      if (isSafe (graph, color))
      {
        printSolution (color);
        return true;
      }
      return false;
    }
  for (int j = 1; j <= m; j++)
    {
      color[i] = j;
      if (graphColoring (graph, m, i + 1, color))
      return true;
      color[i] = 0;
    }
  return false;
```

```
}

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");
}

int main () {
  bool graph[V][V] = {
    {0, 1, 1, 1},
    {1, 0, 1, 0},
    {1, 1, 0, 1},
    {1, 0, 1, 0},
  };
  int m = 3;
  int color[V];
  for (int i = 0; i < V; i++)
    color[i] = 0;
  if (!graphColoring (graph, m, 0, color))
    printf ("Solution does not exist");
  return 0;
}
```

Output:

```
Solution Exists: Following are the assigned colors
1 2 3 2
```

Assignment – 12

1. Implement 15 Puzzle problem

Algorithm:

```
1. If N is odd, then puzzle instance is solvable if number of inversions is even in
the input state.
2. If N is even, puzzle instance is solvable if
      • the blank is on an even row counting from the bottom (second-last, fourth-
last, etc.) and
      number of inversions is odd.
      • the blank is on an odd row counting from the bottom (last, third-last,
fifth-last, etc.) and
      number of inversions is even.
3. For all other cases, the puzzle instance is not solvable.
```

Program:

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

int m = 0, n = 4;

int cal (int temp[10][10], int t[10][10]) {
  int i, j, m = 0;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      {
      if (temp[i][j] != t[i][j])
        m++;
      }
  return m;
}

int check (int a[10][10], int t[10][10]) {
  int i, j, f = 1;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (a[i][j] != t[i][j])
      f = 0;
  return f;
}

void main () {
  int p, i, j, n = 4, a[10][10], t[10][10], temp[10][10], r[10][10];
  int m = 0, x = 0, y = 0, d = 1000, dmin = 0, l = 0;
  printf ("\nEnter the matrix to be solved,space with zero :\n");
  for (i = 0; i < n; i++)
```

```
  for (j = 0; j < n; j++)
    scanf ("%d", &a[i][j]);
printf ("\nEnter the target matrix,space with zero :\n");
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    scanf ("%d", &t[i][j]);
printf ("\nEntered Matrix is :\n");
for (i = 0; i < n; i++)
  {
    for (j = 0; j < n; j++)
    printf ("%d\t", a[i][j]);
    printf ("\n");
  }
printf ("\nTarget Matrix is :\n");
for (i = 0; i < n; i++)
  {
    for (j = 0; j < n; j++)
    printf ("%d\t", t[i][j]);
    printf ("\n");
  }
while (!(check (a, t)))
  {
    l++;
    d = 1000;
    for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      {
        if (a[i][j] == 0)
          {
          x = i;
          y = j;
          }
      }

    for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      temp[i][j] = a[i][j];
    if (x != 0)
    {
      p = temp[x][y];
      temp[x][y] = temp[x - 1][y];
      temp[x - 1][y] = p;
    }
    m = cal (temp, t);
    dmin = l + m;
    if (dmin < d)
    {
      d = dmin;
      for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
          r[i][j] = temp[i][j];
    }
```

```
for (i = 0; i < n; i++)
for (j = 0; j < n; j++)
  temp[i][j] = a[i][j];
if (x != n - 1)
{
  p = temp[x][y];
  temp[x][y] = temp[x + 1][y];
  temp[x + 1][y] = p;
}
m = cal (temp, t);
dmin = l + m;
if (dmin < d)
{
  d = dmin;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      r[i][j] = temp[i][j];
}

for (i = 0; i < n; i++)
for (j = 0; j < n; j++)
  temp[i][j] = a[i][j];
if (y != n - 1)
{
  p = temp[x][y];
  temp[x][y] = temp[x][y + 1];
  temp[x][y + 1] = p;
}
m = cal (temp, t);
dmin = l + m;
if (dmin < d)
{
  d = dmin;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      r[i][j] = temp[i][j];
}

for (i = 0; i < n; i++)
for (j = 0; j < n; j++)
  temp[i][j] = a[i][j];
if (y != 0)
{
  p = temp[x][y];
  temp[x][y] = temp[x][y - 1];
  temp[x][y - 1] = p;
}
m = cal (temp, t);
dmin = l + m;
if (dmin < d)
{
  d = dmin;
  for (i = 0; i < n; i++)
```

```
          for (j = 0; j < n; j++)
            r[i][j] = temp[i][j];
      }
      printf ("\nCalculated Intermediate Matrix Value :\n");
      for (i = 0; i < n; i++)
      {
        for (j = 0; j < n; j++)
          printf ("%d\t", r[i][j]);
        printf ("\n");
      }
      for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        {
          a[i][j] = r[i][j];
          temp[i][j] = 0;
        }
      printf ("Minimum cost : %d\n", d);
    }
}
```

Output:

```
Enter the matrix to be solved,space with zero:
1
2
3
4
5
6
0
8
9
10
7
11
13
14
15
12
Enter the target matrix,space with zero :
1
2
3
4
5
```

```
6
7
8
9
10
11
12
13
14
15
0
Entered Matrix is :
1 2 3 4
5 6 0 8
9 10 7 11
13 14 15 12
Target Matrix is :
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
Calculated Intermediate Matrix Value :
1 2 3 4
5 6 7 8
9 10 0 11
13 14 15 12
Minimum cost : 4
Calculated Intermediate Matrix Value :
1 2 3 4
5 6 7 8
9 10 11 0
13 14 15 12
Minimum cost : 4
Calculated Intermediate Matrix Value :
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
Minimum cost : 3
```