ASSIGNMENT 6 (RITIKA GOYAL) (301401516)

Ques1:

Input: directed graph G

<u>Pseudo code:</u>

```
Is-Semiconnected (G)
    isSemiConnected = true
    Call DFS(G) to compute finishing times u.f for each u;
    Compute $G^T$ ;
    Call DFS($G^T$), but in the main loop of DFS, select nodes in order of decreasing u.f
    computed above;
    Let G’ be the component graph of G.
    Sort G’ using topological sort giving the order of vertices to be $v_1$, $v_2$, ……., $v_n$
    For  i=1 to n-1
                if ($v_i$, $v_{i+1}$) do not belongs to $E_G$
                    isSemiConnected = false;
                    return isSemiConnected;
    return isSemiConnected;
```

<u>Correctness:</u>

Loop invariant: Before every iteration, the graph is semi connected for a pair of (v1,v2) if there is a path from v1 to v2 or v2 to v1.

Initialization: In the beginning before loop starts, G’ has strongly connected components which has path from every vertex to every another vertex within the component which satisfies the loop invariant.

Maintenance: In every loop iteration, it is checked if there is an edge between vertices which are topologically sorted hence maintaining the loop invariant as it satisfies the property of semi connected graph when there is an edge between every v_i and v_{i+1}.

Termination: The loop is terminated when there exists a case when there is no edge between v_i and v_{i+1} and false is returned. After checking all the vertices, if graph have edges between all v_i and v_{i+1}, then loop ends and true is returned.

<u>Running Time:</u>

First the strongly connected component of G are calculated by calling DFS which gives running time of O(|E|+|V|).

In topological sort, DFS is used to sort the vertices which also take O(|E| + |V|).

The while loop evaluates all the consecutive vertices after topological sort and checks edges between them which take O(|n|) where n is vertices after sorting.

Running time: = T(|E| + |V|) + T(|E| + |V|) + T(|n|)

= O(|E|+|V|)

Ques2:
To prove: if all edge weights of a graph are positive, then any subset of edges that connects all vertices and the minimum total weight must be a tree.
Let us use the method of contradiction to prove the fact. Let G be the graph with all the edges of graph be positive and then any subset of edges that connects all vertices and has the minimum total weight.
Let us assume that G is not a tree.
If G is not a tree, then there exists a cycle of minimum 3 vertices in G. Let v_i¸v_{i+1}, ….., v_k be the edges of cycle.
Let m be the minimum total weight of any component G' in G.
If G' contains cycle and all the vertices are positive, then the weight will always be increasing until it reaches infinity which contradicts that m be the minimum weight of tree as we can remove an edge from cycle and still have the components connected with weight less than m.
Since m is the minimum weight is true, it contradicts that G have a cycle.
Since G does not have a cycle, it must be a tree.

The condition that all the edges have positive weight is necessary.
If negative edges are allowed, then When we remove an edge, the resulting graph would not be a minimum spanning tree. Example, if we have 3 edges in a cycle and each edge has weight -2, then total weight of cycle would be -6 and removing an edge would result in total weight to be - -4. Since -6 < -4, thus it will not be a minimum spanning tree. Another example be where two edges are positives lets say 1 and 2 and the other edge is negative lets say -1. Then total weight will be 1 + 2 + -1 = 2. If we remove the edge with negative weight, then total weight will be 1 + 2 = 3. Since 2 < 3, thus the Graph will not be minimum spanning tree.
Thus, all edges must be positive.

Ques3:

**The running time is calculated in seconds.

N ↓	Prims with binary heap min Priority queue			Prims with linear array min Priority queue		
	M = 3*n	M = $n^{1.5}$	M = n(n-1)/2	M = 3*n	M = $n^{1.5}$	M = n(n-1)/2
100	0.000149	0.000305	0.000473	0.000284	0.000535	0.001777
200	0.000147	0.000325	0.003002	0.000332	0.003361	0.015105
400	0.000369	0.001183	0.013709	0.001408	0.020946	0.151607
800	0.01282	0.002979	0.071997	0.005008	0.133317	2.11935

