CSCI 3104, Fall 2017 Final Project
First Last (MM/DD) email@colorado.edu
Coding Language: Python

Problem 1

(20 pts) Given an array $\{a_1, a_2, ..., a_n\}$*, a reverse is a pair* (a_i, a_j) *such that* $i < j$ *but* $a_i > a_j$*. Design a Divide and Conquer Algorithm with a runtime of* $\mathcal{O}(nlogn)$ *for computing the number of reverses in the array. Your solution to this question needs to include both a written explanation and a Python inplementation of your algorithm, including*:

(a) *Explain how your algorithm works, including pseudocode.*

The algorithm works by using mergeSort and and counting the number of times an element is merged further to the left. During the Merge() part of mergeSort(), when an element is added from the right subarray instead of the left subarray, then that means the element on the right is less than the element on the left and there is a reverse pair. The amount of reverses is the number of index that the number moves over from the right subarray to the sorted array. In the code, this is found by j - k (the index of the right subarray minus the new index of the element in the sorted array). We only need to consider this case in mergeSort for the reversals because the other operations do not involve switching the positions of two numbers, which can not determine if there is a reverse in the original array.

```
countReverses(array, start, end, revs)
        \\ array is initial array, start is first index in array
        \\ end is last element in array, revs counts reverses
    mid = floor((end + start) / 2)
        \\ To divide array into 2 subarrays
    if start < end \\ Base case to stop recursion
        \\ Divide array into subarrays
        countReverses(array, start, mid, reverses)
        countReverses(array, mid+1, end, reverses)
        \\ Merge subarrays back into larger array
        Merge(array, start, mid, end, revs)
```

```
Merge (A, start, mind, end, revs)
    \\ Initialize empty array the length of A
    B = [] * len(A)
    \\i is index of left subarray
    i = p
    \\j is index of right subarray
    j = q+1
    \\k is index of B
    k = p
    \\ Loops through right and left subarrays
    while i <= q and j <= r
        \\ Compare the left and right subarrays
        if A[i] > A[j]
        \\ Add smallset element to B so it is sorted
            B[k++] = A[i++]
        else
            B[k++] = A[j++]
            \\ Number of reverses is how much the elements
            \\ move to the left
            revs = revs + j - k
    \\ Add remainder of right or right subarray to B
    while i <= q
        B[k++] = A[i++]
    while j <= r
        B[k++] = A[j++]
    \\ Set A equal to B
    i = p
    while i <= r
        A[i] = B[i]
        i+=1
```

(b) *Implement your algorithm in Python.*

Code is in the submitted .py file.

(c) *Randomly generate an array of 100 numbers and use it as input to run your code. Report on both the input to your code and how the output demonstrates the correctness of your algorithm.*

With the randomly generated array:

[42, 99, 51, 87, 67, 31, 60, 23, 54, 27, 94, 65, 39, 52, 47, 46, 38, 70, 22, 12, 100, 72, 68, 26, 82, 82, 74, 76, 22, 11, 40, 18, 78, 93, 13, 92, 27, 34, 40, 63, 99, 39, 9, 72, 29, 35, 69, 42, 20, 57, 19, 54, 44, 10, 73, 65, 32, 19, 83, 70, 79, 87, 31, 80, 32, 7, 49, 10, 33, 71, 37, 25, 80, 45, 73, 76, 4, 54, 8, 55, 6, 44, 67, 92, 45, 45, 68, 61, 49, 74, 3, 22, 13, 82, 96, 59, 87, 71, 19, 70]

Using the algorithm, we find that there are 2510 reverses and the sorted array is:

[3, 4, 6, 7, 8, 9, 10, 10, 11, 12, 13, 13, 18, 19, 19, 19, 20, 22, 22, 22, 23, 25, 26, 27, 27, 29, 31, 31, 32, 32, 33, 34, 35, 37, 38, 39, 39, 40, 40, 42, 42, 44, 44, 45, 45, 45, 46, 47, 49, 49, 51, 52, 54, 54, 54, 55, 57, 59, 60, 61, 63, 65, 65, 67, 67, 68, 68, 69, 70, 70, 70, 71, 71, 72, 72, 73, 73, 74, 74, 76, 76, 78, 79, 80, 80, 82, 82, 82, 83, 87, 87, 87, 92, 92, 93, 94, 96, 99, 99, 100]

