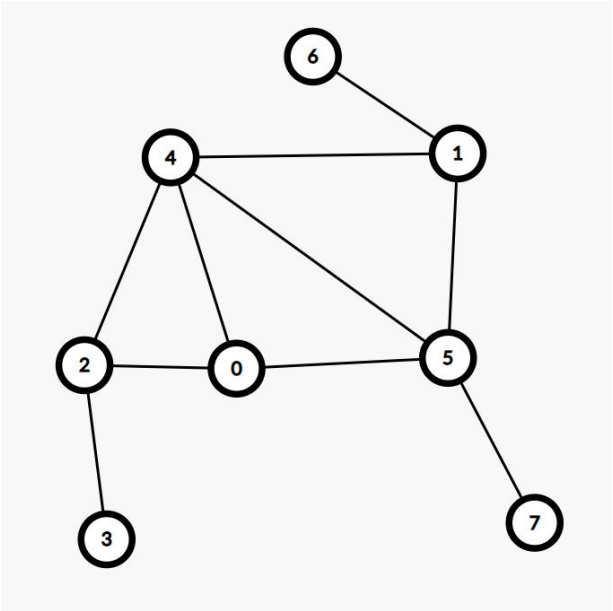
Graph Theory Introduction

Graph representation, BFS and Dijkstra’s

What is a “Graph”?

A graph consists of a bunch of nodes/vertices connected by edges. An edge connects two nodes. Each node and each edge will represent something from the problem. For example, each node might represent a house and each edge might represent a road connecting two houses.



Different types of graphs

Graphs are “directed” or “undirected”. In a directed graph, each edge has a direction. This might represent that you can only go along a road in one direction. In a undirected graph, edges don’t have a direction; you can traverse them in both ways.

Graphs can also be “weighted”. This means that each edge has some sort of additional number associated with it, sometimes called its weight. For example, this might represent how much time it takes to go along a road.

Example problem

https://dmoj.ca/problem/vmss7wc16c3p2

“There are N houses in Shahir's neighbourhood and M roads that connect those N houses. **Each road connects two houses.** All roads can be traveled down both directions. Each house is distinctly numbered and identified by a number in the range 1…N.”

The problem then asks you to check whether it is possible to travel between two specified nodes, A and B, using only the roads given.

This problem can be modeled as an undirected unweighted graph.

Graph Representation: Adjacency List

The most common way to represent a graph is with an adjacency list. In this representation, each node has a list associated with it that represents all the other nodes you can directly travel to from this node.

```cpp
// Maximum number of nodes, given in the problem statement
const int MN = 2005;
int n, m;
vector<int> adjList[MN];

int main(){
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        // Decrease a and b by 1 since input is 1-indexed but I
prefer 0-indexed
        a--; b--;
        // Add an edge both ways since the graph is undirected
        adjList[a].push_back(b);
        adjList[b].push_back(a);
    }
}
```

```python
n, m = map(int, input().split())
adjList = [[] for i in range(n)]

for i in range(m):
    a, b = map(int, input().split())
    # 1 indexed -> 0 indexed
    a -= 1; b -= 1
    adjList[a].append(b)
    adjList[b].append(a)
```

Graph Representation: Adjacency Matrix

Another way of representing graphs is using an adjacency matrix, which is a 2D boolean array where adjMat[i][j] is a boolean representing whether there is an edge from i to j or not. This representation is much less commonly used as it takes O(N^2) memory where N is the number of nodes. It is only viable for very dense graphs (few nodes and many edges), in which case it might be better than an adjacency list because each edge only takes one bit (or one byte, depending on implementation) instead of using an entire 32-bit integer.

```cpp
// Maximum number of nodes, given in the problem statement
const int MN = 2005;
int n, m;
bool adjMat[MN][MN];

int main(){
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        // Decrease a and b by one since input is 1-indexed but I prefer
0-indexed
        a--; b--;
        // Add an edge both ways since the graph is undirected
        adjMat[a][b] = true;
        adjMat[b][a] = true;
    }
}
```

```python
n, m = map(int, input().split())
adjMat = [[False * n] for i in range(n)]

for i in range(m):
    a, b = map(int, input().split())
    # 1 indexed -> 0 indexed
    a -= 1; b -= 1
    adjMat[a][b] = True
    adjMat[b][a] = True
```

Shortest Path Problem

The next two algorithms we will be looking at will solve the following problem: given a graph and two vertices, find the shortest path between them.

Breadth First Search

Breadth First Search

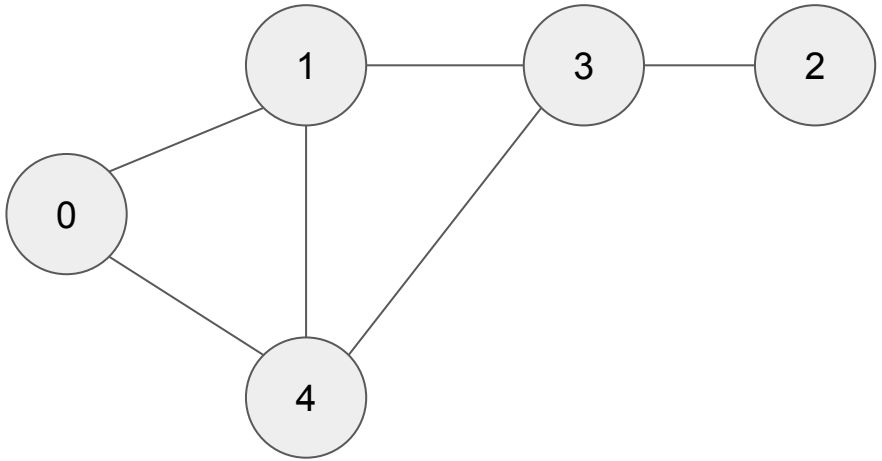
Breadth First Search (BFS) is an algorithm that works only on unweighted graphs. It can be used to find the distance of every node from a specified node, u, in O(M) time, where M is the number of edges in the graph. Since it is quite a simple algorithm, it is also often used simply to check whether there exists a path between two nodes.

The algorithm works by first finding all nodes a distance of 0 from u, then all nodes a distance of 1 from u, then all nodes a distance of 2 from u…

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

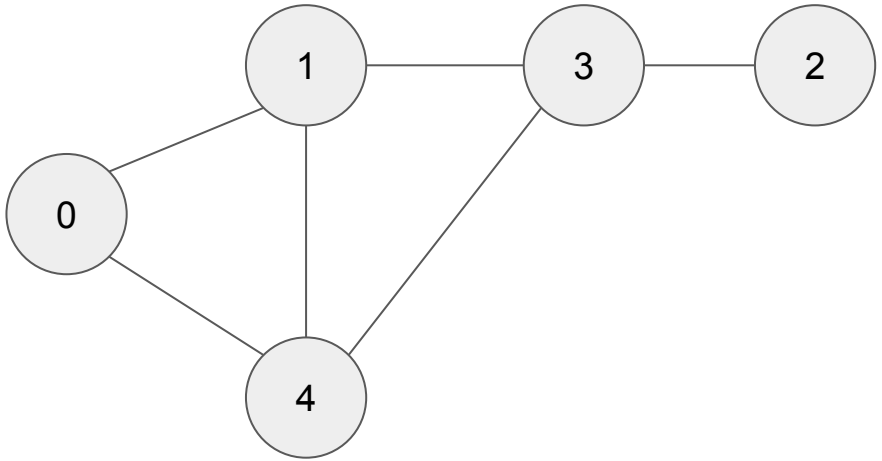



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: false      {}
[1]: {0, 3, 4}    [1]: false
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: false
[4]: {0, 1, 3}    [4]: false
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

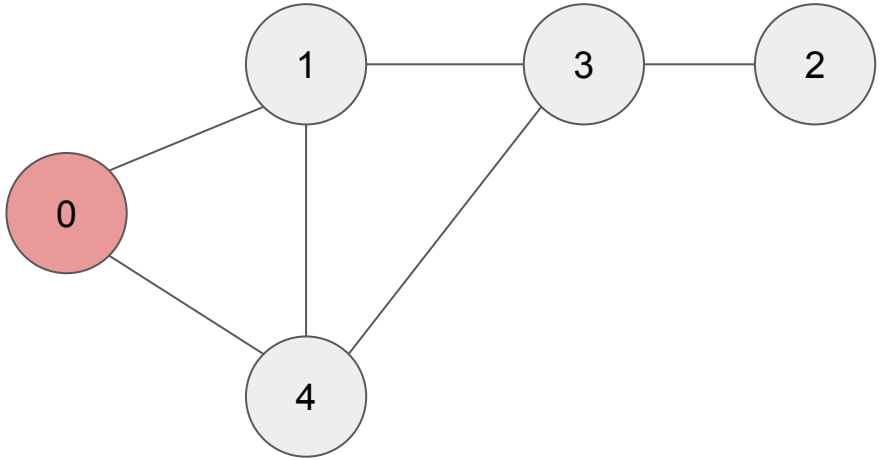



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {0}
[1]: {0, 3, 4}    [1]: false
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: false
[4]: {0, 1, 3}    [4]: false
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

