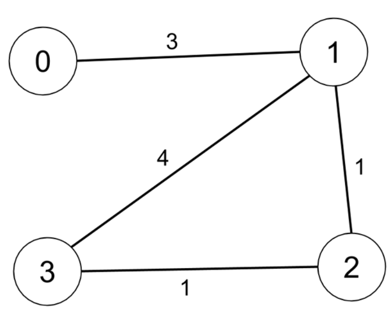
1334. Find the City With the Smallest Number of Neighbors at a Threshold Distance

There are `n` cities numbered from `0` to `n-1`. Given the array `edges` where `edges[i] = [from`$_i$`, to`$_i$`, weight`$_i$`]` represents a bidirectional and weighted edge between cities `from`$_i$ and `to`$_i$, and given the integer `distanceThreshold`.

Return the city with the smallest number of cities that are reachable through some path and whose distance is **at most** `distanceThreshold`, If there are multiple such cities, return the city with the greatest number.

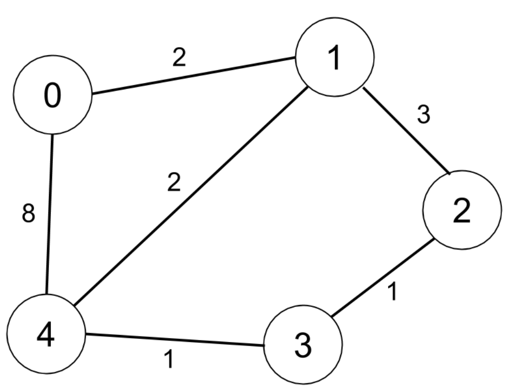
Notice that the distance of a path connecting cities ***i*** and ***j*** is equal to the sum of the edges' weights along that path.

Example 1:




```
Input: n = 4, edges = [[0,1,3],[1,2,1],
[1,3,4],[2,3,1]], distanceThreshold = 4
Output: 3
Explanation: The figure above describes the
graph.
The neighboring cities at a
distanceThreshold = 4 for each city are:
City 0 -> [City 1, City 2]
City 1 -> [City 0, City 2, City 3]
City 2 -> [City 0, City 1, City 3]
City 3 -> [City 1, City 2]
Cities 0 and 3 have 2 neighboring cities at
a distanceThreshold = 4, but we have to
return city 3 since it has the greatest
number.
```

Example 2:




```
Input: n = 5, edges = [[0,1,2],[0,4,8],
[1,2,3],[1,4,2],[2,3,1],[3,4,1]],
distanceThreshold = 2
Output: 0
Explanation: The figure above describes the
graph.
The neighboring cities at a
distanceThreshold = 2 for each city are:
City 0 -> [City 1]
City 1 -> [City 0, City 4]
City 2 -> [City 3, City 4]
City 3 -> [City 2, City 4]
City 4 -> [City 1, City 2, City 3]
The city 0 has 1 neighboring city at a
distanceThreshold = 2.
```

Constraints:

- `2 <= n <= 100`
- `1 <= edges.length <= n * (n - 1) / 2`
- `edges[i].length == 3`
- `0 <= fromi < toi < n`
- `1 <= weighti, distanceThreshold <= 10^4`
- All pairs `(fromi, toi)` are distinct.

1334. Find the City With the Smallest Number of Neighbors at a Threshold Distance

```
/*
   Dijkstra algorithm

    V=n → number of nodes
    E=|edges| → number of edges

   Time complexity: O(V+V(V+E+V)+V)=O(V^2)
   Extra space complexity: O(2VE+3V)
*/
typedef std::pair<int,int> ii;
typedef std::vector<ii> vii;
typedef std::vector<vii> vvii;

class Solution {
   public:
      vvii graph;
   public:
      void build_graph(int n, std::vector<std::vector<int>>& edges){
         graph.resize(n);
         for(auto& edge: edges){
            graph[edge[0]].push_back({edge[1],edge[2]});
            graph[edge[1]].push_back({edge[0],edge[2]});
         }
      }
```

```
 void dijkstra(int start,std::vector<int>& distances,std::vector<bool>& visited){
        distances[start]=0;
        std::priority_queue<ii,vii,std::greater<ii>> q;
        q.push({0,start});
        while(!q.empty()){
           int u=q.top().second;
           q.pop();

           if(visited[u]) continue;
           visited[u]=true;

           for(auto& neighbor: graph[u]){
              int v=neighbor.first;
              int w=neighbor.second;
              if(distances[v]>distances[u]+w){
                 distances[v]=distances[u]+w;
                 q.push({distances[v],v});
              }
           }
        }
     }

     int findTheCity(int n, std::vector<std::vector<int>>& edges, int distanceThreshold) {
        build_graph(n,edges);

        std::vector<int> citie_neighbors_count(n,0);
        for(int i=0;i<n;++i){
           std::vector<int> distances(n,INT_MAX);
           std::vector<bool> visited(n,false);
           dijkstra(i,distances,visited);
           for(auto& d: distances){
              if(d<=distanceThreshold) citie_neighbors_count[i]++;
           }
        }

        int
min_number=*std::min_element(citie_neighbors_count.begin(),citie_neighbors_count.end());

        for(int i=n-1;i>=0;i--){
           if(citie_neighbors_count[i]==min_number) return i;
        }

        return -1;
     }
};
```