CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 12

For: April 5–April 9

Goals:

- Priority Queue
- Union-Find review
- Shortest paths
- Dijkstra's Algorithm

Problem 1.
(Priority queue.)

There are situations where, given a data set, we want to know the top k highest value elements. A possible solution is to store all n elements first, sort the data set in $O(n \log n)$, then report the right-most k elements. Give an algorithm to: (i) find the top k largest elements better than $O(n \log n)$; (ii) find the top k largest elements as the elements are streaming in (for each new element that is given to you, your data structure must be ready to answer queries for the top k largest elements efficiently), and the algorithm runs faster than $O(n \log n)$.

Solution: For part (i), we can quick-select the k^{th} largest element in expected $O(n)$ time. The elements that are required will be found between this element and the end of the array.
For part (ii), we can maintain a min priority queue of no more than k elements. For every element that is given to us, add it into the priority queue. If the priority queue contains more than k elements, keep removing the smallest element until the priority queue size is k. Both insert and remove-min operations run in $O(\log k)$ time since the priority queue contains at most k elements. The overall complexity is $O(n \log k)$. Certain languages (such as C++) use this idea to implement `partial_sort`.

Problem 2.
(Horseplay.)
(Relevant Kattis Problem: https://open.kattis.com/problems/bank)

There are m bales of hay that need to be bucked and n horses to buck them.

The i^{th} bale of hay has a weight of k_i kilograms.

The i^{th} horse has a strength s_i—the maximum weight, in kilograms, it can buck—and can be hired for d_i dollars.

Bucking hay is a very physically demanding task, so to avoid the risk of injury every horse can buck at most one bale of hay.

Determine, in $O(n \log n + m \log m)$, the minimum amount, in dollars, needed to buck all bales of hay. If it is not possible to buck them all, determine that as well.

Solution: Let's sort the bales of hay by non-increasing weight, and similarly sort the horses by non-increasing strength. Assume henceforth that $k_1 \geq k_2 \geq \cdots \geq k_m$ and $s_1 \geq s_2 \geq \cdots \geq s_n$.

Consider the first bale of hay. Which horses can buck this bale of hay? They are the horses $1, 2, \ldots, i-1$, where i is the first smallest index j such that $k_1 > s_j$.

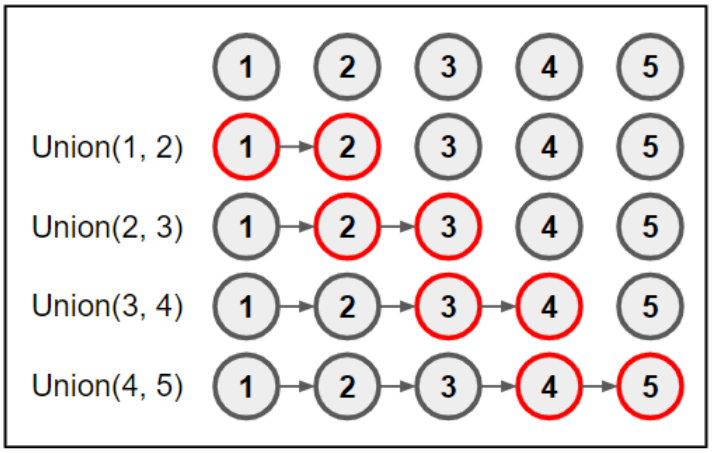
Among all these horses, we should choose the cheapest one and remove it from consideration.

Now, consider the next bale of hay. Which horses can buck this bale of hay? Again, they are the horses $1, 2, \ldots, i'-1$ where i' is the first index j such that $k_2 > s_j$. Note that this is a superset of the previous set of horses—the i' here is not less than the previous i. (Crucially, any horse that can buck the i^{th} bale of hay can also buck the j^{th} bale of hay for all $j > i$.) Among all these horses, we should choose the cheapest one that hasn't previously been chosen.

We simply perform the same process until we have bucked all bales of hay. To do this efficiently, we should maintain a (min-heap) priority queue p. Every time we consider a new bale of hay, we should insert to p the costs of the horses which can now be considered, then pop the one with the minimum cost. (If the priority queue is empty then the task is of course impossible.)

