815. Bus Routes
Hard

You are given an array `routes` representing bus routes where `routes[i]` is a bus route that the i^{th} bus repeats forever.

- For example, if `routes[0] = [1, 5, 7]`, this means that the 0^{th} 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`

Approach #1: Breadth First Search [Accepted]

Intuition

Instead of thinking of the stops as nodes (of a graph), think of the buses as nodes. We want to take the least number of buses, which is a shortest path problem, conducive to using a breadth-first search.

Algorithm

We perform a breadth first search on bus numbers. When we start at `S`, originally we might be able to board many buses, and if we end at `T` we may have many `targets` for our goal state.

One difficulty is to efficiently decide whether two buses are connected by an edge. They are connected if they share at least one bus stop. Whether two lists share a common value can be done by set intersection (HashSet), or by sorting each list and using a two pointer approach.

To make our search easy, we will annotate the depth of each node: `info[0] = node, info[1] = depth`.

```
import java.awt.Point;


class Solution {
    public int numBusesToDestination(int[][] routes, int S, int T) {
        if (S==T) return 0;
        int N = routes.length;

        List<List<Integer>> graph = new ArrayList();
        for (int i = 0; i < N; ++i) {
            Arrays.sort(routes[i]);
            graph.add(new ArrayList());
        }
        Set<Integer> seen = new HashSet();
        Set<Integer> targets = new HashSet();
        Queue<Point> queue = new ArrayDeque();

        // Build the graph.  Two buses are connected if
```

```
// they share at least one bus stop.
for (int i = 0; i < N; ++i)
   for (int j = i+1; j < N; ++j)
      if (intersect(routes[i], routes[j])) {
         graph.get(i).add(j);
         graph.get(j).add(i);
      }

// Initialize seen, queue, targets.
// seen represents whether a node has ever been enqueued to queue.
// queue handles our breadth first search.
// targets is the set of goal states we have.
for (int i = 0; i < N; ++i) {
   if (Arrays.binarySearch(routes[i], S) >= 0) {
      seen.add(i);
      queue.offer(new Point(i, 0));
   }
   if (Arrays.binarySearch(routes[i], T) >= 0)
      targets.add(i);
}

while (!queue.isEmpty()) {
   Point info = queue.poll();
   int node = info.x, depth = info.y;
   if (targets.contains(node)) return depth+1;
```

```
            for (Integer nei: graph.get(node)) {
                if (!seen.contains(nei)) {
                    seen.add(nei);
                    queue.offer(new Point(nei, depth+1));
                }
            }
        }


        return -1;
    }


    public boolean intersect(int[] A, int[] B) {
        int i = 0, j = 0;
        while (i < A.length && j < B.length) {
            if (A[i] == B[j]) return true;
            if (A[i] < B[j]) i++; else j++;
        }
        return false;
    }
}
```

Complexity Analysis

- Time Complexity: Let NNN denote the number of buses, and bib_ibi be the number of stops on the iiith bus.
    - To create the graph, in Python we do O(∑(N−i)bi)O(\sum (N - i) b_i)O(∑(N−i)bi) work (we can improve this by checking for which of `r1, r2` is smaller), while in Java we did a O(∑bilog⁡bi)O(\sum b_i \log b_i)O(∑bilogbi) sorting step, plus our searches are O(N∑bi)O(N \sum b_i)O(N∑bi) work.

    - Our (breadth-first) search is on NNN nodes, and each node could have NNN edges, so it is O(N2)O(N^2)O(N2).
- Space Complexity: O(N2+∑bi)O(N^2 + \sum b_i)O(N2+∑bi) additional space complexity, the size of `graph` and `routes`. In Java, our space complexity is O(N2)O(N^2)O(N2) because we do not have an equivalent of `routes`. Dual-pivot quicksort (as used in `Arrays.sort(int[])`) is an in-place algorithm, so in Java we did not increase our space complexity by sorting.