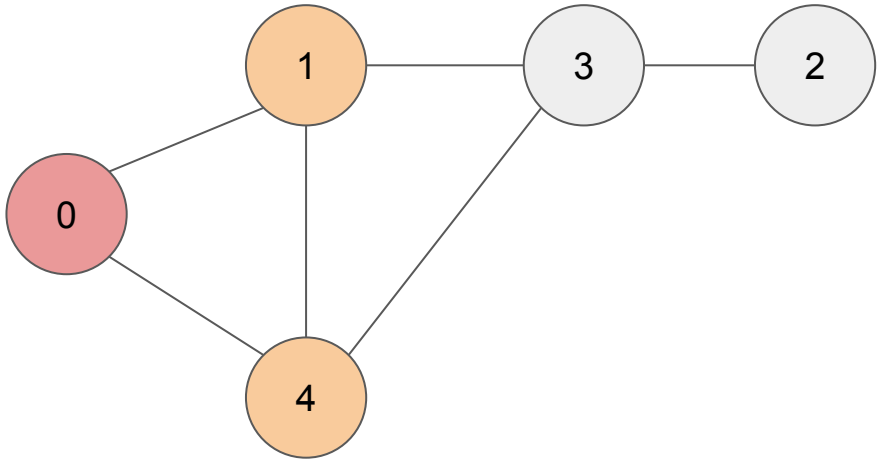



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {}
[1]: {0, 3, 4}    [1]: false
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: false
[4]: {0, 1, 3}    [4]: false
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

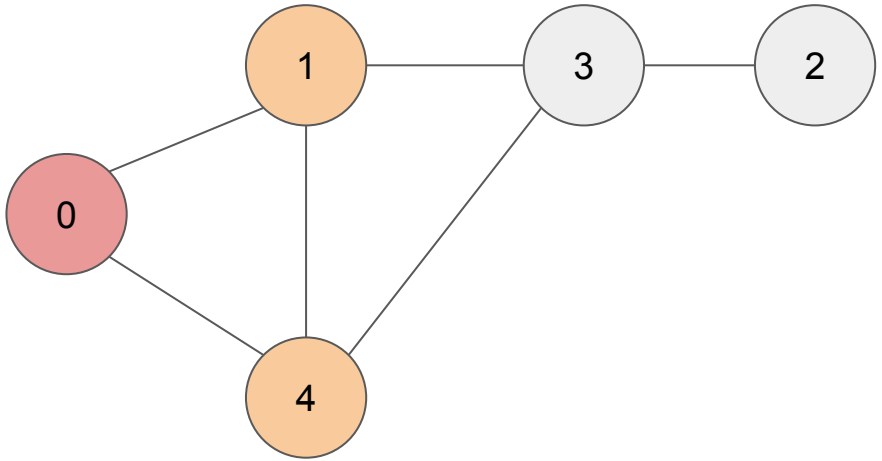



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {}
[1]: {0, 3, 4}    [1]: false
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: false
[4]: {0, 1, 3}    [4]: false
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

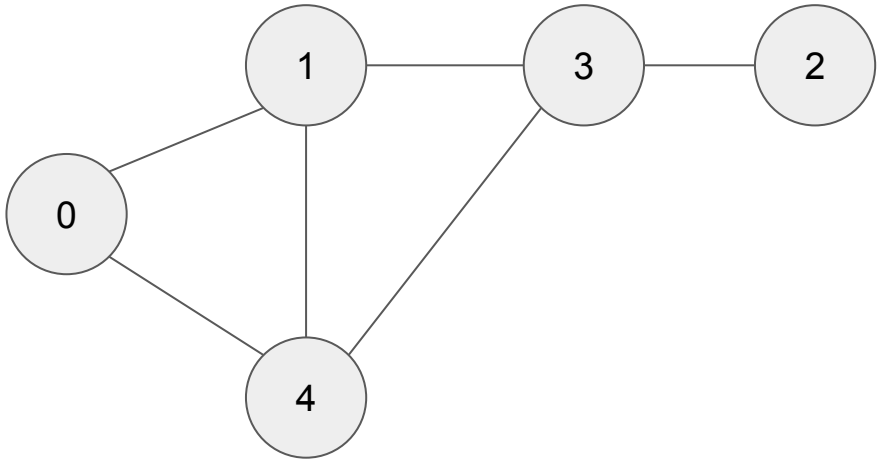



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {1, 4}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: false
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

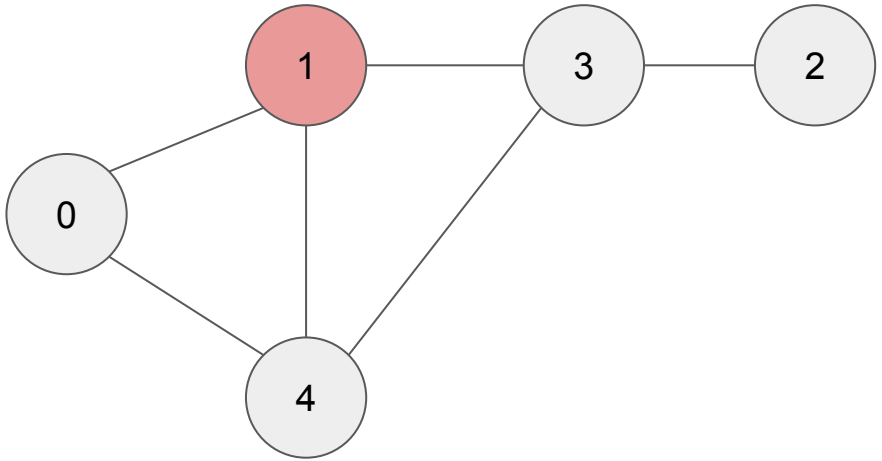



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {1, 4}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: false
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

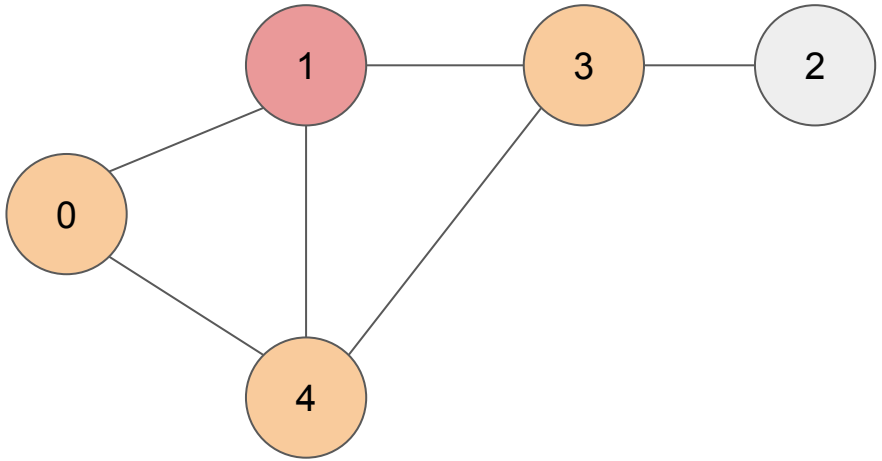



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {4}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: false
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

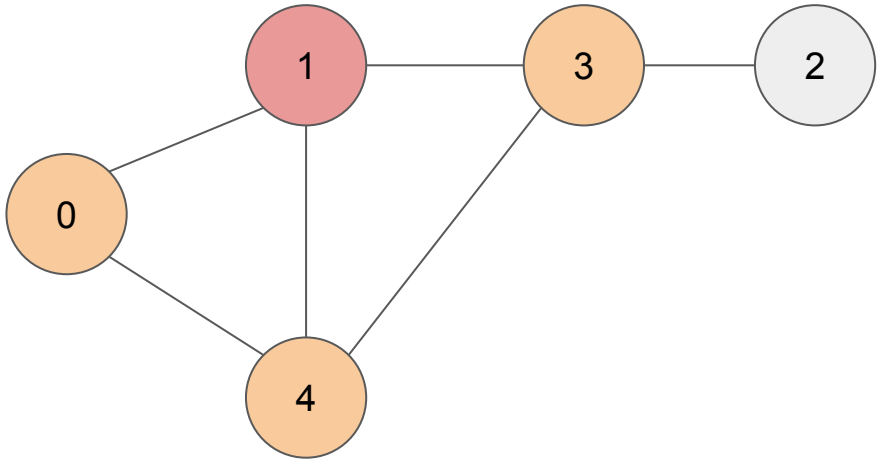



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {4}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: false
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