It is easy to prove that this algorithm is optimal with an exchange argument. It should also be easy to see that this runs in $O(n \log n + m \log m)$ assuming s_i, k_i and d_i are fixed-width integers.

Problem 3. (Union-Find Review)
Problem 3.a. What is the worst-case running time of the find operation in Union-Find with path compression (but no weighted union)?



Solution: The worst-case is $\Theta(n)$. You can construct a linear length tree quite easily by just doing Union(1,2), Union(2,3), Union(3,4), ..., Union(n-1, n) as shown above.

Problem 3.b. Here's another algorithm for Union-Find based on a linked list. Each set is represented by a linked list of objects, and each object is labelled (e.g., in a hash table) with a set identifier that identifies which set it is in. Also, keep track of the size of each set (e.g., using a hash table). Whenever two sets are merged, relabel the objects in the smaller set and merge the linked lists. What is the running time for performing m Union and Find operations, if there are initially n objects each in their own set?

More precisely, there is: (i) an array id where $id[j]$ is the set identifier for object j; (ii) an array $size$ where $size[k]$ is the size of the set with identifier k; (iii) an array $list$ where $list[k]$ is a linked list containing all the objects in set k.

```
Find(i, j):
    return (id[i] == id[j])

Union(i, j):
    if size[i] < size[j] then Union(j,i)
    else // size[i] >= size[j]
        k1 = id[i]
        k2 = id[j]
        for every item m in list[k2]:
            set id[m] = k1
        append list[k2] on the end of list[k1] and set list[k2] to null
        size[k1] = size[k1] + size[k2]
        size[k2] = 0
```

Assume for the purpose of this problem that you can append one linked list on to another in $O(1)$ time. (How would you do that?)

Solution: Find operations obviously cost $O(1)$. For m union operations, the cost is $m \log n$. The only expensive part is relabelling the objects in `list[k2]`. And notice that, just like in Weighted Union, each time we union two sets, the size of the smaller set at least doubles. So each object can be relabelled at most $logn$ times (as we can double the size of a set at most $logn$ times). Note that since there are m union operations, the biggest set after those operations is of size $O(m)$, and as each object in that set was updated at most $logn$ times, the total cost is $m \log n$. Of course, notice that any one operation can be expensive (each union operation have different costs). For example, the last union operation might be combining two sets of size $m/2$ and hence have cost m, while the first union operation would have a cost of $O(1)$.
The appending of one linked list to the end of another is pretty easily done in $O(1)$ through manipulation of the head and tail pointers.

Problem 3.c. Imagine we have a set of n corporations, each of which has a (string) name. In order to make a good profit, each corporation has a set of jobs it needs to do, e.g., corporation j has tasks $T^j[1 \ldots m]$. (Each corporation has at most m tasks.) Each task has a priority, i.e., an integer, and tasks must be done in priority order: corporation j must complete higher priority tasks before lower priority tasks.

Since we live in a capitalist society, every so often corporations decide to merge. Whenever that happens, two corporations merge into a new (larger) corporation. Whenever that happens, their tasks merge as well.

Design a data structure that supports three operations:

- `getNextTask(name)` that returns the next task for the corporation with the specified name.
- `executeNextTask(name)` that returns the next task for the corporation with the specified name and removes it from the set of tasks that corporation does.
- `merge(name1, name2, name3)` that merges corporation with names `name1` and `name2` into a new corporation with `name3`.

Give an efficient algorithm for solving this problem.

Solution: This can be solved using the same technique as the previous one, but now instead of using linked lists, you can use a heap to implement a priority queue for each corporation. To get the next task, just use the usual heap operation of extract-max. To merge two corporations, take the smaller remaining set of tasks and add them all to the heap for the corporation with the larger set of tasks. (Use, e.g., a hash table to store a pointer to the proper heap for each corporation.) Each time corporations merge, each item in the smaller heap pays $\log nm$ to be inserted into the new heap. (There are at most nm tasks in total.) Using the same argument as before, each item only has to be copied into a new heap at most $\log n$ times, and so the total cost of operations is $O(nm(\log n)(\log nm)) = O(nm(\log^2(n) + \log(n)\log(m))$.
You might also observe that there is a better solution. For example, you can implement each of the corporations as a $(2, 3)$ tree, and there is an efficient algorithm for merging two $(2, 3)$ trees of size n in time $O(\log n)$. This, however, requires first designing mergeable $(2, 3)$-trees. Alternatively, you may use heap structures which allow for cheap merging (such as a Binomial Heap or Fibonacci Heap).

