Homework 4
Sara Pesavento
spesaven@usc.edu

SECTION 1: Heaps

1. **Design a data structure that has the following properties (assume n elements in the data structure, and that the data structure properties need to be preserved at the end of each operation):**
   • Find median takes O(1) time
   • Insert takes O(log n) time
   Do the following:
   1. Describe how your data structure will work.
   * Median = The middle number in a sorted list of numbers.

   We could keep two heaps:
   1. A max-heap to store the smaller half of the input numbers
   2. A min-heap to store the larger half of the input numbers

   It will take O(1) time to find the median, because we only need to compare the numbers at the top of the heap (Extract Min/Max).
   e.g. 2 4 5 6 10 1 3 7
   Max-heap [1, 2, 3, **4**]
   Min-heap [**5**, 6, 7, 10] → Average of getMax of max-heap and getMin of min-heap

   Insertion into the heaps will take O(logn) time.

   2. Give algorithms that implement the Find-Median() and Insert() functions.

   Insert function:
   #push to smaller half
   Push the element onto the max-heap
   #take largest and push it to the larger half of numbers
   Pop the max from the max-heap and push it onto the min-heap
   If the max-heap size is less than the min-heap size:
   #adjust so that the smaller half contains the most elements (if odd number)
   Pop the minimum from the max-heap and push it onto the min-heap

   Find-Median function:
   If smaller half > bigger half, it's an odd number:
   return the maximum of the max-heap (smaller half)
   Else:
   return the average of the maximum of max-heap and minimum of min-heap

2. **There is a stream of integers that comes continuously to a small server. The job of the server is to keep track of the k largest numbers that it has seen so far. The server has the following restrictions:**
   - It can process only one number from the stream at a time, which means it takes a number from the stream, processes it, finishes with that number and takes the next number from the stream. It cannot take more than one number from the stream at a time due to memory restriction.

- It has enough memory to store up to k integers in a simple data structure (e.g. an array), and some extra memory for computation (like comparison, etc.).
- The time complexity for processing one number must be better than O(k). Anything that is O(k) or worse is not acceptable.

Design an algorithm on the server to perform its job with the requirements listed above.

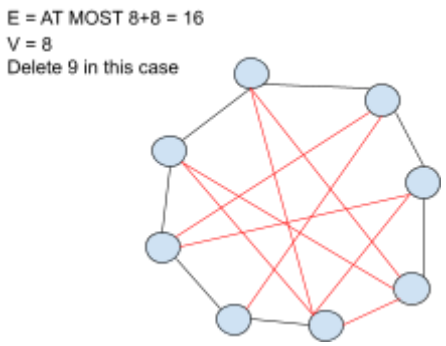
We are looking for the k largest numbers, so we could keep a min-heap to keep track of k elements.
Algorithm:
Create a min-heap
While there are still numbers in the stream:
    Take the next number
    If the min-heap size is k:
        push the next number onto the min-heap and then pop the minimum
    Else:
        push the next number onto the min-heap

Time Complexity:
For processing a single number, we will need to push and/or pop an element, and this will take O(logk).
The total time complexity for all elements would be O(nlogk) for n elements.

SECTION 2: MST

3. **Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.**
Let us suppose that graph G has two different minimum spanning trees - T1 = (V, E1) and T2 = (V, E2) - meaning they contain the same nodes in G but have a different set of edges.That would mean that there exists an edge in T1 (e1) that is not in T2. Let us assume e1 is the minimum weight edge in G. If we added this edge e1 into T2, it would create a cycle, so T2 has some edge e2 that is not in the MST T1. We have already stated that e1 has the least weight. So in the cycle created, we could replace e2 with e1 and get a spanning tree with a smaller weight. This contradicts the assumption that T2 is a MST. Therefore G only has one unique minimum spanning tree.