Source Code:

```
#include <iostream>
#include <ctime>
#include <limits>
#include <cmath>
using namespace std;

struct Adj_List_node{
		int vertex;
		int weight;
		Adj_List_node* next;
};

struct edge{
		int v1, v2;
		int edge_weight;
};

Adj_List_node* create_edge(Adj_List_node* head_ptr, int vert, int w)
{
		Adj_List_node* node = new Adj_List_node();
		node->weight = w;
		node -> vertex = vert;
		node->next = head_ptr;
		head_ptr = node;
		return node;
}

//n = num of vertices
//m = num of edges
//created adjacency list
Adj_List_node** create_edge_weighted_graph(int n, int m)
{
		Adj_List_node** adjacency_list = new Adj_List_node*[n+1];

		for (int i =0; i<n+1; i++)
				adjacency_list[i]= NULL;

		int num_edges_added = 0;
		int increment = 0;

		while(num_edges_added < m)
		{
				for(int i = 1 ; i < n+1 ; i++)
				{
						int random_weight = rand()% 7 + 1;
						if (i + increment < n)
						{
```

```
					adjacency_list[i] = create_edge(adjacency_list[i], i+1+increment ,
random_weight);
					adjacency_list[i+1+increment] = create_edge(adjacency_list[i+1+increment], i,
random_weight);
					num_edges_added++;

				}
				else if (i + increment == n)
				{
					adjacency_list[i] = create_edge(adjacency_list[i], 1 + increment,
random_weight);
					adjacency_list[1 + increment] = create_edge(adjacency_list[1 + increment], i,
random_weight);
					num_edges_added++;
				}
				if(num_edges_added == m)
					break;
			}

			increment++;
		}
		return adjacency_list;
}

//cretes adjecency matrix of graph when adjecency list is its arguement.
int** create_adjacency_matrix(Adj_List_node** adjacency_list, int n)
{
		int** matrix = new int* [n+1];

		for(int i = 0; i < n+1; i++)
			matrix[i] = new int [n+1];

		for(int i =0; i<n+1 ; i++)
			for(int j =0; j< n+1; j++)
				matrix[i][j] = 0;


		Adj_List_node* temp;

		for(int i =1; i<n+1; i++)
		{
			temp = adjacency_list[i];

			while(temp != NULL)
			{
				matrix[i][temp->vertex] = temp->weight;
				matrix[temp->vertex][i] = temp->weight;
				temp = temp->next;
			}
```

```
        }
        return matrix;
}

//minimum priority ques using binary heap
void min_heapify(edge* array_edge, int array_size, int i)
{
        int left = 2*i;
        int right = 2*i + 1;
        int min = 0;

        if(left <= array_size && array_edge[left].edge_weight < array_edge[i].edge_weight)
                min = left;
        else
                min = i;
        if(right<= array_size && array_edge[right].edge_weight < array_edge[min].edge_weight)
                min = right;

        if(min != i)
        {
                //exchange contents of A[i] and A[min]
                edge temp;
                temp.edge_weight = array_edge[i].edge_weight;
                temp.v1 = array_edge[i].v1;
                temp.v2 = array_edge[i].v2;

                array_edge[i].edge_weight = array_edge[min].edge_weight;
                array_edge[i].v1 = array_edge[min].v1;
                array_edge[i].v2 = array_edge[min].v2;

                array_edge[min].edge_weight = temp.edge_weight;
                array_edge[min].v1 = temp.v1;
                array_edge[min].v2 = temp.v2;

                min_heapify(array_edge, array_size, min);
        }
}

void decrease_key (edge* array_edge, int array_size, int i, edge key)
{
        if(key.edge_weight > array_edge[i].edge_weight)
                return;

        array_edge[i] = key;
        array_edge[i].edge_weight = key.edge_weight;
        array_edge[i].v1 = key.v1;
        array_edge[i].v2 = key.v2;

        while(i>0 && array_edge[i/2].edge_weight > array_edge[i].edge_weight)
        {
                edge temp;
                temp.edge_weight = array_edge[i].edge_weight;
                temp.v1 = array_edge[i].v1;
                temp.v2 = array_edge[i].v2;

                array_edge[i].edge_weight = array_edge[i/2].edge_weight;
                array_edge[i].v1 = array_edge[i/2].v1;
                array_edge[i].v2 = array_edge[i/2].v2;

                array_edge[i/2].edge_weight = temp.edge_weight;
                array_edge[i/2].v1 = temp.v1;
                array_edge[i/2].v2 = temp.v2;

                i = i/2;
        }
}

void min_heap_insert(edge* array_edge, int array_size, edge key)
{
        array_size = array_size + 1;
        array_edge[array_size-1].edge_weight = std::numeric_limits<int>::max();

        decrease_key(array_edge, array_size, array_size-1, key);
}

edge minimum_element (edge* array_edge, int array_size)
{
        return array_edge[0];
}

edge minimum_extract(edge* array_edge, int array_size)
{
        edge min = array_edge[0];
        min.edge_weight = array_edge[0].edge_weight;
        min.v1 = array_edge[0].v1;
        min.v2 = array_edge[0].v2;


        array_edge[0]= array_edge[array_size - 1];
        array_edge[0].edge_weight = array_edge[array_size -1].edge_weight;