Problem 4. Elephant Encounters
Related Kattis Problems:

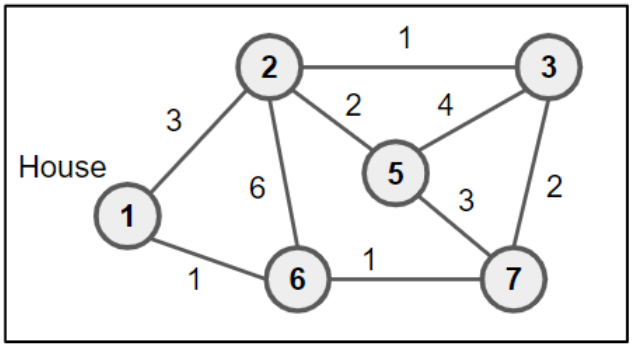
- https://open.kattis.com/problems/arbitrage
- https://open.kattis.com/problems/getshorty

Travelling can be a risky proposition: you never know when you will meet a large pink elephant. Luckily, you have been given a map where each road segment is labelled with the exact probability that someone driving on that road will not encounter a large pink elephant (*i.e.*, the probability of being safe from encountering a large pink elephant). You want to drive from New York to Los Angeles. What route should you take so as to have the smallest chance of meeting a large pink elephant?

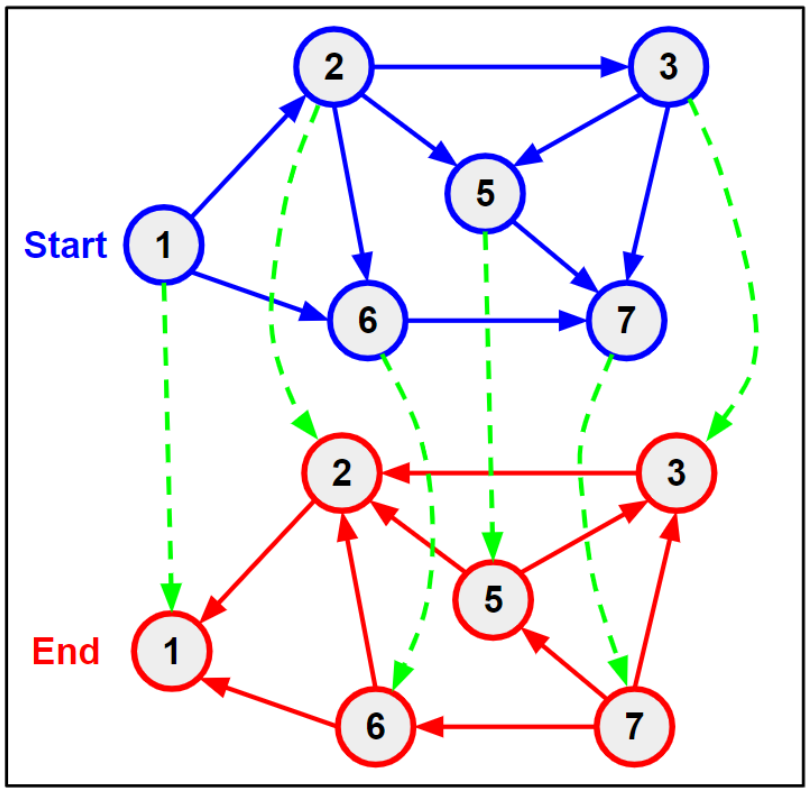
More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The safety value of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t.

Solution: This is just a modified shortest path problem, where instead of summing the edge costs you multiply the probabilities, and find the longest path (since we want to maximise our probability of being safe from the elephant). This can be done via a modification to the relax step to firstly convert the addition to a multiplication, and then relax if the result is longer instead of shorter.
Alternatively we can do the following. First we modify all the edges such that they become $-1 * \log(p(e))$ where $p(e)$ is the probability weighted on that edge. Now, this is done for two reasons. Firstly, since all the probabilities are between 0 and 1, the logarithm will be negative, hence making all edge weights negative. In order to counter any negative cycles, we multiply it by -1. Now, to maximise the cost of the path, we have to minimise the sum of the negated logarithms. Hence, running Dijsktra on this modified graph will give us the answer.

