Joey Zhang (Student ID: 1006750437) and Chris Tong (Student ID: 1005661874)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	...	...	Total
Mark										

Table 1: Marking Table

1 Exercise 1

In accordance with CLRS (Chapter 6.5), suppose we have a priority queue data structure. A new operation DELETE(k) is added to the priority queue that removes the k largest elements from the priority queue. Use the accounting method to show that DELETE(k) can be done in amortized time O(k) while maintaining an amortized time O(logn) for $HEAP\_INSERT$. Describe your implementation for DELETE(k) and do not change the implementation of the other priority queue operations. Hint: what amortized time bound can you obtain for $HEAP\_EXTRACT\_MAX$?

Problem Solution:

1.1 Defining Upper bound

To make this work, the upper bound of HEAP_EXTRACT_MAX is chosen as O(2logn) so as to pay for its original fee of logn and to save an additional logn to pay for its cost when it is being called within the DELETE(K) function. This is in accordance with CLRS as its HEAP_EXTRACT_MAX function has the time complexity of O(logn) and here we are just assigning it a cost of O(2logn) to be in accordance with the invariant of the accounting method.

1.2 Description of Delete Function:

The Delete Function retrieves k largest numbers from within the max binary heap structure. To maintain the heap property, the HEAP_EXTRACT_MAX function is called k times inside Delete(k) function. This would mean that our Delete(k) function would have an upper-bound time complexity of O(k+k*log(n)). This is because it takes O(k) amount of time for extracting the k largest indices as we will show below. Pseudocode:

```
\\heap_A is input heap
DELETE(k, heap_A){
    original_size=heap_A.size()
    //Part one of calling HEAP_EXTRACT_MAX
    for i=1 to k{
        HEAP_EXTRACT_MAX(A) //O(log(n))
    }
    //Since we know HEAP_EXTRAM_MAX modifies the Heap's size and
    //puts the max value extracted at the end of the heap_A
    //array, we can extract the k largest elements after we finish the
    //above for-loop.

    for j from j=original_size to j=original_size-k{
        removed.append(heap_A[j]) //O(1)
    }
    return removed
}
```

Assume both for loops run for k times and that within the second for loop the append operation takes O(1) time, we have the total complexity for DELETE(k,heap_A) as O(k+k*log(n)).

And since each time we call HEAP_EXTRACT_MAX we are storing one credit, the second O(log(n)) effectively will pay for the call of heapify. Essentially O(k+k*log(n))) - O(k*log(n)) = O(k). This allows our DELETE(k,heap_A) function to have an amortized time of O(k) without having to modify HEAP_INSERT's amortized time.

2 Exercise 2

An undirected graph G = (V,E), G is said to be bipartite if the set of nodes V can be partitioned into two subsets V0 and V1 (i.e. V0 V1 = V) where every edge in E connects a node in V0 and a node in V1 . For example, the graph shown below is bipartite; this can be done by taking V0 to be the nodes on the left and V1 to be the nodes on the right.
a) Prove that if G is bipartite then it does not have a simple cycle of odd length. Hint: Prove by contradiction, assume for opposite then use induction.

Solution: Take the following Bipartite graph as an example where numbers of Edges equal to number of nodes and a cycle is exists for all nodes in Figure 1. Here we have four nodes. To see if we can complete a cycle that has odd paths, lets use the top left node in the figure. Since we have 4 edges, the maximum number of steps we can take to form a cycle is 4. If we choose to take 5 paths, we will have odd numbers of paths but will notice that no matter what we do, we can not complete a cycle with odd numbers of paths. Then we can generalize this for however many numbers of nodes.

Figure 1. The simplest formation of a cycle for nodes=4.

The basic idea: The bipartite property holds such that if we have a bipartite graph G= (V,E), it has two subset of nodes V0 and V1 where each Edge in E forms a connection from V0 to V1 or vice versa. Then whenever we travel from one node to the next node we are either travelling from V0 to V1 or V1 to V0. Thus no matter how we traverse from a node to the next, we will never to able to return back to the original node via a odd number of edges. In order for the most basic cycle to form, an "hour glass" shape must be formed. And to form that shape, at least even numbers of edges are needed in the cycle. This is seen in Figure 2 for generalizing for various numbers of nodes.