        array_edge[0].v1 = array_edge[array_size -1].v1;
        array_edge[0].v2 = array_edge[array_size -1].v2;

        array_size = array_size -1;

        min_heapify(array_edge, array_size, 0);

        return min;
```

```
}

Adj_List_node** prim_with_binary_heap(Adj_List_node** adjacency_list, int num_vertices, int num_edges, int
source_vertex)
{
	Adj_List_node** return_array = new Adj_List_node* [num_vertices+1];
	for(int i = 0; i<num_vertices+1; i++ )
		return_array[i] = NULL;

	int vertices_checked[num_vertices+1];

	edge priority_queue[2*num_edges];
	int size = 0;

	int current_vertex = source_vertex;
	int new_vertex;
	Adj_List_node* temp;
	edge temp_edge;

	for(int i =0; i<num_vertices+1; i++)
		vertices_checked[i] =0;

	int i =0;
	while(i<num_vertices)
	{
		if(vertices_checked[current_vertex] == 0)
		{
			vertices_checked[current_vertex] = 1;
			temp = adjacency_list[current_vertex];

			while(temp != NULL)
			{
				temp_edge.edge_weight = temp->weight;
				temp_edge.v1 = current_vertex;
				temp_edge.v2 = temp->vertex;

				if(!vertices_checked[temp_edge.v2])
				{
					min_heap_insert(priority_queue, size, temp_edge);
					size = size+1;
				}

				temp = temp->next;
			}

			temp_edge = minimum_extract(priority_queue, size);
			size = size-1;

			new_vertex = temp_edge.v2;
```

```
                                        current_vertex = temp_edge.v1;

                                        if(vertices_checked[new_vertex] == 0)
                                        {
                                                        return_array[current_vertex] = create_edge(return_array[current_vertex],
new_vertex, temp_edge.edge_weight);
                                                        return_array[new_vertex] = create_edge(return_array[new_vertex],
current_vertex, temp_edge.edge_weight);
                                        }

                                        current_vertex = new_vertex;
                                        i++;
                        }
                        else
                        {
                                        temp_edge = minimum_extract(priority_queue, size);
                                        size = size-1;
                                        new_vertex = temp_edge.v2;
                                        current_vertex = temp_edge.v1;
                                        if(vertices_checked[new_vertex] == 0)
                                        {
                                                        return_array[current_vertex] = create_edge(return_array[current_vertex],
new_vertex, temp_edge.edge_weight);
                                                        return_array[new_vertex] = create_edge(return_array[new_vertex],
current_vertex, temp_edge.edge_weight);
                                        }

                                        current_vertex = new_vertex;

                        }
        }
        return return_array;
}

//minimum priority queue using linear array
void enqueue_array(edge* min_priority_array, int size, edge new_value)
{
        min_priority_array[size] = new_value;
        min_priority_array[size].edge_weight = new_value.edge_weight;
        min_priority_array[size].v1 = new_value.v1;
        min_priority_array[size].v2 = new_value.v2;

        size = size+1;
}

int minimum_array(edge* min_priority_array, int size)
{
        int min = min_priority_array[0].edge_weight;
        int index = 0;
        for(int i =1; i< size; i++)
        {
                if(min_priority_array[i].edge_weight < min)
                {
                        min = min_priority_array[i].edge_weight;
                        index = i;
                }

        }
        return index;
}

edge minimum_extract_array(edge* min_priority_array, int size)
{
        int i= minimum_array(min_priority_array, size);

        edge temp;
        temp.edge_weight = min_priority_array[i].edge_weight;
        temp.v1 = min_priority_array[i].v1;
        temp.v2 = min_priority_array[i].v2;

        min_priority_array[i].edge_weight = min_priority_array[size-1].edge_weight;
        min_priority_array[i].v1 = min_priority_array[size-1].v1;
        min_priority_array[i].v2 = min_priority_array[size-1].v2;

        return temp;
}

void decrease_key_array (edge* min_priority_array, int index, int key)
{
        if(min_priority_array[index].edge_weight > key)
                min_priority_array[index].edge_weight = key;
}

int** prims_with_linear_array(int** adjacency_matrix, int num_vertices, int num_edges, int source_vertex)
{
        int** return_array = new int* [num_vertices+1];

        for(int i = 0; i < num_vertices+1; i++)
                return_array[i] = new int [num_vertices+1];

        for(int i =0; i<num_vertices+1 ; i++)
                for(int j =0; j< num_vertices+1; j++)
