Graphs & Dijkstra's Algorithm

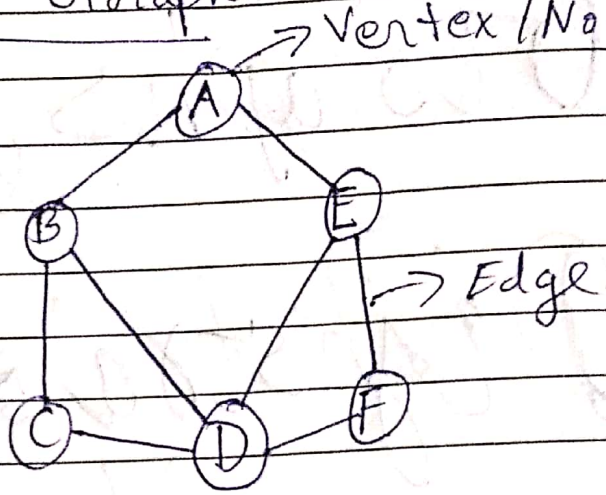
Graph

A Graph is a non-linear data structure consisting of vertices and edges.

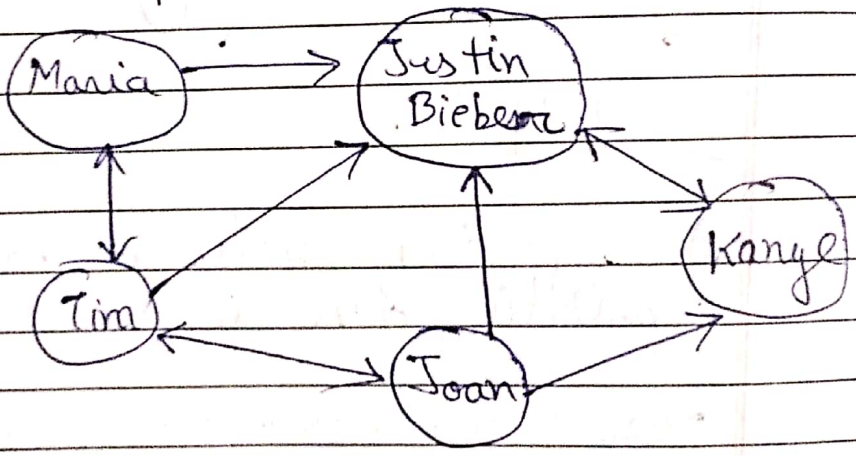
~~set of vertices or nodes or points, together with a set of~~
Together with a set of unordered pairs of these vertices form an undirected graph or a set of ordered pairs for a directed graph.

vertex = node
Edge - connection between nodes
weighted / Unweighted - values assigned to distances between vertices.
Directed / Undirected - directions assigned to distanced between vertices.

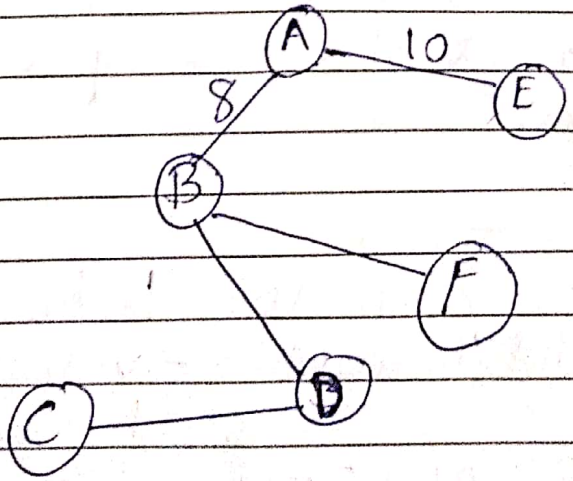
Undirected Graph:



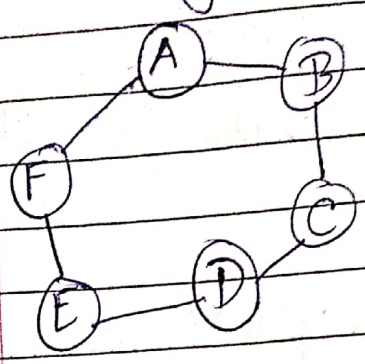
Directed Graph:



Weighted Graph:



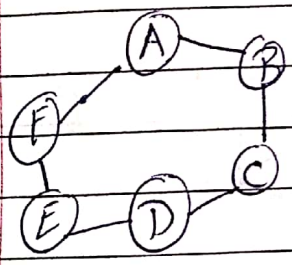
* Storing Graph in Adjacency Matrix :-



-	A	B	C	D	E	F
A	0	1	0	0	0	1
B	1	0	1	0	0	0
C	0	1	0	1	0	0
D	0	0	1	0	1	0
E	0	0	0	1	0	1
F	1	0	0	0	1	0

where, 1 → means connected
2 → means non connected

* Storing Graph in Adjacency List :-



{
A : ["B", F],
B : [A , C],
C : [B , D],
D : [C , E],
E : [D , F],
F : [E , A]
}

Here, Adjacency List represents a graph as an array of object which would act as a HashMap or HashTable.

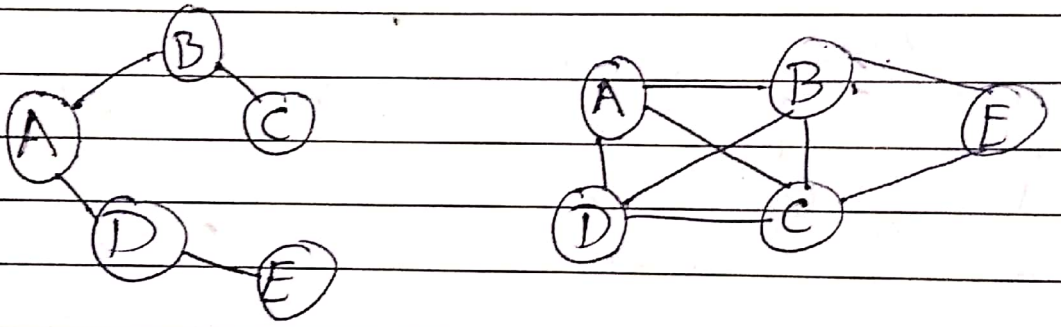
An adjacency list is a collection of unordered lists used to represent a finite undirected graph. Each unordered list within an adjacency list describes the set of neighbors of a particular vertex in the graph.

* Adjacency List is better than matrix?

The adjacency matrix and adjacency list have the same time and space complexity.

However, if the graph is sparse, we need less space to represent the graph.
Therefore, an adjacency list is more space efficient than adjacency matrix when we work on sparse graphs.

= Sparse Graph Vs Dense Graph



Sparse Graph Dense Graph

Sparse Graph: Sparse graph is a graph in which the number of edges is close to the minimal number of edges.

Dense Graph: Dense graph is a graph in which the number of edges is close to the maximal number of edges.

* Code Implementation :-

```
class Graph {
  constructor() {
    this.adjacencyList = {};
  }

  // Add Vertex
  addVertex(vertex) {
      if(!this.adjacencyList[vertex])
        this.adjacencyList[vertex] = [];
  }

  // Add Edge
   addEdge(vertex1, vertex2) {
      if(this.adjacencyList[vertex1])
        this.adjacencyList[vertex1].push(vertex2);
    if(this.adjacencyList[vertex2])
    this.adjacencyList[vertex2].push(vertex1);
  }

  // Removed Edge
 removeEdge(vertex1,vertex2){
   if(this.adjacencyList[vertex1])
   this.adjacencyList[vertex1]= this.adjacencyList[
vertex1].filter(element => element !== vertex2);
   if(this.adjacencyList[vertex2])
    this.adjacencyList[vertex2]= this.adjacencyList
[vertex2].filter(element=>element !== vertex1);
```

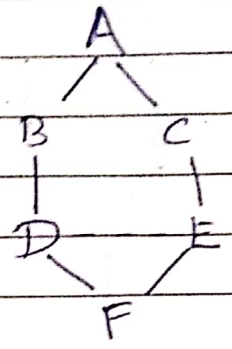
```
// remove vertex
removeVertex(vertex) {
    if (this.adjacencyList[vertex]) {
    let adjacentVertexArr = this.adjacencyList[vertex];

    for(let i=0; i<adjacentVertexArr.length; i++) {
        let adjacentVertex = adjacentVertexArr[i];
        this.removeEdge(vertex, adjacentVertex);
    }
        delete this.adjacencyList[vertex];
    }
}
}
const graph = new Graph();

//Adding Vertex
graph.addVertex("Tokyo");
graph.addVertex("Dallas");
graph.addVertex("Aspen");

// Adding Edge Between Vertices
graph.addEdge("Tokyo", "Dallas");
graph.addEdge("Dallas, Aspen");

// Removing Edge Between Vertices
graph.removeEdge("Dallas", "Tokyo");

// Removing a particular Vertex & their adjacent values
graph.removeVertex("Dallas");
```

* Graph Traversal Algorithm:

ii) Depth First Traversal / Search (DFS):

In DFS, Explore as far as possible down one branch before back tracking.