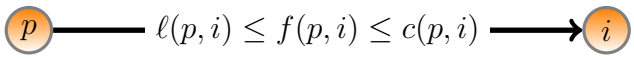
Problem 2

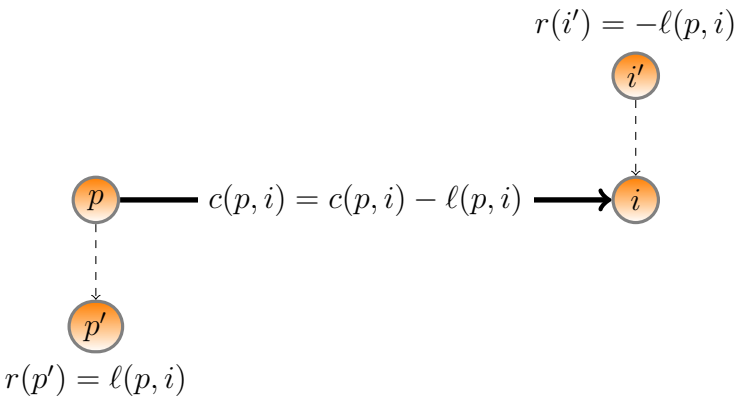
(25 pts) Suppose that you are assigned a task to do a survey about n important issues (such as education policy and health insurance mandate), by asking a group of m persons questions about these issues. Suppose that a person may not have an opinion about all the issues, and you can ask a person about an issue only if s/he has an opinion about it. We use a bipartite graph $G = \{P \cup I, E\}$ to capture whether a person $p \in P$ has an opinion about an issue $i \in I$ or not: $(p, i) \in E$ means that p has an opinion about i. For each issue i, in order to have a reliable survey you need to ask at least l_i persons about it, but you may have certain budget constraint so that you can only ask at most u_i persons about it. For each person p, you may ask her/him between b_p and t_p issues.

Given G and parameters (l_i, u_i), $i \in I$ and (b_p, t_p), $p \in P$, design an algorithm to determine if these parameters are feasible, by formulating it as a problem of finding a routing with lower bounds as in Problem 1 of homework set #9. You shall solve the problem according to the following steps.

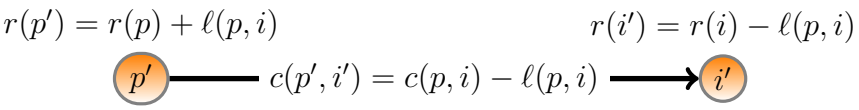
(a) *Show how to formulate the parameter feasibility problem as a problem of finding a routing with lower bounds. The resulting problem should be specified by certain graph $G' = \{V', E'\}$ with capacity $c(e)$ and lower bound $l(e)$ for each edge $e \in E'$ and demand $r(v)$ at each vertex $v \in V'$.*

To begin to make the problem into the feasibility problem, we must add additional nodes and edges to the graph. For every edge e between vertex p and i, we will create a vertex p' with an edge (p, p') and another vertex i' with edge (i', i). We can then set $r(p') = \ell(p, i)$, $r(i') = -\ell(p, i)$, and change the capacity of edge (p, i) to $c(p, i) - \ell(p, i)$. This is demonstrated in the picture bellow:





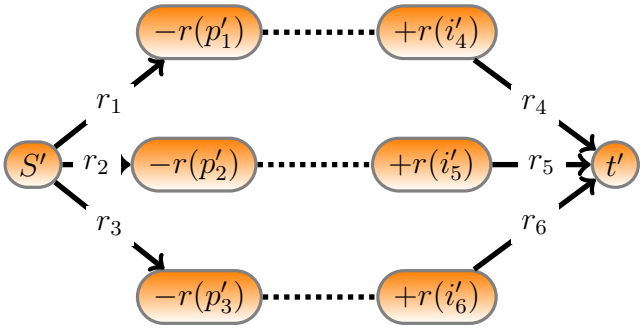
To finish transforming G into G', we need to convert p into p' and i into i':



With this, we have reduced every edge to an edge without lower bound. Only if the demands for all p' and i' can be met then problem can be solved. Hence, there exists a routing $f(e)$ in G if and only if $f'(e) = f(e) - \ell(e)$ is a routing in G'.

(b) *Further formulate the problem as a maximum flow problem as in Problem 1 of homework set # 9. The resulting problem should be specified by certain graph* $\hat{G} = \{\hat{V}, \hat{E}\}$ *with source* s*, sink* t*, and capacity* $c(e)$ *for each edge* $e \in \hat{E}$*.*

To frame G' as a maximum flow problem, we need to a add a source and a sink vertex. All "person vertices" will have negative demands because flow begins there and moves towards the "issue vertices," which will all have positive demands. We can convert graph G' into $\hat{G}$ by adding a source vertex S' with outgoing edges (S', p') to all people vertices and a sink vertex t' with incoming edges (i', t') from all issue vertices. We set the capacity of every edge (S', p') as the demand of the person vertex $r(p')$ and the capacity of every edge (i', t') as the demand of the issue vertex $r(i')$. A example of $\hat{G}$ would look like:



The total capacity of edges (S', p') is D, which is the sum of all of the demands of the sources vertices. If we saturate all edges from S', then all edges going to t' must also be at max capacity because the flow is conserved. Therefor, all demands of all other vertices must have been met. This shows that to find a routing, we can saturate the source vertex and see if the max-flow value is D.