                        return_array[i][j] = 0;

        int vertices_checked[num_vertices+1];

        edge priority_queue[2*num_edges];
```

```
int size = 0;

int current_vertex = source_vertex;
int new_vertex;
int* temp;
edge temp_edge;

for(int i =0; i<num_vertices+1; i++)
        vertices_checked[i] = 0;

int i =0;
while(i<num_vertices)
{
        if(vertices_checked[current_vertex] == 0)
        {
                vertices_checked[current_vertex] = 1;
                temp = adjacency_matrix[current_vertex];

                for (int j =1; j<num_vertices+1; j++)
                {
                        if(adjacency_matrix[current_vertex][j] != 0)
                        {
                                temp_edge.edge_weight = adjacency_matrix[current_vertex][j];
                                temp_edge.v1 = current_vertex;
                                temp_edge.v2 = j;

                                if(!vertices_checked[temp_edge.v2])
                                {
                                        enqueue_array(priority_queue, size, temp_edge);
                                        size = size+1;
                                }
                        }
                }

                temp_edge = minimum_extract_array(priority_queue, size);
                size = size-1;

                new_vertex = temp_edge.v2;
                current_vertex = temp_edge.v1;

                if(vertices_checked[new_vertex] == 0)
                {
                        return_array[current_vertex][new_vertex] = temp_edge.edge_weight;
                        return_array[new_vertex][current_vertex] = temp_edge.edge_weight;

                }

                current_vertex = new_vertex;
                i++;
                }
                else
                {
                        temp_edge = minimum_extract_array(priority_queue, size);
                        size = size-1;
                        new_vertex = temp_edge.v2;
                        current_vertex = temp_edge.v1;
                        if(vertices_checked[new_vertex] == 0)
                        {
                                return_array[current_vertex][new_vertex] = temp_edge.edge_weight;
                                return_array[new_vertex][current_vertex] = temp_edge.edge_weight;
                        }

                        current_vertex = new_vertex;
                }
        }
        return return_array;
}

int main()
{
        for (int n = 100; n<=800; n = 2*n)
        {
                int num[3];
                num[0] = 3*n;
                num[1] = pow(n,1.5);
                num[2] = (n*(n-1))/2;
                cout<<"n = "<<n<<": "<<endl;

                int total = 0;
                for(int m = num[total]; total<3; total++)
                {
                        m = num[total];

                  Adj_List_node** graph  = create_edge_weighted_graph(n, m);

                  int** adjacency_matrix = create_adjacency_matrix(graph, n);

                  clock_t prims_binary = clock();
                  Adj_List_node** tree = prim_with_binary_heap(graph, n, m, 1);
                  prims_binary = clock() - prims_binary;

                  clock_t prims_array = clock();
                  int** tree_matrix = prims_with_linear_array(adjacency_matrix, n, m, 1);
                  prims_array = clock() - prims_array;

                  cout<<"m = "<<m<<" Prims for binary heap: " <<(float)prims_binary/
CLOCKS_PER_SEC<<"seconds"<<"\t Prims for array: "<<(float)prims_array/ CLOCKS_PER_SEC<<" seconds"<<endl;
```

```
                //deleting matrix
                for(int i =0; i< n+1; i++)
                {
                        delete [] adjacency_matrix[i];
                }
                delete [] adjacency_matrix;


                //deleting the graph
                for(int i = 1; i<=n ; i++)
                {
                        Adj_List_node* temp1 = graph[i];
                        Adj_List_node* temp2 = graph[i] -> next;
                        graph[i] = NULL;
                        Adj_List_node* last_node;
                        while(temp2!= NULL)
                        {
                                temp1->next = NULL;
                                delete temp1;
                                temp1 = temp2;
                                last_node = temp1;
                                temp2 = temp2 -> next;

                        }
                        delete last_node;
                }
                delete [] graph;


                //deleting tree matrix
                for(int i =0; i< n+1; i++)
                {
                        delete [] tree_matrix[i];
                }
                delete [] tree_matrix;
              }
              cout<<endl;
        }

        return 0;
}
```