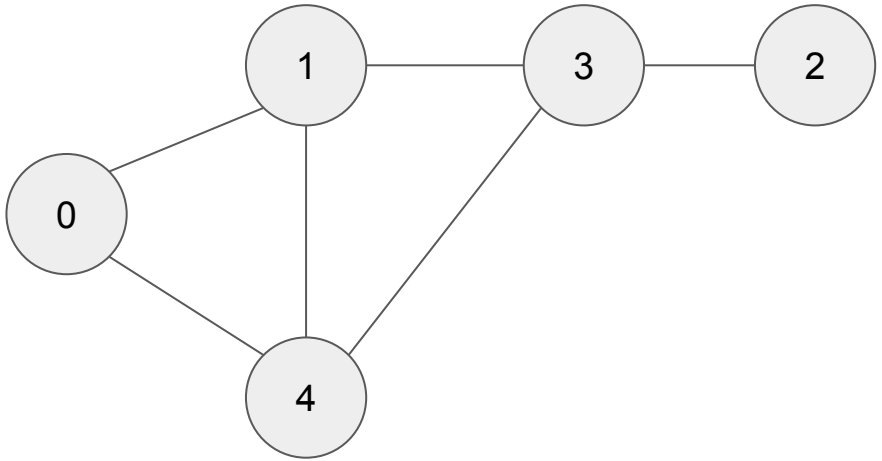



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {4, 3}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: true
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

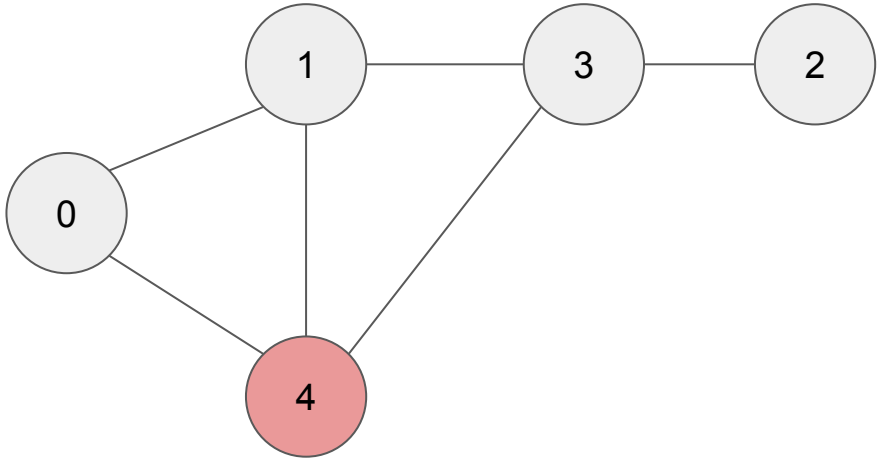



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {4, 3}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: true
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

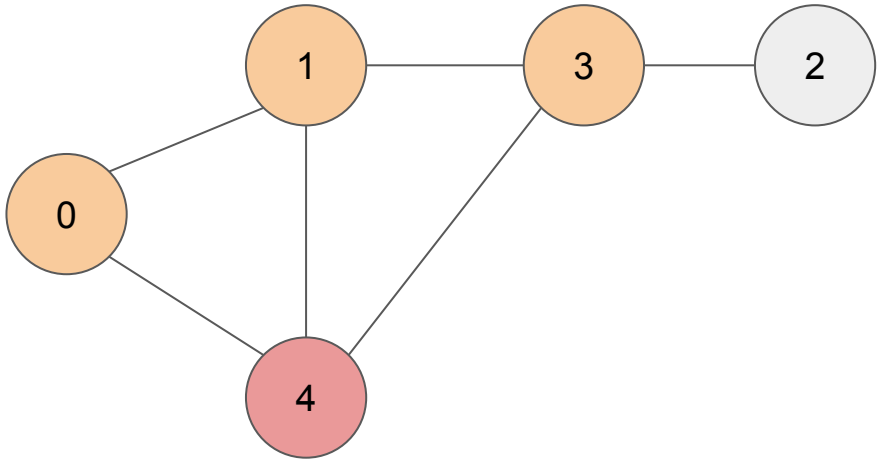



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {3}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: true
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

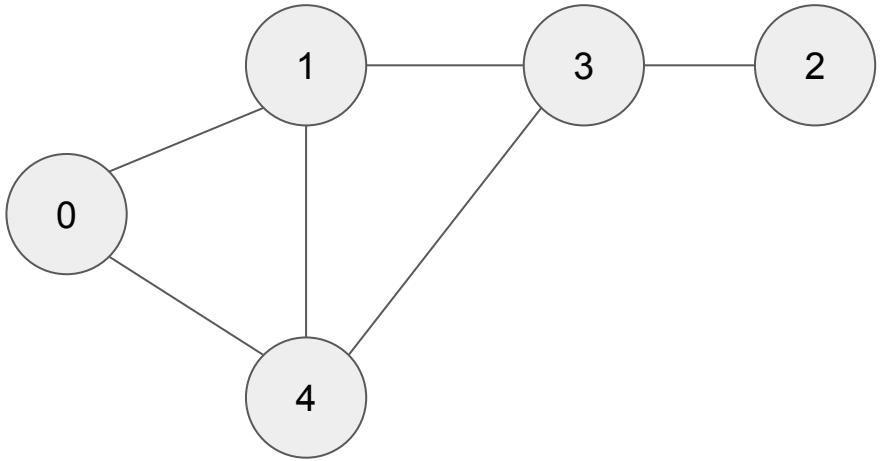



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {3}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: true
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

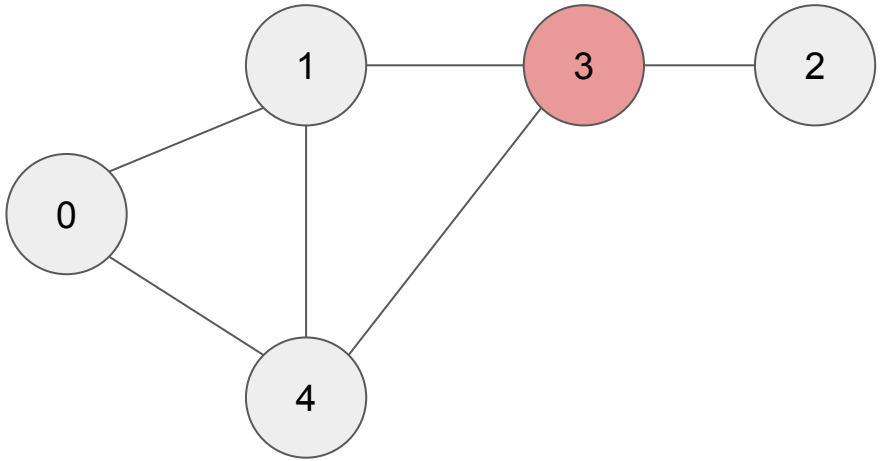



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {3}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: true
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

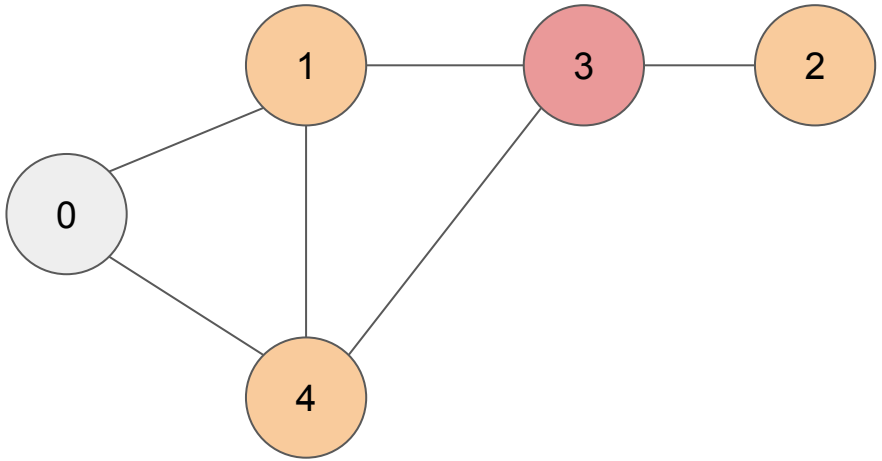



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: true
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

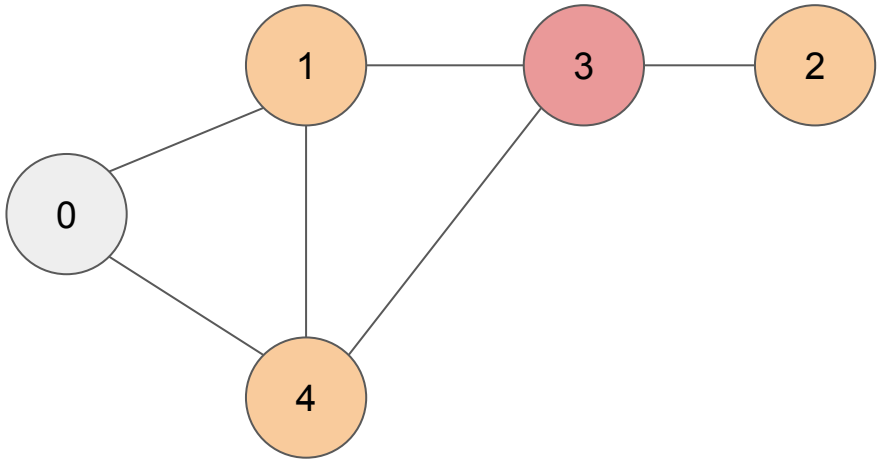



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: false
[3]: {1, 2, 4}    [3]: true
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