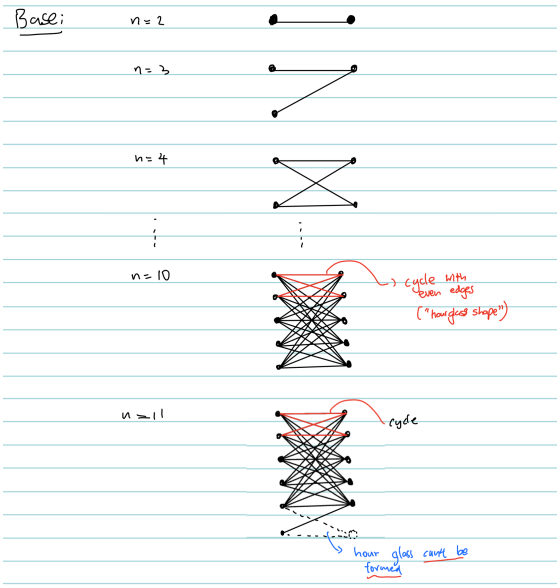

Figure 2. For cases with more numbers of nodes. Shows the "Hour Glass" shape required for the most basic shape of cycle to be formed.

b) Prove that if G does not have a simple cycle of odd length (i.e. every simple cycle of G has even length) then G is bipartite. Hint: Suppose every simple cycle of G has even length. Perform a BFS starting at any node s. Assume for now that G is connected, so that the BFS reaches all nodes; you can remove this assumption later on. Use the distance of each node from s to partition the set of nodes into two sets and prove that no edge connects two nodes placed in the same set.

Solution: We will begin by partitioning the the two set of nodes into set A and set B such that

$$setA = \{v \in V | contain\ EVEN\ shortest-paths\ between\ two\ nodes\ u_0\ and\ v\} \quad (1)$$

$$setB = \{v \in V | contain\ ODD\ shortest-paths\ between\ two\ nodes\ u_0\ and\ v\} \quad (2)$$

Where u_0 is the selected source node where we performed BFS. Suppose now that we try to connect a path from node v1 and v2 that all belong in Set A. Then a closed loop path would be only formed if v2 has a direct edge to v1: [v1, ..., u_0, u_0, ... v2]. Here we show that v1 can connect to u_0 and u_0 can connect to v2. But the only way for v2 to loop back to v1 is for v2 to have a direct connection to v1. And since v1 and v2 are both part of the same set, it makes sense that to have this direct connection formed, an odd number of edges would be needed. Since we have already stated that we don't have a simple cycle of odd length, this proves that

G is bipartite. Note: odd_num + odd_num = even_num, and even_num +even_num = even.

3 Exercise 3

Assume an Alien arrived on Earth and learned the English alphabet to converse with humans. However, the Alien arranged the letters in a distinct order, which is unfamiliar to you. You are given a set of strings consisting of unique words from this Alien language that are sorted lexicographically. Devise an algorithm that employs topological sort to determine the order of all unique letters that appear in the set. Example: words = ["wrt","wrf","er","ett","$rftt$"] $orderofletters : w < e < r < t < f$

Problem Solution:

Description of Algorithm:

1. Before running the topological sort algorithm, we first need to make sure that we have an Acyclic and Directed graph. In this case, we will be constructing our graph by iterating through the list and comparing the lexicographical words with its immediate neighbor.

    - Example: the first comparison we need is between "wrt" and "wrf," we compare their characters one by one until we find one character that is different. In this case "t" and "f" are different, and thus knowing their lexicographical property, we know "t" has to come before "f."
    - To realize this order, we can represent this in a dictionary list that stores the edges extending from each unique letter (vertices). In our case we would access the list for character "t" and store "f" as a directed edge of "t"
    - As we finish comparing the five words given "wrt", "wrf","er","ett","rftt," in four comparison, we will notice that we do have an acyclic and directed graph. A visual representation of the graph constructed after four comparisons is shown in Figure 1. The red numbers represent the order of comparisons that resulted in a directed edge.

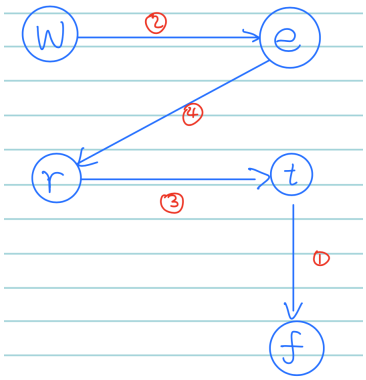

Figure 1. Graph created with given example strings

2. Now we can perform the Topological Sort on our graph obtained above.
   - Create a Stack
   - Create a vector to track Visited Nodes
   - Perform dfs algorithm on each node in the dictionary and skipping the ones that have already been visited
     - Upon finishing dfs on the neighboring nodes of current node, push the current node onto stack.
   - After running the above, pop the nodes from stack to obtain the order.
   - Note that the topological sort doesn't give unique solutions at all times. However in our case with the given lexicographical order and the nature of the problem, we should always get a unique ordering.
3. Algorithm Complexity
   - Time:
     - Creating Graph: O(w*k) where w is number of words in the list and k is the average number of characters in a word. This is because we are iterating through each word and comparing each character of each word.
     - Sorting: recall from lecture Topological sort is O(V+E) where V is number of unique letters and E is the number of total edges in graph.
   - Space:
     - Assuming that our graph is sparse and we use dictionary adjacency list we should have O(V+E) for graph