Ques4:

Structure of optimal solution:

Opt (i, s, v) finds the length of shortest path from s -> v with at most i arcs.

The cases are:

- When there are no arcs and s = v, then 0 is returned because node is at its goal.
- When there are no arcs but $s \neq v$, then s and v does not have any path in between.
- Shortest path from s->v has at most i-1 arcs, opt(i, s, v) = opt(i-1, s, v)
- Shortest path from s-> v has i arcs, opt(i, s, v) = $\min_{(s, w) \varepsilon E}$ {opt(i-1, w, v) + w(s, w)}

In the distance vector algorithm, the above structure is used to calculate the shortest path from source node s to every other node v.

Bellman equation:

$$
\begin{aligned}
Opt(i, s, v) &= 0 \quad \text{if } i = 0 \text{ and } s = v; \\
&= \infty \quad \text{if } i = 0 \text{ and } s \neq v; \\
&= \min \{ opt(i-1,s,v), \min_{(s, w) \varepsilon E} \{opt(i-1, w, v) + w(s, w)\} \} \text{ if } i > 0
\end{aligned}
$$

Pseudocode:

Input: Graph G is represented by adjacency list and s is source node.

Distance-Vector-Algorithm (G, s)

```
    s.d =0;
    for i = 1 to n-1
        for every v belongs to V
            for each arc (w,v) belongs to E
                if v.d > w.d + w(w, v)
                    v.d = w.d + w(w, v);

    for each arc (u, v) belongs to E
        if v.d > u.d + w(u, v) then return False;
    return true;
```

Running time:

The outer for loop runs |V| times as each vertex is checked, and the inner two loops checks each edge and thus it runs |E|. The last for loop before returns runs |E| times and other operations are constant. Thus,

$$
\begin{aligned}
\text{Running time} &= T(|V||E|) + T(|E|) + T(1) \\
&= O(|V||E|) + O(|E|) + O(1) \\
&= O(|V||E|)
\end{aligned}
$$

Ques5:

**The running time is calculated in seconds.

N ↓	Dijkstra with binary heap min Priority queue			Dijkstra with linear array min Priority queue		
	M = 3*n	M = $n^{1.5}$	M = n(n-1)/2	M = 3*n	M = $n^{1.5}$	M = n(n-1)/2
100	0.00004	0.000049	0.000133	0.002463	0.002874	0.004293
200	0.000072	0.000118	0.000663	0.019066	0.020934	0.043719
400	0.000146	0.000309	0.005069	0.180293	0.233735	0.317571
800	0.0003	0.000876	0.062022	1.66563	1.6089	2.65873