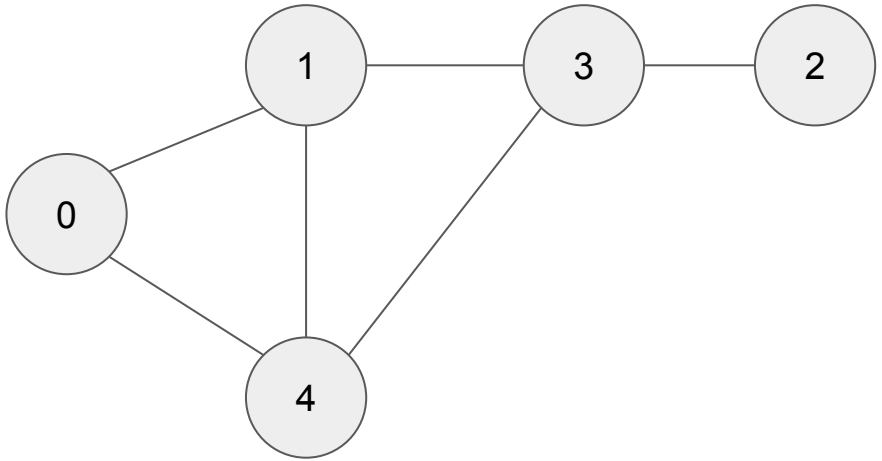



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {2}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: true
[3]: {1, 2, 4}    [3]: true
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

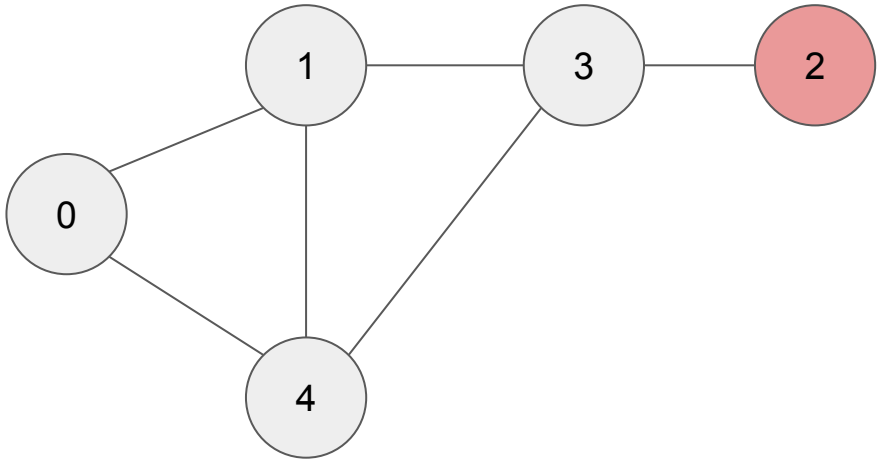



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {2}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: true
[3]: {1, 2, 4}    [3]: true
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

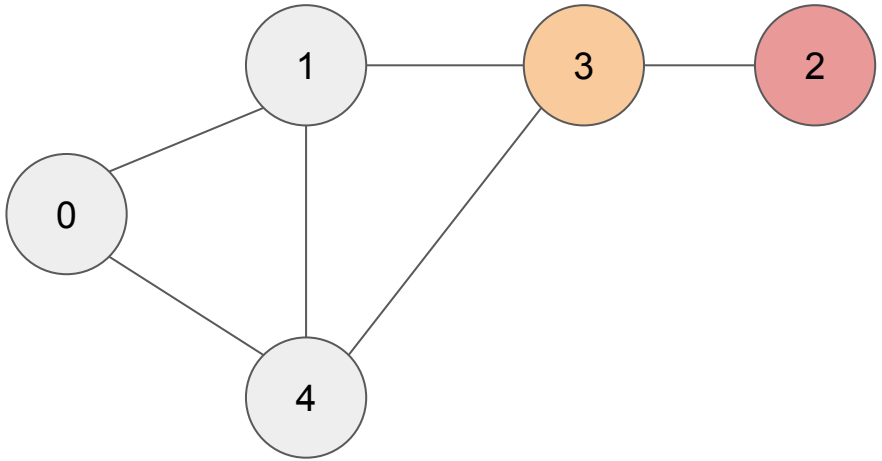



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: true
[3]: {1, 2, 4}    [3]: true
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.

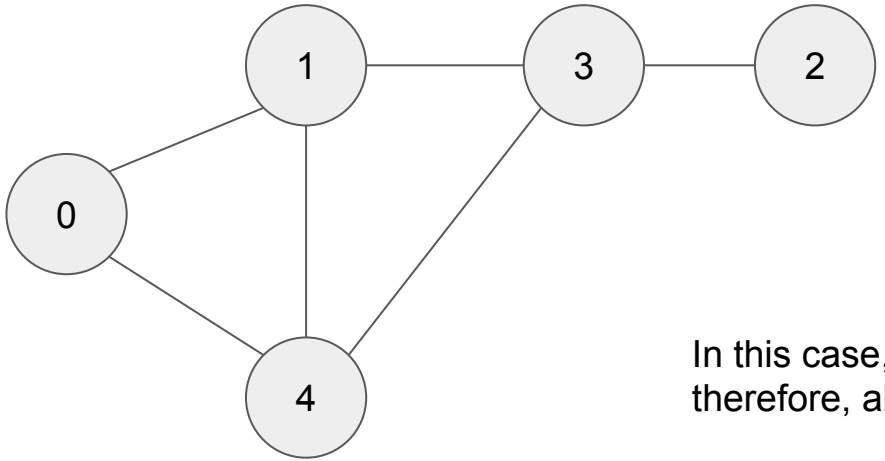



```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: true
[3]: {1, 2, 4}    [3]: true
[4]: {0, 1, 3}    [4]: true
```

BFS: Connectivity Check

Here is how BFS works if you just want to find all the nodes reachable from u.

You keep a first-in-first-out queue and a boolean array. The boolean array will represent whether each node has already been visited by the algorithm. We go through every node in the queue, and iterate through all the nodes adjacent to it. If we come across a node that has not yet been visited, we add it to the end of the queue and mark it as visited in our boolean array. We start by adding the starting vertex to the queue and setting its visited flag.




```
adjList:          visited:        queue:
[0]: {1, 4}       [0]: true       {}
[1]: {0, 3, 4}    [1]: true
[2]: {3}          [2]: true
[3]: {1, 2, 4}    [3]: true
[4]: {0, 1, 3}    [4]: true
```

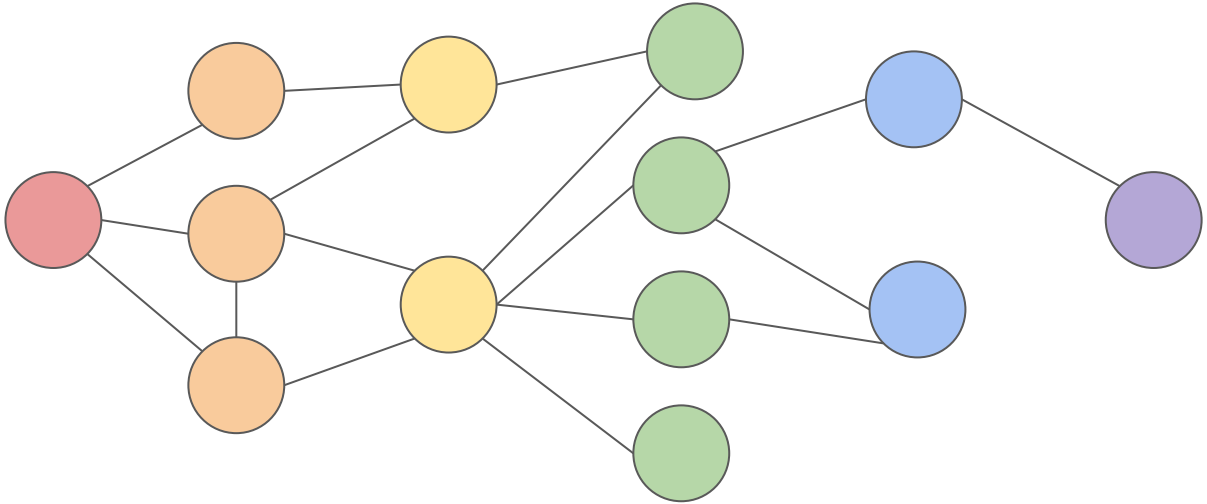
In this case, all the nodes are reachable from node 0, therefore, all the values in the visited array are true.

BFS: Complexity

For a graph with N nodes and M edges, BFS takes O(M) time to run. This is because each vertex is visited at most once, therefore each edge is checked at most twice (once from each side).