Ex:



Undirected Graph

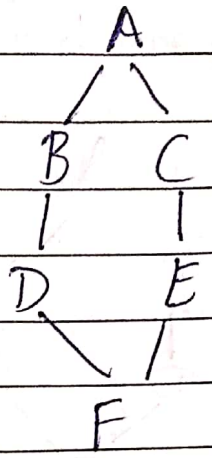
All Possible Solutions:

['A', 'B', 'D', 'E', 'C', 'F']
['A', 'B', 'D', 'F', 'E', 'C']
['A', 'C', 'E', 'D', 'B', 'F']
['A', 'C', 'E', 'F', 'D', 'B']

(iii) Breadth First Traversal / Search (BFS):

In BFS, Visit neighbors (Adjacent Vertices) at current depth first!

Ex:



Undirected Graph

All Possible Solutions:

['A', 'B', 'C', 'D', 'E', 'F']

['A', 'C', 'B', 'E', 'D', 'F'] → reverse

Note:

- Here, we're taking the neighbors of A i.e. ['B', 'C']
- Then, taking the neighbors of B i.e. ['D', 'E']
- Then, taking the neighbors of D i.e. ['F']
- Then, taking the neighbors of E i.e. ['F']
- Since, the neighbors of D & E is same i.e. 'F'. So, we take 'F' only once.

→ Code Implementation:-

```
class Graph{
    constructor(){
    this.adjacencyList = {};
    }
    // Add Vertex
    addVertex(vertex){
        if(!this.adjacencyList[vertex])
            this.adjacencyList[vertex]=[];
    }
    // Add Edge
    addEdge(vertex1, vertex2){
        if(this.adjacencyList[vertex1])
            this.adjacencyList[vertex1].push(vertex2);
        if(this.adjacencyList[vertex2])
        this.adjacencyList[vertex2].push(vertex1);
    }

// Depth First Traversal Recursion
    depthFirstRecursive(start){
        let result = [],
            visited = {},
            adjacencyList = this.adjacencyList;

    (function dfs(vertex){
        if(!adjacencyList[vertex])
            return null;
        result.push(vertex);
        visited[vertex]=true;
```

```
        for(let i=0; i<adjacencyList[vertex].length; i++){
            let adjacentVertex = adjacencyList[vertex][i];
            if(!visited[adjacentVertex]){
                dfs(adjacentVertex);
            }
        }
    })(start);
    return result;
}

// Depth First Traversal Iteration.
depthFirstIterative(start){
    let stack = [start],
        result = [],
        visited = {};
    visited[start] = true;

// till their is some thing in the stack
    while(stack.length){
        let currentVertex = stack.pop();
        result.push(currentVertex);

        this.adjacencyList[currentVertex].forEach(
        neighbour => {
            if(!visited[neighbour]){
                visited[neighbour] = true;
                stack.push(neighbour);
            }
        })
    return result;
}
```

```
// Breadth First Traversal
breadthFirstSearch(start){
    let queue = [start],
        result = [],
        visited = {};

    visited[start] = true;

    while(queue.length){
        let currentVertex = queue.shift();
        result.push(currentVertex);

    this.adjacencyList[currentVertex].reverse().
    forEach(neighbour => {
            if(!visited[neighbour]){
                visited[neighbour] = true;
                queue.push(neighbour);
            }
        })
    }

    return result;
  }
}

const graph = new Graph();
// Adding vertex
graph.addVertex("A");
"   .   "       ('B');
"   .   "       ('C');
"   .   "       ('D');
"   .   "       ('E');
                ('F');
```

```
//Adding Edge between Vertices
graph.addEdge('A', 'B');
   "    "    ('A', 'C');
   "    "    ('B', 'D');
   "    "    ('C', 'E');
   "    "    ('D', 'E');
   "    "    ('D', 'F');
   "    "    ('E', 'F');

// Depth First Search Recursively
graph.depthFirstRecursive('A');

// Depth First Search Iteratively
graph.depthFirstIteratively('A');

// Breadth First Search
graph.breadthFirstSearch('A');
```