Problem 5. Running Trails



I want to go for a run. I want to go for a long run, starting from my home and ending at my home. And I want the first part of the run to be only uphill, and the second part of the run to be only downhill. I have a trail map of the nearby national park, where each location is represented as a node and each trail segment as an edge. For each node, I have the elevation (value shown in the node). Find me the longest possible run that goes first uphill and then downhill, with only one change of direction.



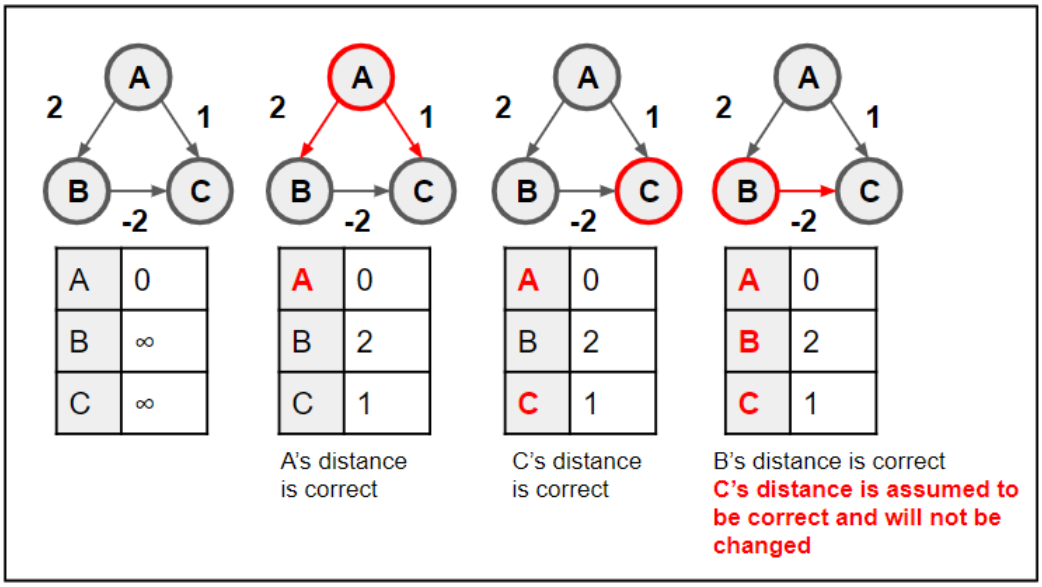
Solution: First, we want to model this as a more useful graph. Create two copies of the map graph, where in the first each edge is directed uphill (blue graph) and in the second each edge is directed downhill (red graph). Connect each node in the first copy with a directed edge to the corresponding node in the second copy (green edges). Edges connecting two nodes with the same height are omitted.

The problem now reduces to finding the longest route from the source (your home) in the first copy to the destination (your home) in the second copy.

In general, longest path is a hard problem. Luckily, there is a special property here: this is a directed acyclic graph. Notice there cannot be a cycle in the graph. If there were a cycle in the first copy, that would imply a cycle of only uphill edges, which is impossible. Similarly, there can't be a cycle in the second copy. And there are no edges from the second copy back to the first copy. So we can solve this problem by using the algorithm for finding the longest path in a directed acyclic graph, i.e., negate the weights, find a topological order of the graph, and relax the outgoing edges of each node in order.

Problem 6. (Bad Dijkstra)

Give an example of a graph where Dijkstra's Algorithm returns the wrong answer.



Solution: The idea here is to create a graph with *negative* weights so that if you run Dijkstra's Algorithm it fails. The graph shown above is one possible way to construct such a graph.

Problem 7. (A Random Problem with a dude called Dan)

Problem 7.a. Dan is on his way home from work. The city he lives in is made up of N locations, labelled from 0 to $(N-1)$. His workplace is at location 0 and his home is at location $(N-1)$. These locations are connected by M **directed** roads, each with an associated *(non-negative)* cost. To go through a road, Dan will need to pay the cost associated with that road. Usually, Dan would try to take the cheapest path home.