Source Code:

```
#include <iostream>
#include <ctime>
#include <limits>
#include <cmath>
using namespace std;

struct Adj_List_node{
        int vertex;
        int weight;
        Adj_List_node* next;
};

struct edge{
        int v1;
        int distance;
};

Adj_List_node* create_edge(Adj_List_node* head_ptr, int vert, int w)
{
        Adj_List_node* node = new Adj_List_node();
        node->weight = w;
        node -> vertex = vert;
        node->next = head_ptr;
        head_ptr = node;
        return node;
}

//n = num of vertices
//m = num of edges
```

```
//created adjacency list
Adj_List_node** create_edge_weighted_graph(int n, int m)
{
	Adj_List_node** adjacency_list = new Adj_List_node*[n+1];

	for (int i =0; i<n+1; i++)
		adjacency_list[i]= NULL;

	int num_edges_added = 0;
	int increment = 0;

	while(num_edges_added < m)
	{
		for(int i = 1 ; i < n+1 ; i++)
		{
			int random_weight = rand()% 7 + 1;
			if (i + increment < n)
			{
				adjacency_list[i] = create_edge(adjacency_list[i], i+1+increment ,
random_weight);
				adjacency_list[i+1+increment] = create_edge(adjacency_list[i+1+increment], i,
random_weight);
				num_edges_added++;

			}
			else if (i + increment == n)
			{
				adjacency_list[i] = create_edge(adjacency_list[i], 1 + increment,
random_weight);
				adjacency_list[1 + increment] = create_edge(adjacency_list[1 + increment], i,
random_weight);
				num_edges_added++;
			}
			if(num_edges_added == m)
				break;
		}

		increment++;
	}
	return adjacency_list;
}

//cretes adjecency matrix of graph when adjecency list is its arguement.
int** create_adjacency_matrix(Adj_List_node** adjacency_list, int n)
{
	int** matrix = new int* [n+1];

	for(int i = 0; i < n+1; i++)
		matrix[i] = new int [n+1];
```

```
        for(int i =0; i<n+1 ; i++)
                for(int j =0; j< n+1; j++)
                        matrix[i][j] = 0;


        Adj_List_node* temp;

        for(int i =1; i<n+1; i++)
        {
                temp = adjacency_list[i];

                while(temp != NULL)
                {
                        matrix[i][temp->vertex] = temp->weight;
                        matrix[temp->vertex][i] = temp->weight;
                        temp = temp->next;
                }
        }
        return matrix;
}

//minimum priority ques using binary heap
void min_heapify(edge* array_edge, int array_size, int i)
{
        int left = 2*i;
        int right = 2*i + 1;
        int min = 0;

        if(left <= array_size && array_edge[left].distance < array_edge[i].distance)
                min = left;
        else
                min = i;
        if(right<= array_size && array_edge[right].distance < array_edge[min].distance)
                min = right;

        if(min != i)
        {
                //exchange contents of A[i] and A[min]
                edge temp;
                temp.distance = array_edge[i].distance;
                temp.v1 = array_edge[i].v1;

                array_edge[i].distance = array_edge[min].distance;
                array_edge[i].v1 = array_edge[min].v1;

                array_edge[min].distance = temp.distance;
                array_edge[min].v1 = temp.v1;
```

```
		min_heapify(array_edge, array_size, min);
	}
}

void decrease_key (edge* array_edge, int array_size, int i, edge key)
{
	if(key.distance > array_edge[i].distance)
		return;

	array_edge[i] = key;
	array_edge[i].distance = key.distance;
	array_edge[i].v1 = key.v1;

	while(i>0 && array_edge[i/2].distance > array_edge[i].distance)
	{
		edge temp;
		temp.distance = array_edge[i].distance;
		temp.v1 = array_edge[i].v1;

		array_edge[i].distance = array_edge[i/2].distance;
		array_edge[i].v1 = array_edge[i/2].v1;

		array_edge[i/2].distance = temp.distance;
		array_edge[i/2].v1 = temp.v1;

		i = i/2;
	}
}

void min_heap_insert(edge* array_edge, int array_size, edge key)
{
	array_size = array_size + 1;
	array_edge[array_size-1].distance = std::numeric_limits<int>::max();

	decrease_key(array_edge, array_size, array_size-1, key);
}

edge minimum_element (edge* array_edge, int array_size)
{
	return array_edge[0];
}

edge minimum_extract(edge* array_edge, int array_size)
{
	edge min = array_edge[0];
	min.distance = array_edge[0].distance;
	min.v1 = array_edge[0].v1;

	array_edge[0]= array_edge[array_size - 1];
```

```
        array_edge[0].distance = array_edge[array_size -1].distance;
        array_edge[0].v1 = array_edge[array_size -1].v1;

        array_size = array_size -1;

        min_heapify(array_edge, array_size, 0);

        return min;
}

Adj_List_node** Dijkstra_with_binary_heap(Adj_List_node** adjacency_list, int num_vertices, int num_edges, int
source_vertex)
{
        Adj_List_node** return_array = new Adj_List_node*[num_vertices+1];

        int S_visited[num_vertices+1];
        edge priority_queue[num_vertices+1];
        int size =0;
        int distance[num_vertices+1];
        int parent[num_vertices+1];

        for(int i =1; i<num_vertices+1; i++)
        {
                distance[i] = std::numeric_limits<int>::max();
                parent[i] = 0;
                S_visited[i] = 0;
                return_array[i] = NULL;
                priority_queue[size].distance = std::numeric_limits<int>::max();
                priority_queue[size].v1 = i;
                size++;
        }

        distance[source_vertex] = 0;
        parent[source_vertex] = -1;

        while(size > 0)
        {
                edge min_edge = minimum_extract(priority_queue, size);
                size--;
                S_visited[min_edge.v1] = 1;

                Adj_List_node* temp = adjacency_list[min_edge.v1];
                while(temp!=NULL)
                {
                        int u = min_edge.v1;
                        int v = temp->vertex;
                        int w = temp->weight;

                        if(S_visited[v] == 0)
                        {
                                if(distance[v]>distance[u] + w && distance[u] !=
std::numeric_limits<int>::max())
                                {
                                        distance[v] = distance[u] + w;
                                        parent[v] = u;

                                        edge changed;
                                        changed.distance = distance[v];
                                        changed.v1 = v;
                                        decrease_key(priority_queue,size,size,changed);
                                }
                        }
                        temp= temp->next;
                }
        }

        for(int i =1; i<num_vertices+1;i++)
        {
                if(parent[i] > 0)
                {
                        return_array[i] = create_edge(return_array[i], parent[i], distance[i]);
