<https://leetcode.com/problems/bus-routes/description/>

You are given an array routes representing bus routes where routes[i] is a bus route that the ith bus repeats forever.

For example, if routes[0] = [1, 5, 7], this means that the 0th bus travels in the sequence 1 -> 5 -> 7 -> 1 -> 5 -> 7 -> 1 -> ... forever.

You will start at the bus stop source (You are not on any bus initially), and you want to go to the bus stop target. You can travel between bus stops by buses only.

Return the least number of buses you must take to travel from source to target. Return -1 if it is not possible.

**Example 1:**

**Input:** routes = [[1,2,7],[3,6,7]], source = 1, target = 6

**Output:** 2

**Explanation:** The best strategy is take the first bus to the bus stop 7, then take the second bus to the bus stop 6.

**Example 2:**

**Input:** routes = [[7,12],[4,5,15],[6],[15,19],[9,12,13]], source = 15, target = 12

**Output:** -1

**Constraints:**

1 <= routes.length <= 500.

1 <= routes[i].length <= 10^5

All the values of routes[i] are **unique**.

sum(routes[i].length) <= 10^5

0 <= routes[i][j] < 10^6

0 <= source, target < 10^6

**Attempt 1: 2024-12-1**

**Solution 1: BFS + Level Order Traversal (30 min)**

class Solution {

    public int numBusesToDestination(int[][] routes, int source, int target) {

        // Test out by:

        // routes = [[1,7],[3,5]]

        // source = 5, target = 5

        if(source == target) {

            return 0;

        }

        // {bus stop -> all buses stop at this bus stop}

        Map<Integer, List<Integer>> map = new HashMap<>();

        for(int i = 0; i < routes.length; i++) {

            for(int j = 0; j < routes[i].length; j++) {

                List<Integer> buses = map.getOrDefault(routes[i][j], new ArrayList<Integer>());

                buses.add(i);

                map.put(routes[i][j], buses);

            }

        }

        Set<Integer> usedBuses = new HashSet<>();

        Queue<Integer> q = new LinkedList<>();

        q.offer(source);

        int busCount = 0;

        while(!q.isEmpty()) {

            busCount++;

            int size = q.size();

            for(int i = 0; i < size; i++) {

                int curBusStop = q.poll();

                List<Integer> busPassCurBusStop = map.get(curBusStop);

                // Test out by:

                // routes = [[1,2,7],[3,6,7]]

                // source = 8, target = 6

                if(busPassCurBusStop != null) {

                    for(int curBus : busPassCurBusStop) {

                        if(!usedBuses.contains(curBus)) {

                            usedBuses.add(curBus);

                            for(int j = 0; j < routes[curBus].length; j++) {

                                if(routes[curBus][j] == target) {

                                    return busCount;

                                }

                                q.offer(routes[curBus][j]);

                            }

                        }

                    }

                }

            }

        }

        return -1;

    }

}

**Refer to**

<https://www.cnblogs.com/grandyang/p/10293947.html>

这道题给了我们一堆公交线路表，然后给了起点和终点，问最少要换乘几辆公交可以从起点到达终点。这种原本只需要使用谷歌地图或者百度地图轻松实现的事，现在需要自己来实现。但这毕竟是简化版，真实情况一定要复杂得多。 这题容易进的一个误区就是把 routes 直接当作邻接链表来进行图的遍历，其实是不对的，因为 routes 数组的含义是，某个公交所能到达的站点，而不是某个站点所能到达的其他站点。这里出现了两种不同的结点，分别是站点和公交。而 routes 数组建立的是公交和其站点之间的关系，那么应该将反向关系数组也建立出来，即要知道每个站点有哪些公交可以到达。由于这里站点的标号不一定是连续的，所以可以使用 HashMap，建立每个站点和其属于的公交数组之间的映射。由于一个站点可以被多个公交使用，所以要用个数组来保存公交。既然这里求的是最少使用公交的数量，那么就类似迷宫遍历求最短路径的问题，BFS 应该是首先被考虑的解法。用队列 queue 来辅助，首先将起点S排入队列中，然后还需要一个 HashSet 来保存已经遍历过的公交（注意这里思考一下，为啥放的是公交而不是站点，因为统计的是最少需要坐的公交个数，这里一层就相当于一辆公交，最小的层数就是公交数），这些都是 BFS 的标配，应当已经很熟练了。在最开头先判断一下，若起点和终点相同，那么直接返回0，因为根本不用坐公交。否则开始 while 循环，先将结果 res 自增1，因为既然已经上了公交，那么公交个数至少为1，初始化的时候是0。这里使用 BFS 的层序遍历的写法，就是当前所有的结点都当作深度相同的一层，至于为何采用这种倒序遍历的 for 循环写法，是因为之后队列的大小可能变化，放在判断条件中可能会出错。在 for 循环中，先取出队首站点，然后要去 HashMap 中去遍历经过该站点的所有公交，若某个公交已经遍历过了，直接跳过，否则就加入 visited 中。然后去 routes 数组中取出该公交的所有站点，如果有终点，则直接返回结果 res，否则就将站点排入队列中继续遍历，参见代码如下：