BFS: Shortest Path

We can easily modify BFS to find the shortest path from a single source to every node instead of only checking whether they are reachable. We do so by keeping a distance array, each of whose elements represents the minimum distance required to go from the source to that node. Whenever we visit a new node, we set its distance equal to one greater than the node from which it was visited. This works because BFS already visits the nodes in order of non-decreasing distance (because of the first-in-first-out queue that we use).



BFS: Implementation

```cpp
const int MN = 2005;
int n, m;
vector<int> adjList[MN];
bool vis[MN];
int dist[MN];

void bfs(int src){
    vis[src] = true;
    queue<int> qu;
    qu.push(src);

    while(!qu.empty()){
        int node1 = qu.front();
        qu.pop();
        for(int node2 : adjList[node1]){
            if(vis[node2]) continue;
            vis[node2] = true;
            dist[node2] = dist[node1] + 1;
            qu.push(node2);
        }
    }
}
```

```python
from collections import deque

n, m = map(int, input().split())
adjList = [[] for i in range(n)]
vis = [False] * n
dist = [0] * n

def bfs(src):
    vis[src] = True
    qu = deque()
    qu.append(src)
    while qu:
        node1 = qu.popleft()
        for node2 in adjList[node1]:
            if vis[node2]: continue
            vis[node2] = True
            dist[node2] = dist[node1] + 1
            qu.append(node2)
```

Dijkstra’s Algorithm

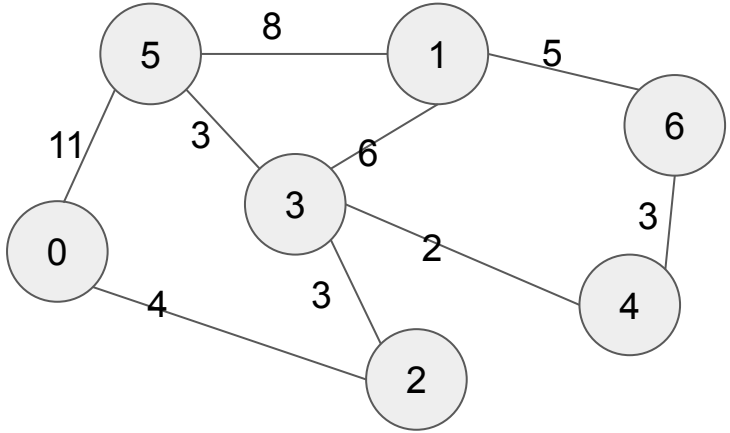
Dijkstra’s Algorithm

Problem:
You are given a **weighted** graph and want to find the shortest path between a pair of nodes.

We can modify BFS to work quite easily. The main difference between Dijkstra’s and BFS is that Dijkstra’s uses a priority queue instead of a normal queue. A priority queue is a queue where the elements are sorted by their value and the first element is always the one with the least value. With Dijkstra’s, we want to sort the elements by distance, so our queue will store elements that are pairs/tuples, with the first value being the distance of the node and the second value being the actual index of the node. We do not need to worry about the implementation of a priority queue, since most languages have one in their standard library, all that is important to us is that inserting and removing an element from the priority queue takes O(logN) time (where N is the number of elements currently in the priority queue).

Dijkstra’s Algorithm

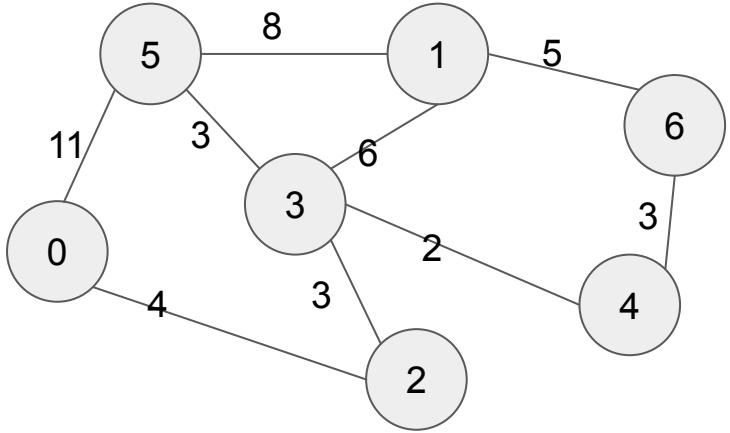
Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.




```
dist:          pq:
[0]: INF       {}
[1]: INF
[2]: INF
[3]: INF
[4]: INF
[5]: INF
[6]: INF
```

Dijkstra’s Algorithm

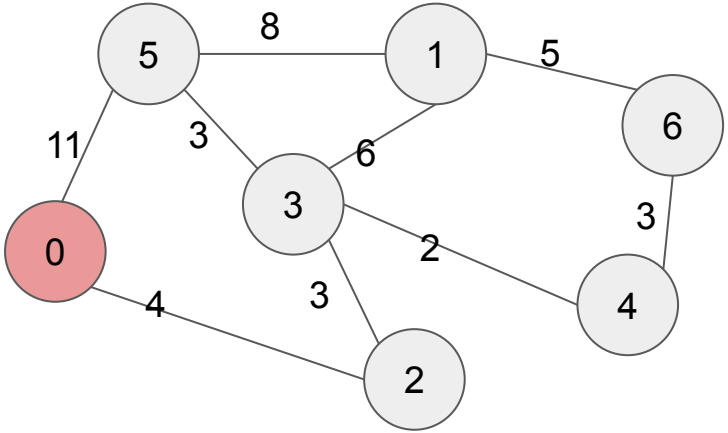
Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.




```
dist:          pq:
[0]: 0         {{0, 0}}
[1]: INF
[2]: INF
[3]: INF
[4]: INF
[5]: INF
[6]: INF
```

Dijkstra’s Algorithm

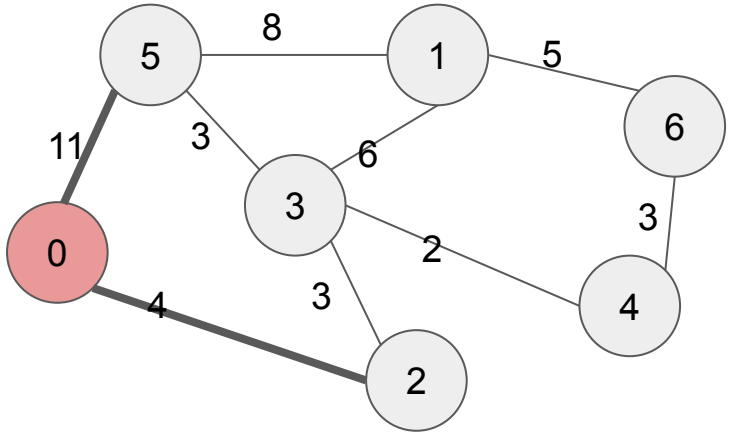
Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.




```
dist:          pq:
[0]: 0         {{0, 0}}
[1]: INF
[2]: INF
[3]: INF
[4]: INF
[5]: INF
[6]: INF
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

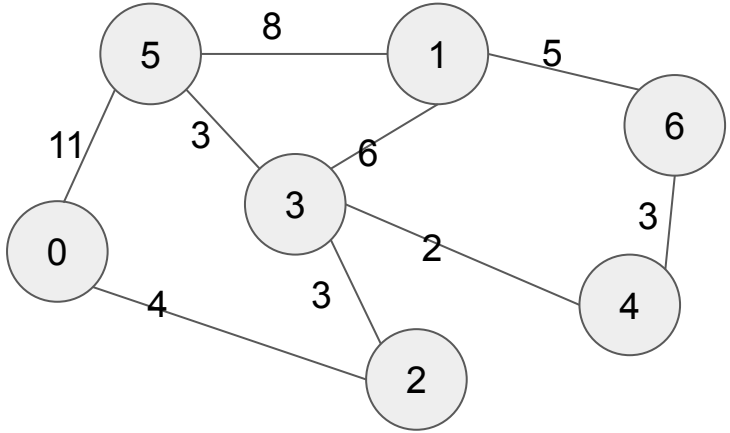



```
dist:
[0]: 0
[1]: INF
[2]: 4
[3]: INF
[4]: INF
[5]: 11
[6]: INF
```

```
pq:
{{0, 0}, {4, 2}, {11, 5}}
```

Dijkstra’s Algorithm

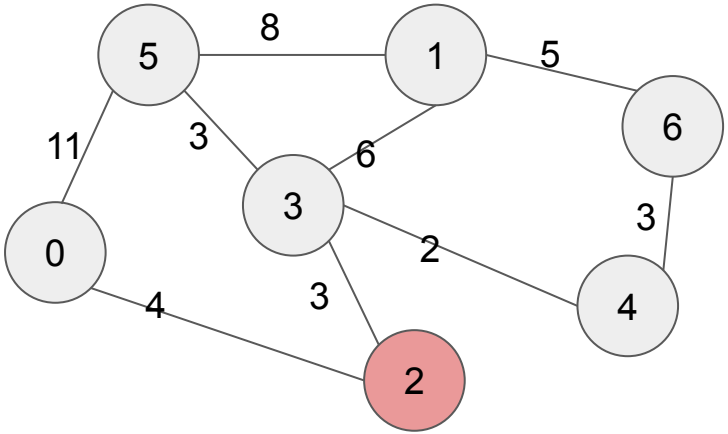
Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.




