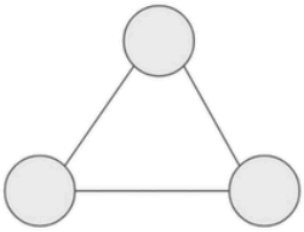
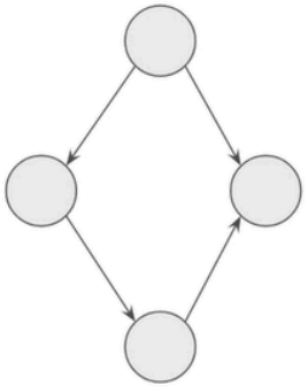
Graph

类似LinkedList的概念，内存中不一定连续的数据，由各个节点的Reference串起来组成

- 可能有环
- 分为无向图和有向图
- 没有固定入口
- 可能有多个入口



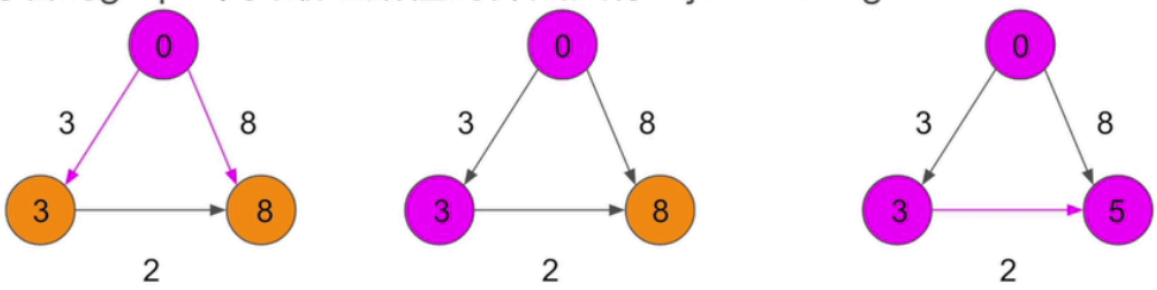


BFS (Best-First Search)

针对Non-uniform cost graph的一种算法，核心思想是优先展开最“优”的节点

如何每次快速计算最“优” → Heap

最有名的graph中找最短路径的算法又称Dijsktra’s Algorithm



BFS模板

General Steps:

1. Initialize a Heap with all starting points marked with some initial costs, a HashSet to record visited nodes
2. While heap is not empty
   a. Poll out one node
   b. If it has already been expanded (visited), skip it
   c. Otherwise mark the node as visited, update its cost
   d. If this is the destination node, return
   e. For all of its neighbors, offer them in to the heap with current node's cost + edge cost

Time: O((E + V) logV) Space: O(V)

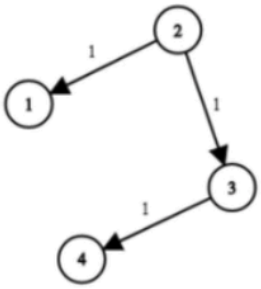
743. Network Delay Time

There are `N` network nodes, labelled `1` to `N`.

Given `times`, a list of travel times as **directed** edges `times[i] = (u, v, w)`, where `u` is the source node, `v` is the target node, and `w` is the time it takes for a signal to travel from source to target.

Now, we send a signal from a certain node `K`. How long will it take for all nodes to receive the signal? If it is impossible, return `-1`.

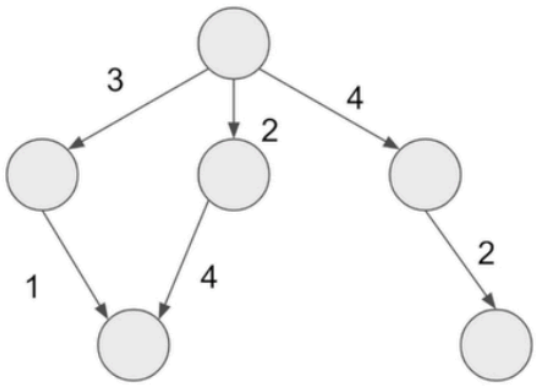
Example 1:




```
Input: times = [[2,1,1],[2,3,1],[3,4,1]], N = 4, K = 2
Output: 2
```

743. Network Delay Time

计算从初始节点到离初始节点最“远”节点的最短路径



743. Network Delay Time

建图, Adjacency List Map

1. For each edge (src, dst, cost)
    a. map[src] = {dst, cost}

```
Map<Integer, List<Cell>> map = new HashMap<>();
for (int[] time : times) {
    List<Cell> edges = map.getOrDefault(time[0], new ArrayList<>());
    edges.add(new Cell(time[1], time[2]));
    map.put(time[0], edges);
}
```

```
class Cell implements Comparable<Cell> {
    int node, time;
    Cell(int node, int time) {
        this.node = node;
        this.time = time;
    }

    public int compareTo(Cell c2) {
        return time - c2.time;
    }
}
```

743. Network Delay Time

1. Initialize Heap with all starting points marked with cost=0, a HashMap to re visited nodes and their costs
2. While heap is not empty
   a. Poll out one node
   b. If it has already been expanded (visited), skip it
   c. Otherwise mark the node as visited, update its cost
   d. If this is the destination node, return
   e. For all of its neighbors, offer them in to the heap with current node's cost + edge cost (delay time)