- Stack: O(V) only need to store unique vertices
- visited: O(V)
- Total: O(V+E)

4 Exercise 4

Suppose you run a currency exchange involving n different currencies. The exchange rates between these currencies can be represented as a directed graph where each vertex represents a currency and the edge weights correspond to the exchange rates. For instance, if you can exchange one US dollar for five Swiss francs, then the corresponding edge weight from the US dollar vertex to the Swiss franc vertex would be 5. It can be assumed that every pair of currencies is exchangeable, i.e., there exists a directed edge between every two vertices. Your goal is "arbitrage": to follow a sequence of exchanges that results in more US dollars for you at the end than you had at the beginning. For example, if the US-Swiss exchange is 5, the Swiss-British exchange is 1/7, and the British-US exchange is 3/2, then you can exchange 7 US dollars for 35 Swiss francs for 5 British pounds for 7.5 US dollars and make a profit of 0.5 US dollars. Devise an algorithm to determine whether you can arbitrage given a graph. Hint: Try to transform the edge-weights of this graph that allows the Bellman-Ford algorithm to tell us whether such a sequence of money-making exchanges exists.

Problem solution:

4.1 Transforming graph for Belman-Ford algorithm

Recall that the Belman-Ford algorithm allows for the detection of negative cycles for graphs with negative weights. Since in our graph we realistically require multiplication of edge weights to arrive at a solution, we need to convert all existing weights to have new modified weights of -log(w), so we can work with addition instead.
To verify that negative cycles are what we are looking for to make money we can try the example given in the problem.

- US $\rightarrow$ Swiss: -log(5)
- Swiss $\rightarrow$ British: -log(1/7)
- British $\rightarrow$ US: -log(3/2)

$$Summation = -log(5) - log(1/7) - log(3/2) = -0.0299 \tag{3}$$

If instead our summation results in positive value, it means that there is no existence of Arbitrage. The following example should demonstrate that

- US → Country A: -log(5)
- Country A → Country B: -log(1/5)
- Country B → US: -log(1/3)

$$Summation = -log(5) - log(1/5) - log(1/3) = 0.477 \tag{4}$$

This results in a positive summation and means that there is no existence of Arbitrage.

4.2 Detecting negative cycles

With the above observation, we can now devise an algorithm that allows for the detection of negative cycles in the graph. Here is the pseudo code making use of the Bellman-Ford algorithm:

```
\\assuming current_graph is a adj matrix
Arbitrage_detection(currency_graph){

    //initialize distance array elements to  be +Inf
    distance= [Inf] * V //V is all nodes in graph
    distance[start] = 0 //initialize source distance as 0

    //transform weights
    for edges_weight in nodes{ //e in |E|
        edges_weight transformed= -log(edges_weight)
    }


    //Perform Belmann-Ford for the first time
    for |V|-1 times{
        loop through all edges in currency_graph{
            //where each edge is formed by a given u and v that connect
            //update distance vector
            if(distance[v]>distance[u]+edge_weight)
                distance[v]=distance[u]+edge_weight
```

```
        }
    }

    //Perform one more Iteration of Belmann-Ford to
    \\detect a negative circle
    for |V|-1 times{
        loop through all edges in currency_graph{ //where each edge is forn
        \\by a given u and v that connect
            //update distance vector
            if(distance[v]>distance[u]+edge_weight)
                distance[v]=distance[u]+edge_weight
                return True // negative edge detected
        }
    }
}
```

If during the second iteration of Belmann-Ford algorithm, we detect an update made for any edge, it means that a negative cycle has formed and that we have detected an Arbitrage. Therefore, we return True.

The time complexity of the above algorithm is O(E+ 2*V*E) since we ran the Belmann algorithm two times and had to transform weights, but getting rid of the constant, it has a runtime of O(V*E).

5 Exercise 5: Programming part of Assignment 3

To run our code:

```
format: ./influence.py gnutella.txt T
    Where T = any floating point number for deadline
```

Additionally, please ensure to give permission to influence.py as an executable via chmod -x influence.py

5.1 how are you representing the graph, using an adjacency matrix or an adjacency list, and why?

For the graph an adjacent list was used. An adjacent list stores a list of vertices adjacent to each vertex. The adjacency matrix is mainly good for dense representations otherwise it is wasteful in terms of storage. It is optimal for small and dense

graphs or large and dense graphs, however, the density of our data set was relatively low. For this reason adjacency lists are typically a more appropriate representation because it does not use memory to represent edges that do not exist. Additionally when visiting neighbours of a vertex it's typically faster in an adjacency list since it's based on the number of neighbours which is usually less than the number of vertices.