The thing is, Dan has just received his salary! For reasons unknown, he wants to flaunt his wealth by going through a *really expensive road.* However, he still needs to be able to make it back home with the money he has. Given that Dan can afford to spend up to D dollars on transportation, help him find **the cost of the most expensive road that he can afford to go through** on his journey back home.

Take note than Dan only cares about the most expensive road in his journey; the rest of the journey can be really cheap, or just as expensive, so long as the entire journey fits within his budget of D dollars. He is also completely focused on this goal and does not mind visiting the same location multiple times, or going through the same road multiple times.

For example, suppose Dan's budget is $D = 13$ dollars. Consider the city given in Figure 1, consisting of $N = 8$ locations and $M = 10$ roads.

The path that Dan will take is $0 \to 1 \to 4 \to 7$. In this journey, the total cost is 11 dollars and the most expensive road has a cost of 7 dollars - the road from locations 1 to 4. Therefore, the expected output for this example would be "7".

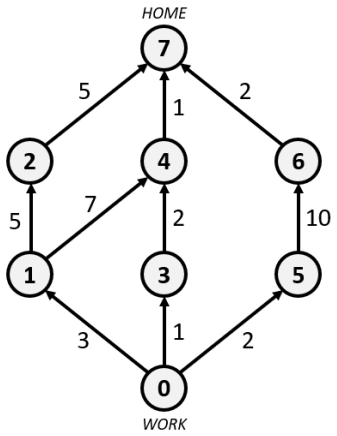


Figure 1: Example city 1

Note that this path is neither the cheapest path ($0 \to 3 \to 4 \to 7$), nor is it the most expensive path that fits within his budget of 13 dollars ($0 \to 1 \to 2 \to 7$).

There is also a more expensive road within this city - the road from locations 5 to 6 with a cost of 10 dollars. However, the only path that goes through this road, $0 \to 5 \to 6 \to 7$, has a total cost of 14 dollars which exceeds Dan's budget.

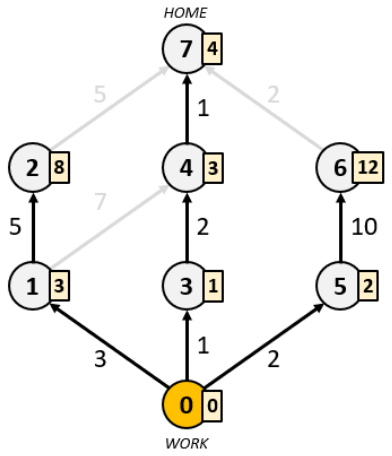
Solution: There are two possible solutions, both with differing approaches.
The first solution tests every edge ($u \to$ v) and checks if the sum of the following does not exceed D:

- The (weight of the) shortest path from vertex 0 to u
- The weight of the edge $u \to v$
- The (weight of the) shortest path from v to vertex $N-1$

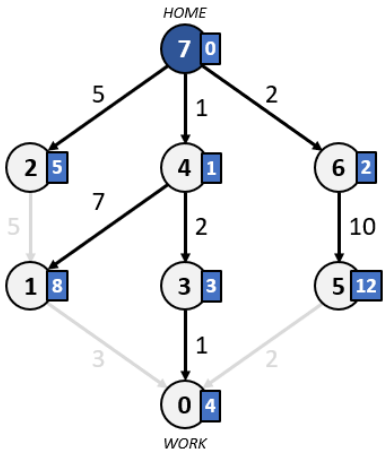
If (and only if) that sum does not exceed D, then Dan can afford to go through the edge $(u \to v)$ on his trip home. Therefore, a simple algorithm would be to iterate through all the edges and find the edge with the maximum weight that satisfies the aforementioned property.
To perform the above check efficiently, we will need to perform some pre-processing. In particular, we need an efficient way to obtain the following:

- The shortest path from vertex 0 to any vertex.
- The shortest path from any vertex to vertex $N-1$.