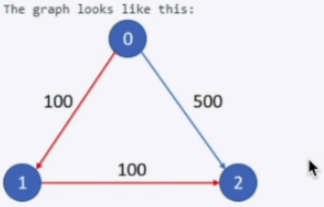
```java
public int networkDelayTime(int[][] times, int N, int k) {
    Map<Integer, List<Cell>> map = new HashMap<>();
    for (int[] time : times) {
        List<Cell> edges = map.getOrDefault(time[0], new ArrayList<>());
        edges.add(new Cell(time[1], time[2]));
        map.put(time[0], edges);
    }

    Map<Integer, Integer> costs = new HashMap<>();
    PriorityQueue<Cell> heap = new PriorityQueue<>();
    heap.offer(new Cell(k, 0));
    while (!heap.isEmpty()) {
        Cell cur = heap.poll();
        if (costs.containsKey(cur.node))
            continue;
        costs.put(cur.node, cur.time);
        if (map.containsKey(cur.node)) {
            for (Cell nei : map.get(cur.node)) {
                if (!costs.containsKey(nei.node)) {
                    heap.offer(new Cell(nei.node, cur.time + nei.time));
                }
            }
        }
    }
    if (costs.size() != N)
        return -1;
    int res = 0;
    for (int x : costs.values())
        res = Math.max(res, x);
    return res;
}
```

787. Cheapest Flights Within K Stops

There are `n` cities connected by `m` flights. Each flight starts from city `u` and arrives at `v` with a price `w`.

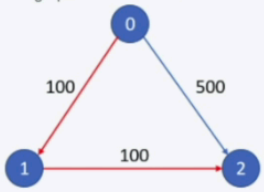
Now given all the cities and flights, together with starting city `src` and the destination `dst`, your task is to find the cheapest price from `src` to `dst` with up to `k` stops. If there is no such route, output `-1`.

Example 1:
Input:
`n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]`
`src = 0, dst = 2, k = 1`
Output: `200`
Explanation:
The graph looks like this:



The cheapest price from city `0` to city `2` with at most `1` stop costs `200`, as marked red in the picture.

Example 2:
Input:
`n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]`
`src = 0, dst = 2, k = 0`
Output: `500`
Explanation:
The graph looks like this:



The cheapest price from city `0` to city `2` with at most `0` stop costs `500`, as marked blue in the picture.

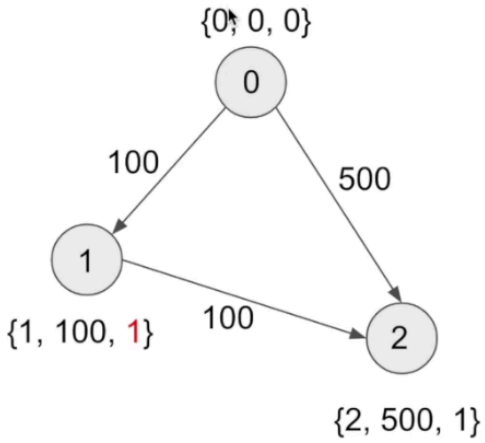
Constraints:

- The number of nodes `n` will be in range `[1, 100]`, with nodes labeled from `0` to `n - 1`.
- The size of `flights` will be in range `[0, n * (n - 1) / 2]`.
- The format of each flight will be `(src, dst, price)`.
- The price of each flight will be in the range `[1, 10000]`.
- `k` is in the range of `[0, n - 1]`.
- There will not be any duplicated flights or self cycles.

787. Cheapest Flights Within K Stops

以往我们Heap中存的节点信息有 {node, cost}，这次要多存一个stop的信息记录我们来到当前节点已经经过了多少次转机 → {node, cost, stop}，如果一个node的stop数超出K，那就不能展开此node（termination state）

k = 0



787. Cheapest Flights Within K Stops

建图，Adjacency List Map

1. For each edge (src, dst, cost)
    a. map[src] = {dst, cost}

```
Map<Integer, List<int[]>> map = new HashMap<>();
for (int[] flight : flights) {
    List<int[]> to = map.getOrDefault(flight[0], new ArrayList<>());
    to.add(new int[] {flight[1], flight[2]});
    map.put(flight[0], to);
}
```

787. Cheapest Flights Within K Stops

附件.rar

1. Initialize Heap with starting city marked with cost=0, a HashSet to record visited cities
2. While heap is not empty
   a. Poll out one node
   b. If node.stop > k, skip it
   c. Otherwise mark the node as visited, update its cost
   d. If this is the destination node, return node.cost
   e. For all of its neighbors, offer them in to the heap with current node's cost + edge cost (flight cost)

```java
public int findCheapestPrice(int n, int[][] flights, int src, int dst, int K) {
    Map<Integer, List<int[]>> map = new HashMap<>();
    for (int[] flight : flights) {
        List<int[]> to = map.getOrDefault(flight[0], new ArrayList<>());
        to.add(new int[] {flight[1], flight[2]});
        map.put(flight[0], to);
    }

    PriorityQueue<Cell> heap = new PriorityQueue<>();
    heap.offer(new Cell(src, K, 0));

    while (!heap.isEmpty()) {
        Cell cur = heap.poll();
        if (cur.dst == dst)
            return cur.price;
        if (cur.stop >= 0 && map.containsKey(cur.dst)) {
            for (int[] next : map.get(cur.dst)) {
                heap.offer(new Cell(next[0], cur.stop - 1, cur.price + next[1]));
            }
        }
    }
    return -1;
}
```

```java
class Cell implements Comparable<Cell> {
    int dst, stop, price;
    Cell(int dst, int stop, int price) {
        this.dst = dst;
        this.stop = stop;
        this.price = price;
    }

    public int compareTo(Cell other) {
        return price - other.price;
    }
}
```

Ugly Number II (264)

Find K Pairs with Smallest Sums (373)

Swim in Rising Water (778)

Kth Smallest Element in a Sorted Matrix (378)