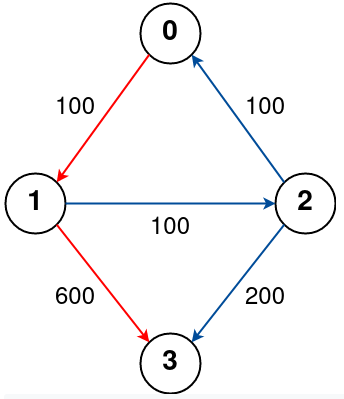
<https://leetcode.com/problems/cheapest-flights-within-k-stops/>

There are n cities connected by some number of flights. You are given an array flights where flights[i] = [fromi, toi, pricei] indicates that there is a flight from city fromi to city toi with cost pricei.

You are also given three integers src, dst, and k, return ***the cheapest price*** *from* src *to* dst *with at most* k *stops.* If there is no such route, return-1.

**Example 1:**



Input: n = 4, flights = [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]], src = 0, dst = 3, k = 1

Output: 700

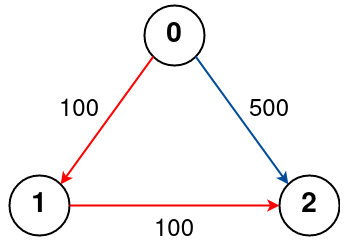
Explanation:

The graph is shown above.

The optimal path with at most 1 stop from city 0 to 3 is marked in red and has cost 100 + 600 = 700.

Note that the path through cities [0,1,2,3] is cheaper but is invalid because it uses 2 stops.

**Example 2:**



Input: n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 1

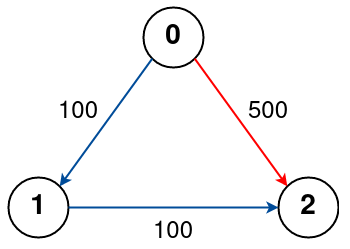
Output: 200

Explanation:

The graph is shown above.

The optimal path with at most 1 stop from city 0 to 2 is marked in red and has cost 100 + 100 = 200.

**Example 3:**



Input: n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 0

Output: 500

Explanation:

The graph is shown above.

The optimal path with no stops from city 0 to 2 is marked in red and has cost 500.

**Constraints:**

* 1 <= n <= 100
* 0 <= flights.length <= (n \* (n - 1) / 2)
* flights[i].length == 3
* 0 <= fromi, toi < n
* fromi != toi
* 1 <= pricei <= 104
* There will not be any multiple flights between two cities.
* 0 <= src, dst, k < n
* src != dst

**Attempt 1: 2022-11-26**

**Solution 1:  Find minimum distance in a Directed & Weighted Graph using BFS [Dijkstra's algorithm] (120min)**

class Solution {

public int findCheapestPrice(int n, int[][] flights, int src, int dst, int k) {

// Build graph

Map<Integer, List<int[]>> graph = new HashMap<Integer, List<int[]>>();

for(int[] flight : flights) {

graph.put(flight[0], new ArrayList<int[]>());

}

for(int[] flight : flights) {

graph.get(flight[0]).add(new int[] {flight[1], flight[2]});

}

// Dijkstra

int[] stops = new int[n];

Arrays.fill(stops, Integer.MAX\_VALUE);

PriorityQueue<int[]> minPQ = new PriorityQueue<int[]>((a, b) -> a[1] - b[1]);

// minPQ -> {src, price, steps}

minPQ.offer(new int[] {src, 0, 0});

while(!minPQ.isEmpty()) {

int[] cur = minPQ.poll();

int cur\_src = cur[0];

int cur\_price = cur[1];

int cur\_steps = cur[2];

// We have already encountered a path with a lower cost and fewer stops,

// or the number of stops exceeds the limit.

if(cur\_steps > stops[cur\_src] || cur\_steps > k + 1) {

continue;

}

stops[cur\_src] = cur\_steps;

if(cur\_src == dst) {

return cur\_price;

}

if(!graph.containsKey(cur\_src)) {

continue;

}

for(int[] neighbour : graph.get(cur\_src)) {

minPQ.offer(new int[] {neighbour[0], cur\_price + neighbour[1], cur\_steps + 1});

}

}

return -1;

}

}

**Refer to**

<https://leetcode.com/problems/cheapest-flights-within-k-stops/solution/>

**Approach 3: Dijkstra**

**Intuition**

If you are new to Dijkstra's algorithm, please see our [Leetcode Explore Card](https://leetcode.com/explore/featured/card/graph/622/single-source-shortest-path-algorithm/3862/) for more information on it!

Dijkstra's algorithm is used to find the shortest paths from a source node to all the other nodes in a weighted graph where the edge weights are positive numbers. It makes use of a priority queue (heap) to decide which edges to use.

Dijkstra's works by greedily choosing which node to investigate next. A priority queue is used to select the node that currently has the lowest price. In the previous two approaches, we used an array dist that made sure we only traversed an edge to node x if we could make an improvement on dist[x]. **In this approach, we will instead use an array stops which tracks the minimum number of stops needed to reach each node instead of the minimum price. Then, we will only traverse an edge to a node x if x has not already been visited with fewer stops. Because we are greedily choosing the node with the lowest total price, the first time we reach dst, we will have the answer.**

As per the problem, we also need to restrict the number of stops to k i.e., we can take at most k + 1 steps from the source node, so we will store the current number of stops along with each node since we aren't iterating level by level anymore.

**Algorithm**

1. Create an adjacency list where adj[X] contains all the neighbors of node X and the corresponding price it takes to move to a neighbor.
2. Initialize the stops array, storing the steps required to reach a node from the src node. We would initialize it with large values to indicate we've not reached any nodes yet.
3. Initialize a min-heap that stores a triplet {dist\_from\_src\_node, node, number\_of\_stops\_from\_src\_node}. Insert {0, src, 0} as the first triplet into the queue.
4. Perform Dijkstra's until the heap is empty:
   * Pop {dist, node, steps} from the heap
   * **If steps > stops[node], then we already visited this node with fewer steps earlier, so ignore the current triplet and move on.**
   * If steps > k + 1, then we have taken too many stops, so ignore the current triplet and move on.
   * Otherwise, check if we are at dst. If we are, then return dist as the answer.
   * If not, iterate over the neighbors of node and for each neighbor, push {dist + price, neighbor, steps + 1} onto the heap.
5. If we reach the end of the loop without returning the answer, it means we cannot reach the destination. Our answer would be -1 in this case.

**Implementation**

class Solution {

public int findCheapestPrice(int n, int[][] flights, int src, int dst, int k) {

Map<Integer, List<int[]>> adj = new HashMap<>();

for (int[] i : flights)

adj.computeIfAbsent(i[0], value -> new ArrayList<>()).add(new int[] { i[1], i[2] });

int[] stops = new int[n];

Arrays.fill(stops, Integer.MAX\_VALUE);

PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[0] - b[0]);

// {dist\_from\_src\_node, node, number\_of\_stops\_from\_src\_node}

pq.offer(new int[] { 0, src, 0 });

while (!pq.isEmpty()) {

int[] temp = pq.poll();

int dist = temp[0];

int node = temp[1];

int steps = temp[2];

// We have already encountered a path with a lower cost and fewer stops,

// or the number of stops exceeds the limit.

if (steps > stops[node] || steps > k + 1)

continue;

stops[node] = steps;

if (node == dst)

return dist;

if (!adj.containsKey(node))

continue;

for (int[] a : adj.get(node)) {

pq.offer(new int[] { dist + a[1], a[0], steps + 1 });

}

}

return -1;

}

}