The former is achieved by performing Dijkstra's algorithm from vertex 0. This will get us the shortest path from vertex 0 to all other vertices in the graph.
The latter is achieved by performing Dijkstra's algorithm from vertex $N-1$ on the transpose graph (i.e. the same graph, but with all the edge directions reversed). For any vertex u, the shortest path from vertex $N-1$ to u in the transpose graph represents the shortest path from u to vertex $N-1$ in the original graph. Therefore, this will get us the shortest path from all vertices in the graph to vertex $N-1$.



Original Graph



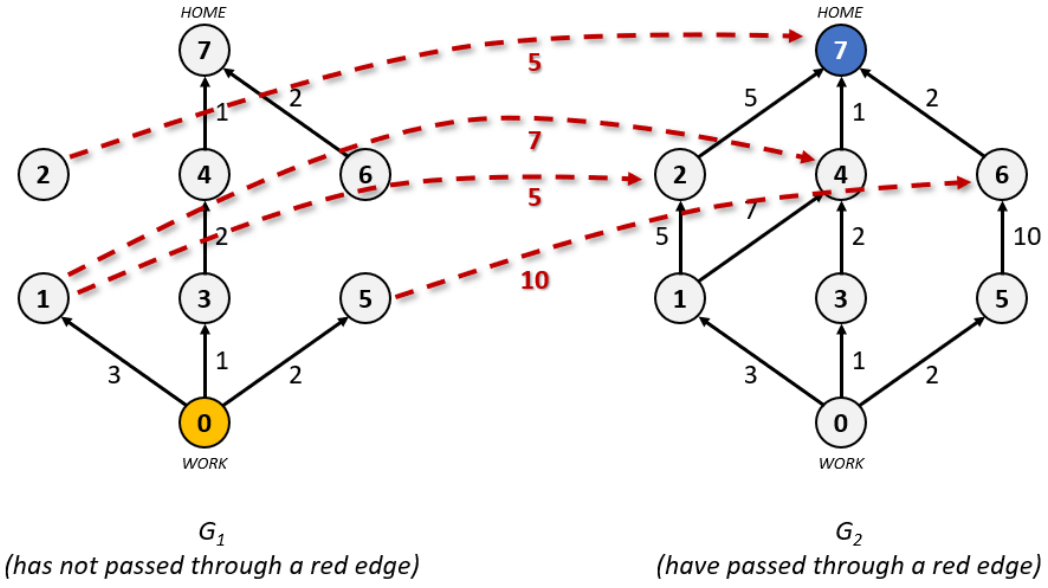
Transpose Graph

Now we can test each edge in $O(1)$ time. Including the time taken during the pre-processing, the overall time complexity is $O(M \log N + M) = O(M \log N)$.

Solution: For the second solution, we will define the following property:
Let P_K denote the property that there exists a path *(with possibly repeating edges)* from vertices 0 to $N-1$ such that:

- The path goes through an edge of weight at least K.
- The total weight of the path does not exceed D.

For example, in the given example, P_4 is True because the path $(0 \to 1 \to 2 \to 7)$ satisfies the above conditions. However, P_9 is False because there is no such path.
It follows that the answer we are looking for is the largest value of K for which P_K is true. Let $K_{\max}$ represent that largest value. Notice that P_K will also be true for any value $K \leq K_{\max}$. This means we can find $K_{\max}$ by binary searching on the set of all edge weights in the graph. In the example, this means binary searching on the array $[1, 1, 2, 2, 2, 3, 5, 5, 7, 10]$.
Testing if P_K is true for some value of K can be done in the following way:
For all the edges in the graph of weight at least K, colour those edges red. Now, we want to find a path from vertices 0 to $N-1$ while forcing it through a red edge. Duplicate the graph, labelling them G_1 and G_2. For any red edge $(u \to v)$, connect that edge from u in G_1 to v in G_2.
For example, if we're testing P_5 in the example graph, the duplicated graph looks like this:



G_1
(has not passed through a red edge)

G_2
(have passed through a red edge)

Now, any path from vertex 0 in G_1 to vertex $N-1$ in G_2 must pass through a red edge, since they are the only edges joining G_1 and G_2. Therefore, to check if P_K is true, we simply need to check the shortest path from vertex 0 in G_1 to vertex $N-1$ in G_2. P_K is true if and only if the shortest path does not exceed D.
Testing each value of K takes $O(2M \log(2N)) = O(M \log N)$, for a total time complexity of $O(M \log N \log M) = O(M(\log N)^2)$