                        return_array[parent[i]] = create_edge(return_array[parent[i]],i,distance[i]);
                }
        }

        return return_array;
}

//minimum priority queue using linear array
void enqueue_array(edge* min_priority_array, int size, edge new_value)
{
        min_priority_array[size] = new_value;
        min_priority_array[size].distance = new_value.distance;
        min_priority_array[size].v1 = new_value.v1;
        size = size+1;
}

int minimum_array(edge* min_priority_array, int size)
{
        int min = min_priority_array[0].distance;
        int index = 0;

        for(int i =1; i< size; i++)
        {
                if(min_priority_array[i].distance < min)
                {
                        min = min_priority_array[i].distance;
                        index = i;
```

```
				}

		}
		return index;
}

edge minimum_extract_array(edge* min_priority_array, int size)
{
		int i= minimum_array(min_priority_array, size);

		edge temp;
		temp.distance = min_priority_array[i].distance;
		temp.v1 = min_priority_array[i].v1;

		min_priority_array[i].distance = min_priority_array[size-1].distance;
		min_priority_array[i].v1 = min_priority_array[size-1].v1;

		return temp;
}

void decrease_key_array (edge* min_priority_array, int index, edge key)
{
		if(min_priority_array[index].distance > key.distance)
		{
				min_priority_array[index].distance = key.distance;
				min_priority_array[index].v1 = key.v1;
		}
}

int** Dijkstra_with_linear_array(int** adjacency_matrix, int num_vertices, int num_edges, int source_vertex)
{
		int** return_array = new int* [num_vertices+1];

		for(int i = 0; i < num_vertices+1; i++)
				return_array[i] = new int [num_vertices+1];

		for(int i =0; i<num_vertices+1 ; i++)
				for(int j =0; j< num_vertices+1; j++)
						return_array[i][j] = 0;

		int S_visited[num_vertices+1];
		edge priority_queue[num_vertices+1];
		int size =0;
		int distance[num_vertices+1];
		int parent[num_vertices+1];

		for(int i =1; i<num_vertices+1; i++)
		{
				distance[i] = std::numeric_limits<int>::max();
```

```
                parent[i] = 0;
                S_visited[i] = 0;
                priority_queue[size].distance = std::numeric_limits<int>::max();
                priority_queue[size].v1 = i;
                size++;
        }

        distance[source_vertex] = 0;
        parent[source_vertex] = -1;

        while(size > 0)
        {
                edge min_edge = minimum_extract_array(priority_queue, size);
                size--;
                S_visited[min_edge.v1] = 1;


                for(int i =1; i<num_vertices+1; i++)
                {
                        for(int j =1; j<num_vertices+1;j++)
                        {
                                if(adjacency_matrix[i][j] != 0)
                                {
                                        int u = i;
                                        int v = j;
                                        int w = adjacency_matrix[i][j];

                                        if(S_visited[v] == 0)
                                        {
                                                if(distance[v]>distance[u] + w && distance[u] !=
std::numeric_limits<int>::max())
                                                {
                                                        distance[v] = distance[u] + w;
                                                        parent[v] = u;

                                                        edge changed;
                                                        changed.distance = distance[v];
                                                        changed.v1 = v;
                                                        decrease_key_array(priority_queue,size,changed);
                                                }
                                        }
                                }
                        }
                }
        }

        for(int i =1 ; i<num_vertices+1; i++)
        {
                for(int j =0 ; j<num_vertices+1; j++)
                {
                        if(parent[i] == j)
                        {
                                return_array[i][parent[i]] = distance[i];
                                return_array[parent[i]][i] = distance[i];
                        }
                }
        }

        return return_array;
}


int main()
{
        for (int n = 100; n<=800; n = 2*n)
        {
                int num[3];
                num[0] = 3*n;
                num[1] = pow(n,1.5);
                num[2] = (n*(n-1))/2;
                cout<<"n = "<<n<<": "<<endl;

                int total = 0;
                for(int m = num[total]; total<3; total++)
                {
                        m = num[total];

                  Adj_List_node** graph  = create_edge_weighted_graph(n, m);

                  int** adjacency_matrix = create_adjacency_matrix(graph, n);

                  clock_t dij_binary = clock();
                  Adj_List_node** tree = Dijkstra_with_binary_heap(graph, n, m, 1);
                  dij_binary = clock() - dij_binary;

                  clock_t dij_array = clock();
                  int** tree_matrix = Dijkstra_with_linear_array(adjacency_matrix, n, m, 1);
                  dij_array = clock() - dij_array;

                  cout<<"m = "<<m<<" Dijkstra for binary heap: " <<(float)dij_binary/
CLOCKS_PER_SEC<<"seconds"<<"\t Dijkstra for array: "<<(float)dij_array/ CLOCKS_PER_SEC<<" seconds"<<endl;

                  //deleting matrix
                  for(int i =0; i< n+1; i++)
                  {
                        delete [] adjacency_matrix[i];
                  }
                  delete [] adjacency_matrix;
                //deleting the graph
                for(int i = 1; i<=n ; i++)
                {
                        Adj_List_node* temp1 = graph[i];
                        Adj_List_node* temp2 = graph[i] -> next;
                        graph[i] = NULL;
                        Adj_List_node* last_node;
                        while(temp2!= NULL)
                        {
                                temp1->next = NULL;
                                delete temp1;
                                temp1 = temp2;
                                last_node = temp1;
                                temp2 = temp2 -> next;

                        }
                        delete last_node;
                }
                delete [] graph;


                //deleting tree matrix
                for(int i =0; i< n+1; i++)
                {
                        delete [] tree_matrix[i];
                }
                delete [] tree_matrix;
              }
              cout<<endl;
        }

        return 0;

}
```