Output:

```
['A', 'B', 'D', 'E', 'C', 'F']
['A', 'C', 'E', 'F', 'D', 'B']
['A', 'C', 'B', 'E', 'D', 'F']
```

Dijkstra's Algorithm

Dijkstra's Algorithm is a shortest Path Algorithm.

When working with weighted ~~ad e~~ and directed/ undirected graphs, we very commonly want to know how to get from one vertex to another! What's the fastest way to get from point A to point B?

Dijkstra's algorithm is a greedy algorithm not Dynamic programming algorithms.

* The Approach :-

1. Everytime we look to visit a new node, we pick the node with the smallest known distance to visit first.

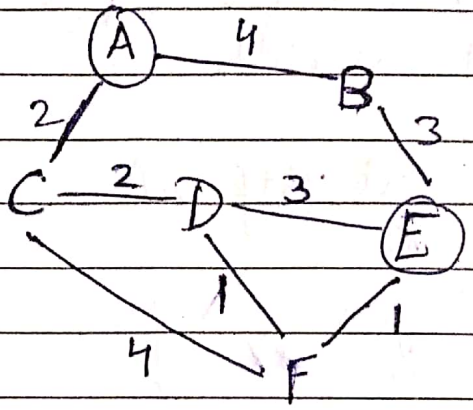
2. Once we've moved to the node we're going to visit, we look at each of its neighbors.

3. For each neighboring node, we calculate the distance by summing the total edges that lead to the node we're checking from the starting node.

4. If the new total distance to a node is less than the previous total, we store the new shorter distance for that node.

* Working of Dijkstra's Algorithm:-

Find the shortest path from A to E



Note:- Here, we're going in alphabetical order to start. Then, we always pick the node that has the smallest known distance from A and we visit that first.

Step 1:- ~~The~~ Store the shortest distance from A for each vertex (Initilization step).

Vertex	Shortest Dist. from A
A	0
B	Infinity
C	Infinity
D	Infinity
E	Infinity
F	Infinity

Visited:
[]

Previous:
{
A: null,
B: null,
C: null,
D: null,
E: null,
F: null
}

Since, We don't know the shortest distance from A to each vertex. But we know from A to A i.e. 0. So, assume 'Infinity' for other vertices.

Step2:- Every time we look to visit a new node, we pick the node with the smallest known distance to visit first.

0 < Infinity, so we pick the A and make it visited.

vertex	Shortest Dist from A
A	0
B	Infinity
C	"
D	"
E	"
F	"

Visited:
[A]

Previous:
{
A: null
:
F: null
}

Step3:- From A we have two choices: A to B or A to C, we're going to go alphabetically, doesn't matter.
So, A to B we're going

Update:
4 < Infinity ⇒ new Short Dist. from A to B
Since, we got to B through A. So, update previous. B: A

Vertex	Shortest Dist from A
A	0
B	~~Infinity~~ 4
C	Infinity
D	"
E	"
F	"

Visited:
[A]

Previous:
{ A: null,
B: A,
C: null,
:
F: "
}

Why we need Previous Data Structure?
Later on something like F if we update the shortest distance to F, we want to remember, did we get there from D or did we get there from C? There's multiple options, so this structure helps us piece together where we came from.

similarly, from A → C

Vertex	Shortest dist from A	Visited:
A	0	[A]
B	4	
C	~~Infinity~~, 2	Previous:
D	Infinity	{ A: null,
E	"	B: A,
F	"	C: A,
		D: null,
		F: "",

(C was from A)

Step 4: Now we are to repeat the process from Step 1. We pick the smallest known value, the shortest distance from A that we haven't visited i.e. C. Remember, we've been A (present in visited array). So, we're not going to pick A again.