class Solution {

public:

int numBusesToDestination(vector<vector<int>>& routes, int S, int T) {

if (S == T) return 0;

int res = 0;

unordered\_map<int, vector<int>> stop2bus;

queue<int> q{{S}};

unordered\_set<int> visited;

for (int i = 0; i < routes.size(); ++i) {

for (int j : routes[i]) {

stop2bus[j].push\_back(i);

}

}

while (!q.empty()) {

++res;

for (int i = q.size(); i > 0; --i) {

int t = q.front(); q.pop();

for (int bus : stop2bus[t]) {

if (visited.count(bus)) continue;

visited.insert(bus);

for (int stop : routes[bus]) {

if (stop == T) return res;

q.push(stop);

}

}

}

}

return -1;

}

};

**Java version**

class Solution {

public int numBusesToDestination(int[][] routes, int S, int T) {

// Test out by intput routes = {{1,7},{3,5}}, S = 5, T = 5

if(S == T) {

return 0;

}

// key -> stop ID, value -> buses pass the stop

Map<Integer, List<Integer>> map = new HashMap<Integer, List<Integer>>();

for(int i = 0; i < routes.length; i++) {

for(int j = 0; j < routes[i].length; j++) {

List<Integer> buses = map.getOrDefault(routes[i][j], new ArrayList<Integer>());

buses.add(i);

map.put(routes[i][j], buses);

}

}

Set<Integer> usedBuses = new HashSet<Integer>();

int busNum = 0;

Queue<Integer> q = new LinkedList<Integer>();

q.offer(S);

// Level order traverse, each new level means need 1 more bus to reach target stop

while(!q.isEmpty()) {

busNum++;

int size = q.size();

for(int i = 0; i < size; i++) {

int curStop = q.poll();

List<Integer> busesPassCurStop = map.get(curStop);

if(busesPassCurStop == null) {

continue;

}

for(int curBus : busesPassCurStop) {

if(!usedBuses.contains(curBus)) {

usedBuses.add(curBus);

// Check if current bus able to reach target stop

for(int j = 0; j < routes[curBus].length; j++) {

if(routes[curBus][j] == T) {

return busNum;

}

q.offer(routes[curBus][j]);

}

}

}

}

}

return -1;

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the code is composed of three main parts:

Constructing the s list: This loops over each route and each stop within that route, which results in O(R\*S) time complexity, where R is the number of routes and S is the average number of stops per route.

Building the graph g: This involves nested loops over each stop's routes. In the worst case, if every stop is on every route, the inner loop could run O(R^2) times for each stop. However, this is not very likely in a real-world scenario. Generally, the number of buses a stop connects to would be a smaller constant K. So this part is more accurately represented as O(S\*K^2), where K is the average number of connecting routes to any stop.

BFS traversal of the graph: In the worst case, this could visit every vertex and edge in the graph constructed in the previous step. The vertices in the graph are the bus routes, and the edges are connections between routes that share a common stop. At worst, this results in O(V+E) complexity, where V is the number of vertices (bus routes) and E is the number of edges (connections between routes).

Hence, the overall time complexity is approximately O(R\*S + S\*K^2 + V+E).

**Space Complexity**

The space complexity of the code is determined by:

The s list: This list stores a set of stops for each route, which requires O(R\*S) space.

The d dictionary: This contains at most S keys (each stop) and, for each key, a list of routes that pass by that stop. So, this contributes an additional O(S\*K) space complexity where K is the average number of routes per stop.

The g graph: The graph contains a vertex for each route, and an edge for each connection between routes that share a stop. The number of connections is at most S\*K^2 but could be less if not all buses are fully connected. Thus, this adds O(V+E) space.

The vis set and q deque: These could contain each route once, resulting in O(R) space for vis and O(R) for q at worst.

The overall space complexity is: O(R\*S + S\*K + V+E).