4. **Let us say that a graph G = (V, E) is a near tree if it is connected and has at most n+8 edges, where n = |V|. Give an algorithm with running time O(n) that takes a near tree G with costs on its edges, and returns a minimum spanning tree of G. You may assume that all edge costs are distinct.**
We want to check for cycles first. If there are no cycles, the graph is already connected so this would be a minimum spanning tree. Otherwise, we want to delete the maximum weight edge from the cycle. With V + 8 edges, we will need to iterate through 9 times.



Algorithm:

```
FindCycle(G, s):
    mark s as explored
    for each neighbor of s:
        if the node is already explored:
            return edge that cycle was found
        if node is not already explored:
            FindCycle(node)
    return None

NearTreeMST(G):
    Choose an initial node s
    for loop from 1 to 9:
        (u, v) = FindCycle(G, s)
        if (u, v) == None:
            No cycles found so return
        else:
            Find common ancestor of u and v returned in FindCycle
            Search for heaviest edge from u to ancestor
            Search for heaviest edge from v to ancestor
            Delete the heaviest edge of the two from G
```

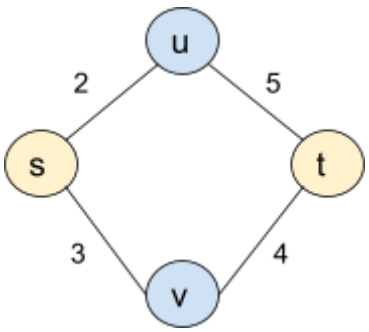
Time complexity:

O(n) - DFS takes O(n + m) or O(n) on a near tree. Finding the common ancestor takes O(n) time. Finding the heaviest edge takes O(n) time. Algorithm is run 9 times or O(9n).

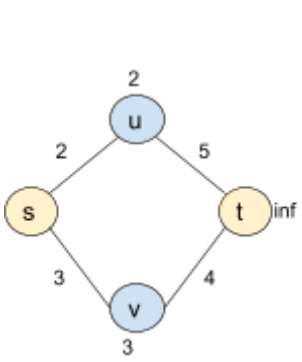
SECTION: Shortest Path

5. **Given a connected graph G = (V,E) with positive edge weights. In V, s and t are two nodes for shortest path computation, prove or disprove with explanations:**
    1) If all edge weights are unique, then there is a single shortest path between any two nodes in V .
    2) If each edge's weight is increased by k, the shortest path cost between s and t will increase by a multiple of k.
    3) If the weight of some edge e decreases by k, then the shortest path cost between s and t will decrease by at most k.
    4) If each edge's weight is replaced by its square, i.e., w to w2, then the shortest path between s and t will be the same as before but with different costs.

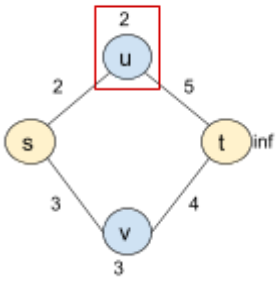
1) **False**. Consider the following case, where s → t is 7 in either path it takes. There are two possible shortest paths between these nodes.



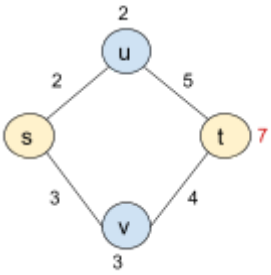
Running Djikstra's algorithm:



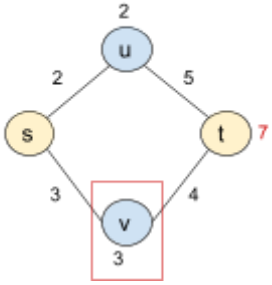
select the shortest path out of 2, 3, inf



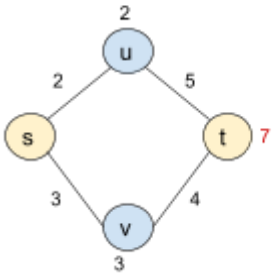
Relaxation step - check who are connected and relax them.
Just t, inf. 2 + 5 < inf, update.