Vertex	Shortest Dist from A
~~A~~	0
B	4
C	2
D	Infinity
E	Infinity
F	Infinity

~~Visited~~: Visited: [A, C]

Previous: { A: null
B: A,
C: A
D: null,
F: "",

Step 5: Move to the unvisited neighbor of C:
Since, we're moving Alphabatically (doesn't matter)

Let's move to C → D, ShortestDist: A → C → D

Vertex	Shortest Dist from A
A	0
B	4
C	2
D	~~Infinity~~, 4
E	Infinity
F	Infinity

Visited:
[A, C]
Previous:
{ A: null
B: A,
C: A,
D: C,
E: null,
F: null
}

(reach D from C)

Similarly for its another neighbor i.e. F

Vertex	Shortest Dist from A
A	0
B	4
C	2
D	4
E	Infinity
F	~~Infinity~~, 6

Visited:
[A, C]
Previous:
{ A: null,
B: A,
C: A,
D: C,
E: null
F: C
}

Now, we've visited all the neighbors of C.
So, again now its time to pick the unvisited vertex having smallest distance from A.
Unvisited means: Vertex that is not present in Visited array.

Step 6:

Vertex	Shortest Dist from A
~~A~~	0
B	4
~~C~~	2
D	4
E	Infinity
F	6

→ make it visited

Visited:
[A, C, B]

Previous:
{ A: null
B: A,
C: A
D: C,
E: null
F: C
}

Neighbor of B:
B → E.

Shortest Distance from A:
A → B → E: 4 + 3 → 7

Vertex	Shtst Dist from A
A	0
B	4
C	2
D	4
E	~~Infinity~~, 7
F	6

Visited:
[A, C, B]

Previous:
{ A: null
B: A,
C: A;
D: C
E: B
} F: C

Step 7:

Vertex	
~~A~~	0
~~B~~	4
~~C~~	2
D	4
E	7
F	6

Visited:
[A, C, B, D]

Previous:
{ A: null,
B: A,
C: A,
D: C,
E: B,
F: C
}

D's neighbor i.e. E

Since, shortest Distance from D to E is 7. And the previous value of E is 7 as well. So, it is same. Hence no need to update the value.

Another D's neighbor i.e F
shortest Dist from D to F is :
2 + 2 + 1 = 5 < previous value of F. Hence, Update.
Also, update the previous vertex of F.

Vertex	"	Visited:
A	0	[A, C, B, D]
B	4	Previous:
C	2	{ A : null,
D	4	B : A,
E	7	C : A,
F	~~6~~ 5	D : C,
		E : B,
		F : D }

Step-8: Make Vertex F visited

Vertex	"	Visited:
~~A~~	0	[A, C, B, D, F]
~~B~~	4	Previous:
~~C~~	2	{ A : null,
~~D~~	4	B : A,
E	7	C : A
F	5	D : C,
		E : B,
		F : D }

Neighbor of F is E

Shortest Dist from F → E i.e.
A → C → D → F → E ⇒ 2 + 2 + 1 + 1 = 6

6 < Prev. value of E i.e. 7

So, Update the value of the previous vertex of E

Vertex	Shortest Dist from A
A	0
B	4
C	2
D	4
E	~~7~~, 6
F	5

Visited:
[A, C, B, D, F]

Previous:
{ A: null,
B: A
C: A
D: C
E: F
F: D

∴ Shortest Path would be for A to E:

[A → C → D → F → E]