In addition, when test running our code using the Adjacency matrix, our algorithm failed to even compute under the required 2 minutes time. It took around 2 minutes to finish running on facebook_small.txt and about an hour's worth of time on facebook_large.txt. This is because the space complexity of the adjacency matrix is $O(V^2)$ where V is the number of edges. This decreased our algorithm efficiency as we had to loop through all of $O(V^2)$ of elements in the matrix to find each node's neighbouring vertices even when certain elements in the matrix are equal to 0. And since our graph is a lot sparser, $O(V^2)$ space does not help with access time.

This is why we ended up choosing the adjacency list over the adjacency matrix. Sometimes trying different implementations can truly help us decide between the most effective data structure to use.

5.2 Which shortest path algorithm did you use and why?

Shortest-path algorithm chosen: dijkstra's.

We used dijkstra's algorithm which is a variant of breadth first search. This was chosen over a depth first search algorithm because we are searching for ALL nodes within a deadline (ie shortest path). BFS and DFS have the same time complexity O(V+E) if all edges and vertices are visited. However, since our hypothesis is that the Top influencer lies at a shallower point in the graph, BFS will return the correct answer faster. Djkstra's algorithm is "greedy" and prioritizes visiting vertices with the shortest known distance. In addition, since we are dealing with only positive weighted edges, we didn't even have to consider using Bellman-Ford's algorithm as Dijkstra's is just so much more efficient.

5.3 Discuss the plot below. What do you observe and why?

Here the graph is showing a steady increase with sometimes some inconsistencies that make is less smooth. This could be due to that the size of our testing files with 100 nodes are sensitive to changes in density. Moreover, this could be due to caching on the operating system level where accessing a dictionary require evicting as the size of our dictionary increases with density. Those are just some assumptions for what we are observing and we are not sure if that is exactly what happens.

That being said, our plot does show the general trend of steady increase consistent with the more dominant V^2. The data table and Graph are shown below.

Table 1: Dijkstra's algorithm runtime results for files with different edge densities.

All have 100 nodes	#of edges	Density (#edge/#node)	Top influencer	spread	Runtime2
	200	2	84	80	0.01649
	300	3	61	100	0.028393
	400	4	0	101	0.030186
	500	5	0	101	0.03418
	600	6	76	101	0.04033
	700	7	0	101	0.04152
	800	8	0	101	0.04504
	900	9	0	101	0.046263
	950	9.5	0	101	0.046815
	1000	10	0	101	0.051584

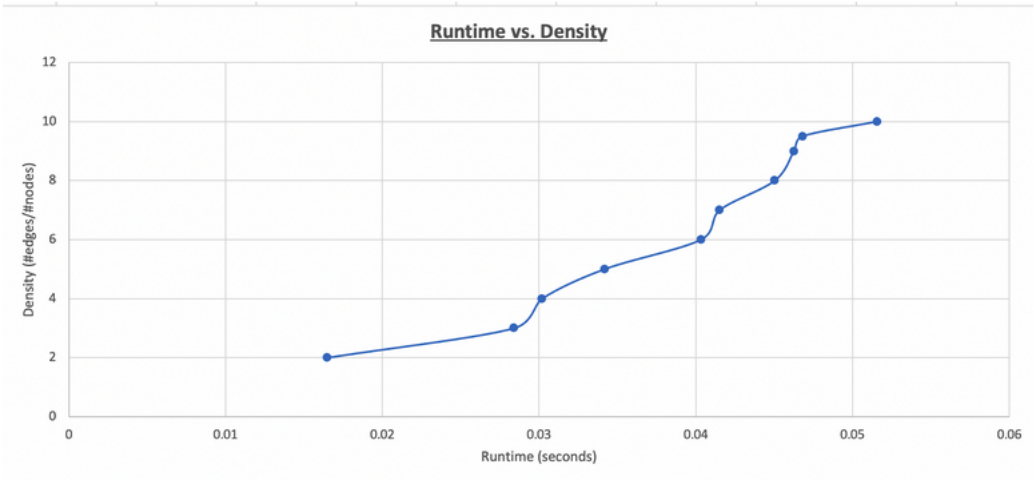

Figure 1: When plotting the runtime vs density graph of the dijkstra's algorithm on an increasingly dense graph, the runtime also increases. For Dijkstra's algorithm with a binary heap, for sparse graphs the run time is O((|E|+|V|)log|V| and O((V^2)logV) for dense graphs. In our case since we are running dijkstra's on all nodes a more accurate complexity for our overall algorithm would be O((V^2)*log(V)).

5.4 References

https://www.askpython.com/python/array/initialize-a-python-array
https://www.geeksforgeeks.org/python-program-for-dijkstras-shortest-path-algorithm-greedy-algo-7/
https://www.geeksforgeeks.org/bipartite-graph/