select the shortest path out of 3, 7



Relaxation step - 3 + 4 is not < 7, so no updates



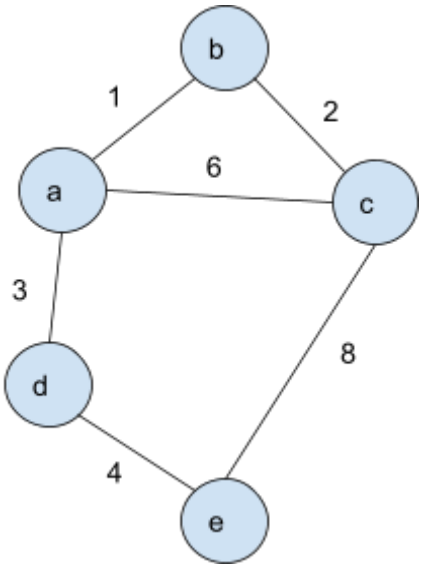
Running from s to t, Dijkstra's would choose the path s → u → t
Running from t to s, Dijkstra's would choose the path t → v → s

2) If each edge's weight is increased by k, the shortest path cost between s and t will increase by a multiple of k.
False. By increasing the paths, the shortest path may change. For example, imagine k = 10, and the shortest path has a weight of 15 with 5 edges. There may be another path of weight 18 with 2 edges. Then the new paths would be 15 + 5*10 = 65 and 18 + 2*10 = 38. The new path chosen would be 38 which is not 15 increased by a multiple of 10.

3) If the weight of some edge e decreases by k, then the shortest path cost between s and t will decrease by at most k.
False. We are not guaranteed that decreasing the edge will result in the edge maintaining a positive weight. If the weight was positive still, then the following would hold true:

> Imagine two shortest paths P and P'. If e is in the shortest path P between s and t, then by decreasing the edge by at most k, it will still remain the shortest path. If e is not in the shortest path between s and t (meaning it's in P'), then it can only decrease that path by at most k, otherwise P' would have been the shortest path to begin with. For example, if the weight of P is 15 and e belonged to another path P' that had a weight of 20. If e is 10, then the new shortest path would be 20-10 = 10. Which is decreased by 5 < 10.

However, if edges can become negative, see the following:



The edge 6 is decreasing by 56 (= -50). If the loop b-->a-->c cycles twice, the new shortest path will be -87, which is more than 56 under the original shortest path length of 8.

4) If each edge's weight is replaced by its square, i.e., w to w^2, then the shortest path between s and t will be the same as before but with different costs.
False. By taking the square of path weights, the shortest path will remain the same, and the cost may also remain the same. Imagine a shortest path from s to t where s→ u is 1 and u → t is 1. By squaring the values of 1, the path length remains 2.

6. **Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which allows you to change the weight of any one edge to zero. In other words, you may change the weight of any one edge to zero. Propose an efficient method based on Dijkstra's algorithm to find a lowest-cost path from node s to node t, given that you may set one edge weight to zero.**
Note: you will receive 10 points if your algorithm is efficient. This means your method must do better than the naive solution, where the weight of each node is set to 0 per time and the Dijkstra's algorithm is applied every time for lowest-cost path searching. You will receive full points (16 points) if your algorithm has the same run time complexity as Dijkstra's algorithm.

We want an algorithm with the same time complexity as Dijkstra's algorithm, which is O(Elog(V)). The first thing to note is that by changing any node to 0, the shortest path may change as well. You might have s and t separated by a single edge of weight 999, and took a shortest path of a few edges with a total weight of 15. By changing the 999 to 0, that would now become the shortest path.

One way to do this would be to execute Dijkstra's algorithm from node s. This will find the minimum path from node s to every other node in V. Then we can execute Dijkstra's algorithm again from node t. This will find the minimum path from node t to every other node in V. Next, for each vertex (u, v), we can consider s(u) + t(v). This will exclude the edge weight between (u, v) from consideration (essentially making it 0).

The time complexity of this algorithm would be O(Elog(V)) for running Dijkstras from node s and O(Elog(V)) for running Dijkstras from node t. Then V for looping through each nodes to track the shortest path when excluding them.

$O(Elog(V)) + O(Elog(V)) + V =$ **$O(Elog(V))$**