Ques6:

```
Extended-shortest-Paths ($L^{(r-1)}$, $\Pi^{(r-1)}$,W)
    Let $L^{(r)}$ [1….n, 1….n] be a new n x n matrix
    Let $\Pi^{(r)}$ [1….n, 1…..n] be a new n x n matrix
    For i =1 to n do
        For j = 1 to n do
            $L^{(r)}$ [i,j] = ∞;
            $\Pi^{(r)}$ [I,j] = nil;
        For k =1 to n do
            $L^{(r)}$ [i,j] = min {$L^{(r)}$[i,j], $L^{(r-1)}$[i,k] + W[k,j]};
            If ($L^{(r-1)}$[i,k] + W[k,j] < $L^{(r)}$ [i,j])
                $\Pi^{(r)}$[i,j] = $\Pi^{(r-1)}$ [k,j];
    Return $L^{(r)}$, $\Pi^{(r)}$;
```

```
Slow-All-Pairs-Shortest-Paths(W)
    $L^{(1)}$ = W;
    For int i =1 to n
        For int j = 1 to n
            If (i,j) belongs to E
                $\Pi^{(1)}$ [i,j] = i;
            Else
                $\Pi^{(1)}$ [i,j] = nil;
    For r = 2 to n-1
        Let $L^{(r)}$ be a new n x n matrix;
        $L^{(r)}$, $\Pi^{(r)}$  = Extended-shortest-Paths($L^{(r-1)}$, $\Pi^{(r-1)}$,W);
    Return $L^{(n-1)}$, $\Pi^{(n-1)}$;
```

Ques7:

```
Faster-All-Pairs-Shortest-Paths (W)
    L = W; r =1;
    While r < n-1
        L = Extended-Shortest-Paths(W, W);
        W = L
    return L;
```

Space complexity: Since only L is used and matrix multiplication in each loop is overwritten in L which is n x n matrix. Hence, the space complexity remains $\Theta(n^2)$ as just n x n matrix is used.