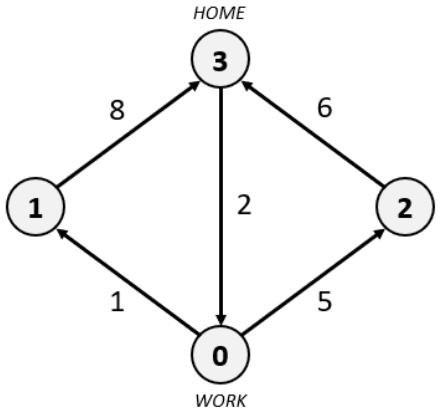


Figure 2: Example city 2

Problem 7.b. *(Optional)* Another month, another salary for Dan to flaunt. The situation is similar to that of the previous part.

This time, however, instead of maximizing the cost of the *most expensive road* in his journey, he wants to maximize the cost of *the second most expensive road* in his journey. In other words, he no longer cares about the most expensive road in his journey; that road can be 100 times more expensive than the second most expensive road in his journey for all he cares.

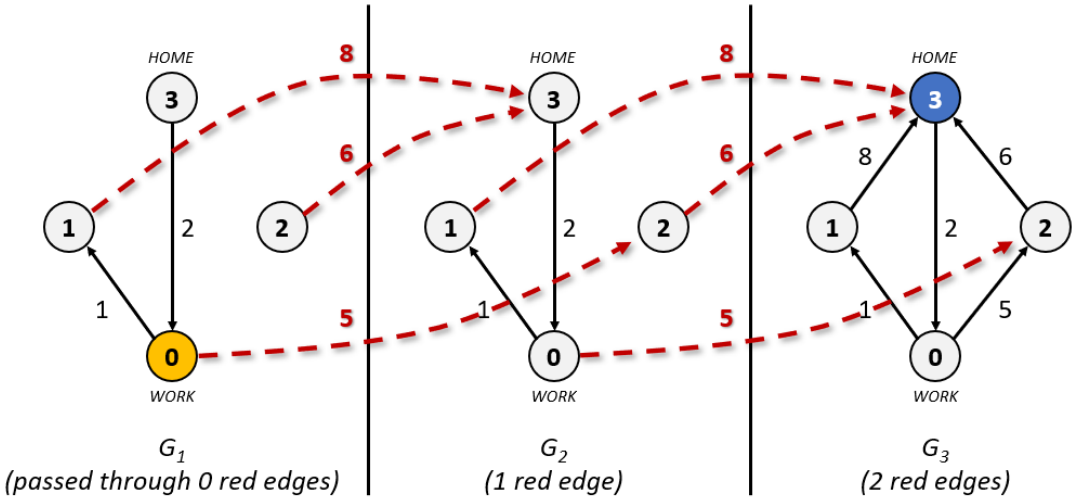
For example, consider the city in Figure 2 with $N = 4$ locations and $M = 5$ roads.

If Dan's budget is $D = 12$ dollars, the path that he will take is $0 \to 2 \to 3$. In this journey, the total cost is 11 dollars and the second most expensive road has a cost of 5 dollars, the road from locations 0 to 2. Therefore, the expected output for this example would be "5".

Notice that while he can afford to go through the path $0 \to 1 \to 3$ with an expensive 8 dollar road, the second most expensive road in that journey only costs 1 dollar.

If Dan's budget is $D = 20$ dollars, then the path he will take is $0 \to 1 \to 3 \to 0 \to 1 \to 3$. As irrational as this 20 dollar journey is, it allows him to go through the road from locations 1 to 3 twice, thus making the second most expensive road in his journey cost 8 dollars.

Solution: We can't really use the first solution from the previous part, but we can adapt the second solution.
Simply duplicate the graph twice to force a path to go through 2 red edges. For example, to test for P_5 on the example graph, the doubly-duplicated graph will look like this:



Testing each value of K takes $O(3M \log(3N)) = O(M \log N)$. Binary searching over the M possible edge weights, we have a total time complexity of $O(M \log N \log M) = O(M(\log N)^2)$, the same as the previous part.