Code Implementation : with Binary Heap (an Optimized) Priority Queue

```
// Weighted Graph
class WeightedGraph {
    constructor() {
        this.adjacencyList = {};
    }

// Add Vertex
addVertex(vertex) {
    if(!this.adjacencyList[vertex])
        this.adjacencyList[vertex] = [];
}
// Add Edge between the two vertices along with weight
addEdge(vertex1, vertex2, weight) {
    if(this.adjacencyList[vertex1])
    this.adjacencyList[vertex1].push({node:vertex2, weight:
                                      weight});
    if(this.adjacencyList[vertex2])
    this.adjacencyList[vertex2].push({node:vertex1, weight:
                                      weight});
}

// Dijkstra Algorithm
Dijkstra(start, finish) {
    const nodes = new PriorityQueue();

//store the shortest distances from Vertex 'A' or
starting Vertex to each vertex
const distances = {};
```

```
//store the vertex that tell from where it came from for
each corresponding vertex
  const previous = {};

//store the vertex having smallest distance from
starting vertex, by dequeue from the PriorityQueue
  let smallest;

  //build up initial state
for(let vertex in this.adjacencyList){

  if(start = vertex){
    distances[vertex] = 0;
//enqueue d vertex in PriorityQueue with distance
as priority.
    nodes.enqueue(vertex, 0);
  else {
    distances[vertex] = Infinity;
    nodes.enqueue(vertex, Infinity);
  }
  previous[vertex] = null;
}
//as long as there is something to visit
while(nodes.value.length){
  //get the value of smallest node (smallest node
  will whatever the actual node we're visiting is)
  smallest = nodes.dequeue.val;

  if(smallest === finish){
  //WE'RE DONE
```

```
// BUILD UP PATH TO RETURN AT END
let path = [];

while (previous[smallest]) {
    path.push(smallest);
    // updating smallest till starting vertex (having value = null);
    smallest = previous[smallest];
}
return path.concat(smallest).reverse();
}

if (smallest // distance[smallest] !== Infinity) {
    for (let neighbor in this.adjacencyList[smallest]) {
        let nextNode = this.adjacencyList[smallest][neighbor];

        // Calculate new distances to neighboring node from the starting vertex
        let newDistance = distances[smallest] + nextNode.weight;
        nextNeighbor = nextNode.node;

        // if the newDistance is Minimum then only, update the
        distances & previous of nextNeighbor and enqueue
        with that new Minimum distance as new Priority.
        if (newDistance < distances[nextNeighbor]) {
            // updating new smallest distance to neighbor
            distances[nextNeighbor] = newDistance;

            // Updating previous - How we got to neighbor
            distances[nextNeighbor] = smallest;
            ~~newDistance;~~

            ~~// updating pq~~
```

```
//enque in priorityqueue with new priority
(Minimum newDistance)
nodes.enqueue(nextNeighbor, newDistance);
 }
}
 }
  }
 }
}

//An optimized Priority Queue
class PriorityQueue{
   constructor(){
   this.value=[];
  }
//Adding to create MinBinaryHeap
bubbleUp(){
   let idx = this.value.length-1,
     element = this.value[idx];

  while(idx>0){
     let parentIdx = Math.floor((idx-1)/2),
      parent  = this.value[parentIdx];
  if(element.priority >= parent.priority) break;

    this.value[parentIdx]= element;
     this.value[idx]=parent;
      idx = parentIdx;
   }
 }
```

```
// Removing from MinBinary Heap
dequeue() {
    const min = this.value[0];
        end = this.value.pop();
    if (this.value.length > 0) {
        this.value[0] = end;
        this.sinkDown();
    }
    return min;
}

// Bubbling Down or say sinkDown to adjust as Min Binary heap.
sinkDown() {
    let idx = 0,
        element = this.value[0]
        length = this.value.length;

    while (true) {
        let leftChildIdx = 2*idx + 1,
            rightChildIdx = 2*idx + 2,
            leftChild, RightChild, swap = null;

        if (leftChildIdx < length) {
            leftChild = this.value[leftChildIdx];

            if (element.priority > leftChild.priority)
                swap = leftChildIdx;
        }

        if (rightChildIdx < length) {
```

```
RightChild = this.value[rightChildIdx];
if(swap === null && RightChild.priority <
  element.priority || swap !== null &&
    RightChild.priority < leftChild.priority){
      swap = rightChildIdx;
  }
}

if(swap === null) break;

  this.value[idx] = this.value[swap];
  this.value[swap] = element;
    idx = swap;
  }
}
}

class Node{
  constructor(val, priority){
    this.val = val;
    this.priority = priority;
  }
}

let graph = new WeightedGraph();
//Adding Vertex
graph.addVertex('A');
      .addVertex('B');
      .addVertex('C');
      .addVertex('D');
      .addVertex('E');
      .addVertex('F');
```

//Adding Edge between vertices along with weight
graph.addEdge('A', 'B', 4);
graph.addEdge('B', 'E', 3);
graph.addEdge('A', 'C', 2);
graph.addEdge('C', 'D', 2);
graph.addEdge('C', 'F', 4);
graph.addEdge('D', 'F', 1);
graph.addEdge('D', 'E', 3);
graph.addEdge('E', 'F', 1);

//INPUT: Find the shortest path from vertex 'A' to 'E'
graph.Dijkstra('A', 'E');

//OUTPUT:
['A', 'C', 'D', 'F', 'E']