```
dist:          pq:
[0]: 0         {{4, 2}, {11, 5}}
[1]: INF
[2]: 4
[3]: INF
[4]: INF
[5]: 11
[6]: INF
```

Dijkstra’s Algorithm

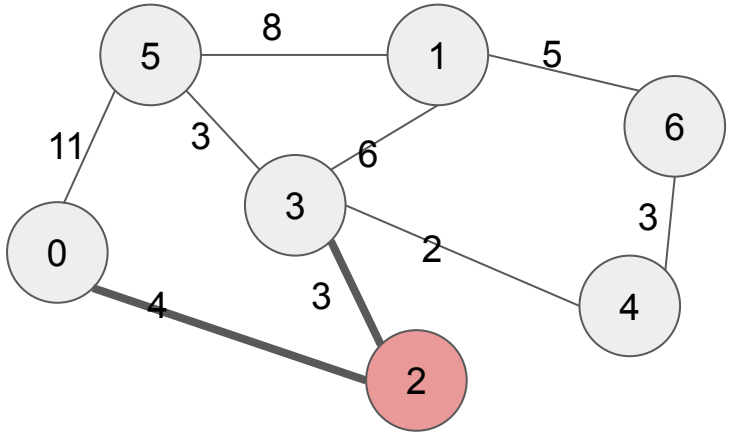
Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.




```
dist:          pq:
[0]: 0         {{4, 2}, {11, 5}}
[1]: INF
[2]: 4
[3]: INF
[4]: INF
[5]: 11
[6]: INF
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

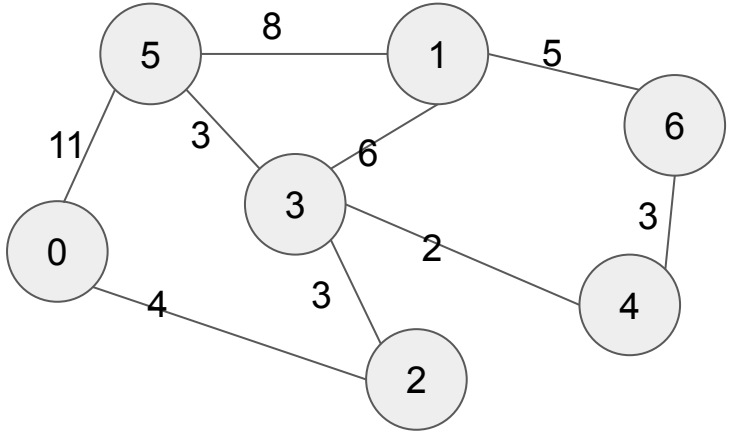



```
dist:
[0]: 0
[1]: INF
[2]: 4
[3]: 7
[4]: INF
[5]: 11
[6]: INF
```

```
pq:
{{4, 2}, {7, 3}, {11, 5}}
```

Dijkstra’s Algorithm

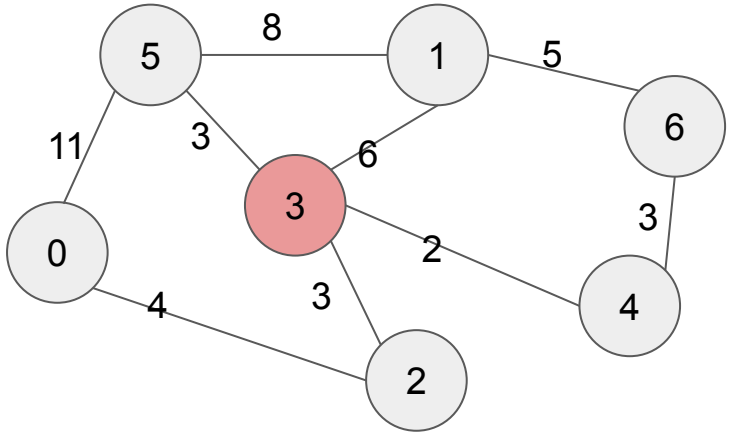
Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.




```
dist:          pq:
[0]: 0         {{7, 3}, {11, 5}}
[1]: INF
[2]: 4
[3]: 7
[4]: INF
[5]: 11
[6]: INF
```

Dijkstra’s Algorithm

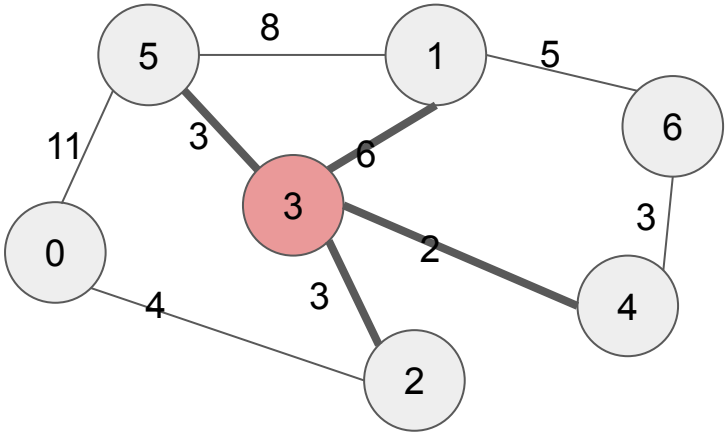
Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.




```
dist:          pq:
[0]: 0         {{7, 3}, {11, 5}}
[1]: INF
[2]: 4
[3]: 7
[4]: INF
[5]: 11
[6]: INF
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

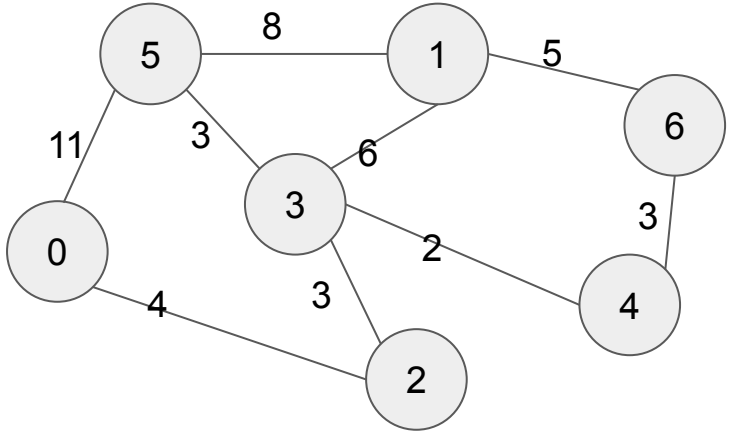



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: INF
```

```
pq:
{{7, 3}, {9, 4}, {10, 5}, {11, 5}, {13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

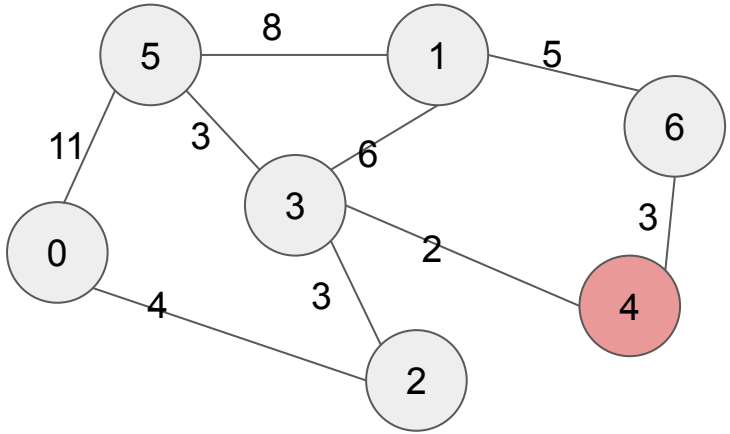



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: INF
```

```
pq:
{{9, 4}, {10, 5}, {11, 5}, {13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

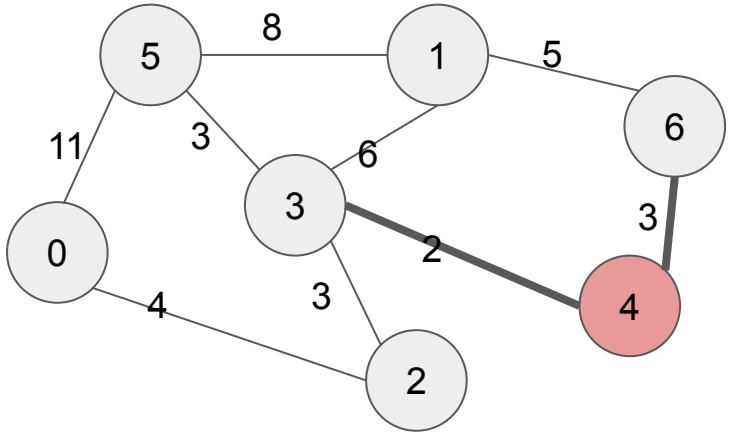



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: INF
```

```
pq:
{{9, 4}, {10, 5}, {11, 5}, {13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

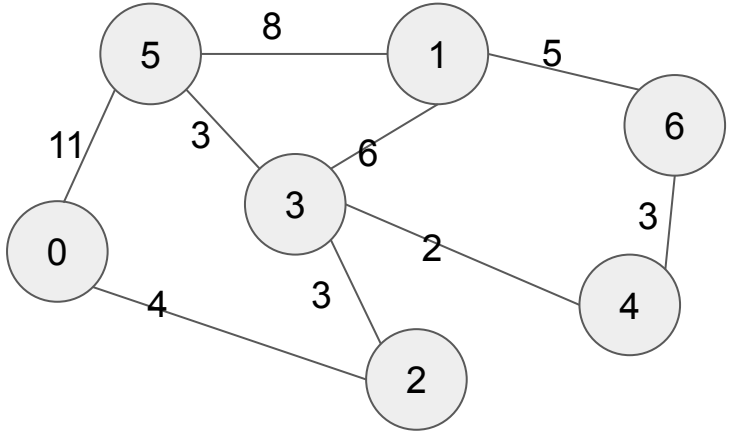



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: 12
```

```
pq:
{{9, 4}, {10, 5}, {11, 5}, {12, 6},
 {13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

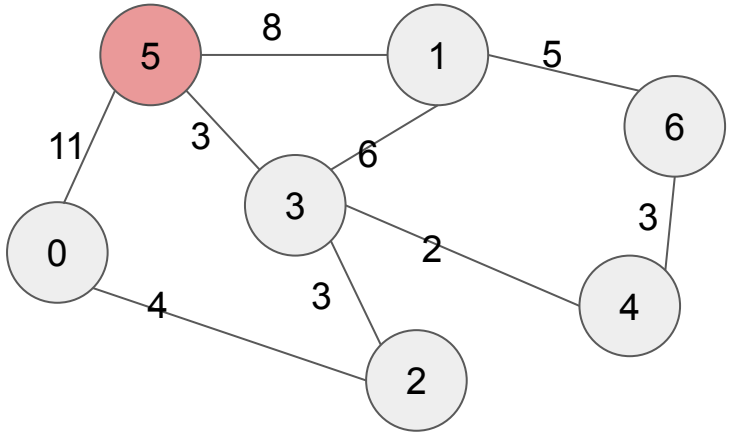



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: 12
```

```
pq:
{{10, 5}, {11, 5}, {12, 6}, {13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

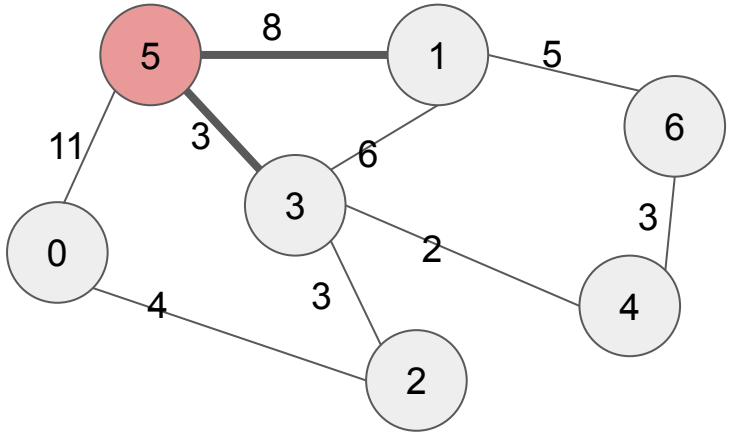



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: 12
```

```
pq:
{{10, 5}, {11, 5}, {12, 6}, {13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

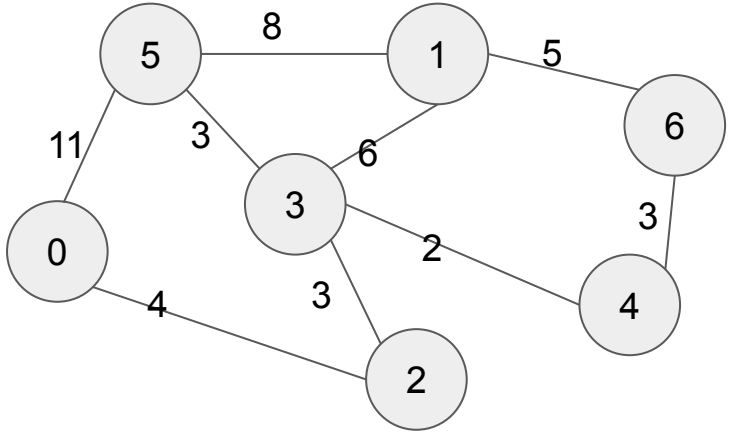



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: 12
```

```
pq:
{{10, 5}, {11, 5}, {12, 6}, {13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

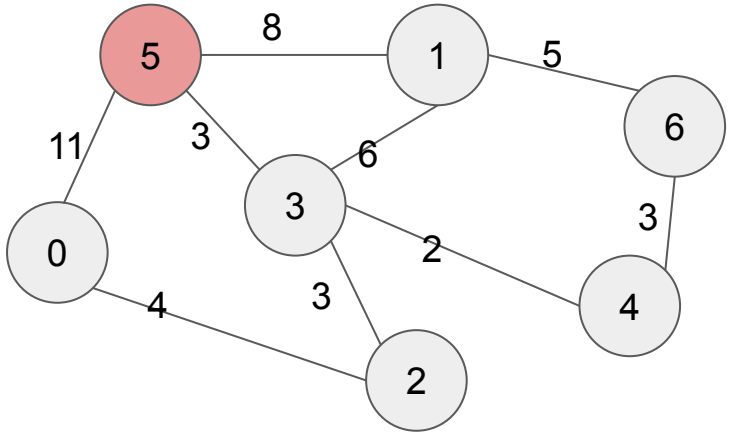



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: 12
```

```
pq:
{{11, 5}, {12, 6}, {13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.



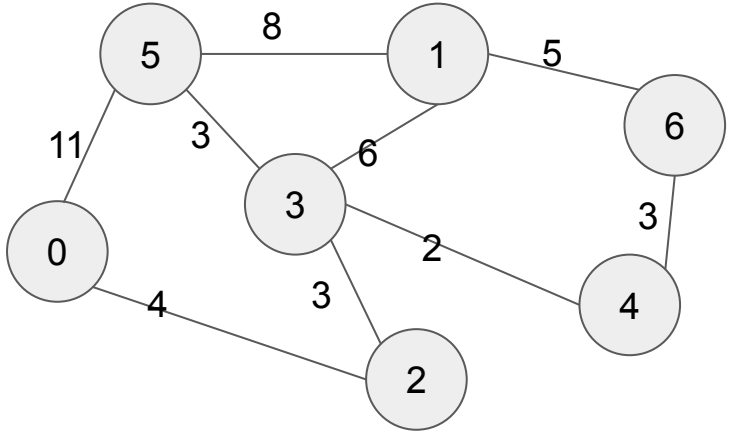

```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: 12
```

```
pq:
{{11, 5}, {12, 6}, {13, 1}}
```

Skip because 10 < 11

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

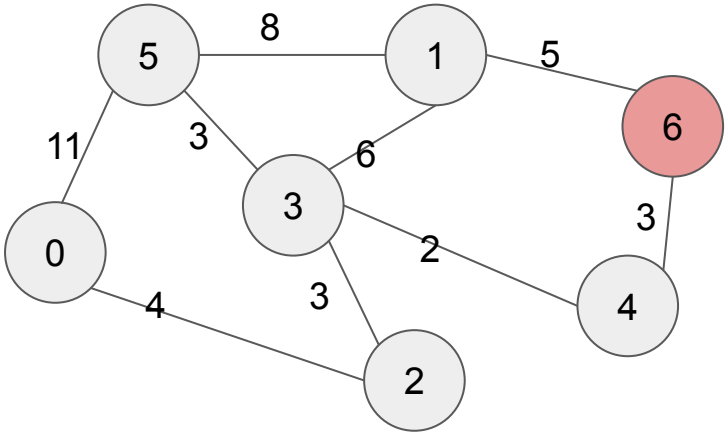



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: 12
```

```
pq:
{{12, 6}, {13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

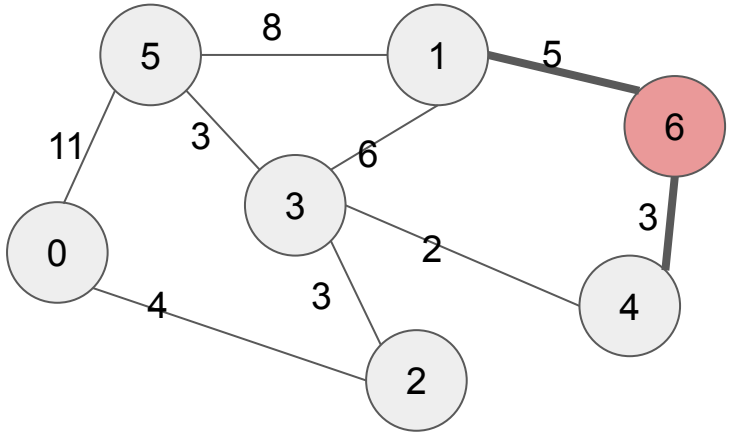



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: 12
```

```
pq:
{{12, 6}, {13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

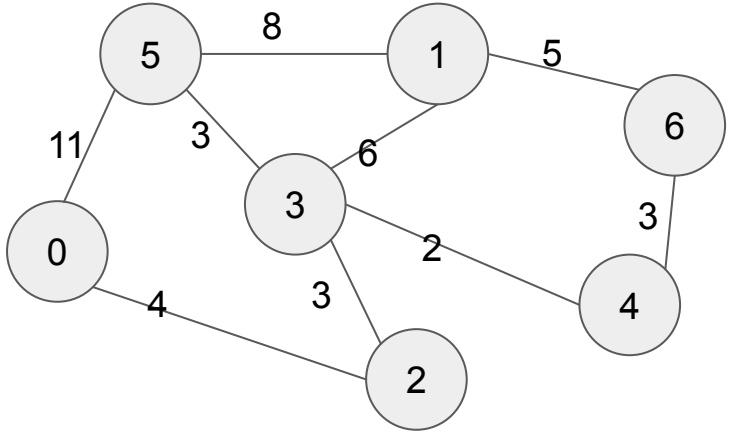



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: 12
```

```
pq:
{{12, 6}, {13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

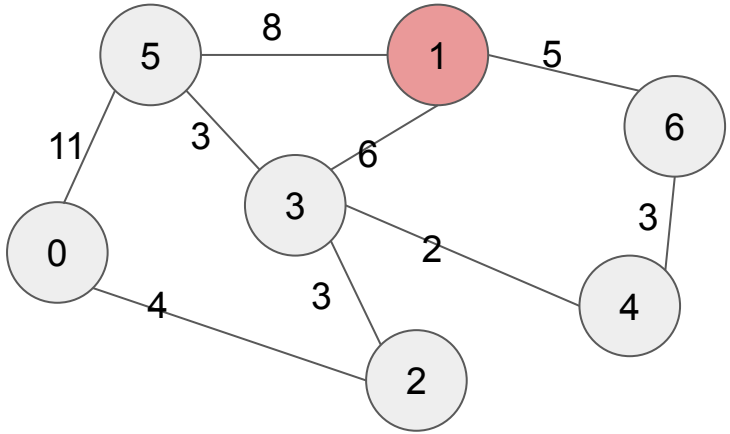



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: 12
```

```
pq:
{{13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

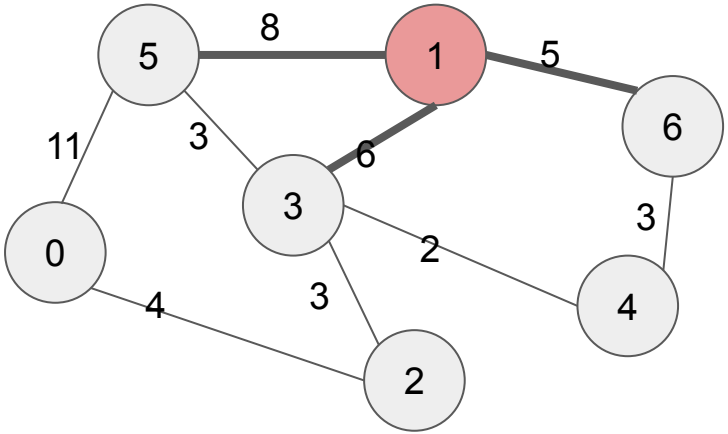



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: 12
```

```
pq:
{{13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.

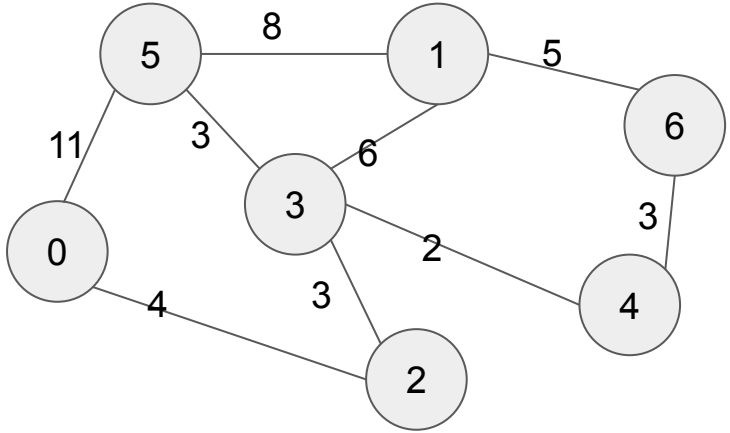



```
dist:
[0]: 0
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: 12
```

```
pq:
{{13, 1}}
```

Dijkstra’s Algorithm

Store a distance array like before, with all values initialized to infinity.
Store a priority queue of pairs, where the first value of each pair is the distance of the node and the second value is the actual node. This priority queue represents all the nodes we plan on checking soon.
For every node in the priority queue, we remove it, iterate through all of its neighbours, check whether going to that neighbour through this node would be better than the previously best path leading to that neighbour, and if it is, push it to the priority queue and update its distance value.




```
dist:          pq:
[0]: 0         {}
[1]: 13
[2]: 4
[3]: 7
[4]: 9
[5]: 10
[6]: 12
```

Dijkstra's Algorithm: Proof of Correctness and Complexity

Claim: Dijkstra's algorithm correctly finds the shortest path to every node, and when a node is popped from the priority queue, the path calculated to that node is already the shortest possible. Let dist[u] denote the distance our algorithm calculates from the source to node u.

Proof: Dijkstra's algorithm obviously never underestimates the length of the shortest path (it never calculates the length of the path to be shorter than it actually is). Now we only need to prove that the path calculated is already optimal when the node is popped from the priority queue. To do so, assume that the statement is not true. Meaning there is some node u such that u is the first node whose distance was incorrectly calculated (but the distance of all previous nodes was calculated correctly). This implies that there exists some node v such that dist[v] + edgeWeight(u, v) < dist[u]. However, this is impossible since, assuming all edge weights are non-negative, this implies that dist[v] < dist[u]. This is important because with the priority queue, we would check node v before node u.

Complexity: Dijkstra's algorithm runs in O(MlogM) time because we insert/remove at most M elements into/from the priority queue, and each of those operations takes MlogM time.

Priority Queues in C++/Python

C++

`priority_queue<T, cont, cmp>`
Stores elements of type T.
Uses an underlying container of cont.
Compares elements using cmp.

Greatest element is at the top.

Cont and cmp are optional and default to vector<T> and less<T>.

```
priority_queue<T> pq;
pq.push(val);
pq.top();
pq.pop();
```

Python

Python doesn't have an actual priority queue type only like a helper class called `heapq`.

```
heapq.heappush(arr, val)
```

```
heapq.heappop(arr)
```

Dijkstra's Algorithm: Implementation

```cpp
// Maximum number of nodes, given in the problem statement
const int MN = 2005;
int n, m;
// An entry {n2, c} in adjList[n1] means that there is an edge with weight
c from n1 to n2
vector<pair<int, int>> adjList[MN];
int dist[MN];

void dijkstra(int src){
    priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
    pq.push({0, src});
    fill(dist, dist + n, INT_MAX);
    dist[src] = 0;
    while(!pq.empty()){
        auto [dist1, node1] = pq.top();
        pq.pop();
        if(dist[node1] < dist1) continue;
        for(auto [node2, edgeLen] : adjList[node1]){
            int dist2 = dist1 + edgeLen;
            if(dist[node2] <= dist2) continue;
            dist[node2] = dist2;
            pq.push({dist2, node2});
        }
    }
}
```

```python
import heapq
from math import inf

n, m = map(int, input().split())
adjList = [[] for i in range(n)]
dist = [inf] * n

def dijkstra(src):
    pq = [src]
    dist[src] = 0
    while pq:
        dist1, node1 = heapq.heappop(pq)
        if dist[node1] < dist1: continue
        for node2, edgeLen in adjList[node1]:
            dist2 = dist1 + edgeLen
            if dist[node2] <= dist2: continue
            dist[node2] = dist2
            heapq.heappush(pq, (dist2, node2))
```

Practice

https://dmoj.ca/problem/vmss7wc16c3p2 (simple BFS)
https://dmoj.ca/problem/dmopc13c3p3 (BFS on a grid)
https://dmoj.ca/problem/ccc09s4 (simple Dijkstra's)
https://dmoj.ca/problem/ccc01s3 (BFS + brute force)
https://dmoj.ca/problem/vmss7wc15c4p3 (Dijkstra's)
https://dmoj.ca/problem/cco14p2 (Dijkstra's)
https://dmoj.ca/problem/ioi11p4 (modified Dijkstra's)