(c) *Implement (a)-(b) in Python. Your code should take the graph G and parameters $(l_i, u_i), i \in I$ and $(b_p, t_p), p \in P$ as the input, and produce the graph $\hat{G}$ with source s, sink t, and capacity $c(e)$, $e \in \hat{E}$ as the output.*

In the submitted python file

(d) *Further implement the Ford-Fulkerson Algorithm in Python to find the maximum flow from s to t over the graph $\hat{G}$.*

The $maximum_flow$ function can be used to find the maximum flow of the graph. For one case, I found that the maximum flow was 4940 across the graph.

(e) *Generate a test case of parameters according to the following specifications, and run your code to see if the parameters generated are feasible.*

- *The number of issues $n = 10$ and the number of persons $m = 1000$;*
- *For any person p and for any issue i, s/he has a probability of 50% to have an opinion about the issue, i.e., there is a 50% probability that there is a link from p to i in graph G;*
- *For any person p, denote by h_p the number of issues that s/he has an opinion about. Let $b_p = \lfloor h_p/2 \rfloor$ and $t_p = h_p$;*
- *For each issue i, l_i is drawn uniformly from the interval $[300, 400]$ and u_i uniformly from $500, 700$.*

With these parameters, the problem is feasible.

Problem 3

(10 pts) Suppose that you have been sent back in time and have arrived at the scene of an ancient Roman battle. It is your job to assign n spears to n Roman soldiers so that each soldier has a spear. It is best if your assignments minimize the difference in height between the height of the man and the height of the spear. That is, if the i^{th} man has height m_i, and his spear has height s_i, then you want to minimize: $\sum_i |m_i - s_i|$

(a) *Design an algorithm to find the optimal, or near optimal, solution without evaluating all possible combinations. Include an explanation and pseudocode showing how your algorithm works.*

The algorithm begins by sorting both arrays from least to greatest. This can be done with the assumption of time complexity $\mathcal{O}(n \log n)$ if we use a sorting method like Mergesort. Then we can iterate through the array of spears, in runtime n, and pair each spear with the soldier at the same index. For example, the smallest soldier (soldier[0]) would pair with the smallest spear (spear[0]), the next smallest soldier (soldier[1]) would pair with the next smallest spear (spear[1]), and so on. This algorithm gives the same solution as manually pairing each soldier with its most "preferred" spear. This is because by switching two numbers from their sorted position to make their difference smaller, it would increase the difference between the other two swapped numbers by the same or possibly greater amount. For example:

$$M = [...a...b...]$$
$$S = [...c...d...]$$

If we compare the differences $Sum_1 = |(a-c)+(b-d)|$ and $Sum_2 = |(a-d)+(b-c)|$, we can see that $Sum_1 \leq Sum_2$ in all cases. This is because the arrays are sorted, so it is impossible for $|(a-c)| > |(a-d)|$ **AND** $|(b-d)| > |b-c|$. In words, it is impossible for a to "prefer" d more than c AND for b to "prefer" c more than d because the arrays are sorted. The best scenario is that a and b "prefer" c and d the same so by switching the values, the overall $\sum_i |m_i - s_i|$ stays the same.

Therefor, by matching the elements at the same indexes of the two arrays, we will get an optimal solution for which soldier should get which spear.

```
// Array of spears and soldiers, unsorted
// n is length of arrays
findMatching(spears, soldiers, n)
    // Sort both from least to greatest
    sort(spears)
    sort(soldiers)
    // Initialize empty set of pairs
    S = []
    // Iterate through arrays and add pairs to new array
    for i = 0 to n
        S.append([soldiers[i], spears[i])
    return S
```

(b) *Compare the runtime complexity of your algorithm with the complexity of a brute force solution.*

This algorithm has a runtime complexity of $\mathcal{O}(n \log n)$. This is because we had to sort both arrays. After sorting, we iterated through the array and added pairs to a new array. The total complexity of the algorithm would be $T(n) = 2n \log n + n + c$, which can be reduced to the runtime of $\mathcal{O}(n \log n)$. The worst case scenario would be similar to the best case as in both cases, both arrays would still go through the sorting algorithm. If that sorting algorithm were Mergesort, then the runtime would be the roughly the same no matter sorted the original arrays are.

In the brute force method, we would compare every spear to every soldier to find the pair that was the best fit. As we iterate through, if there was a better spear than the one the soldier was holding, then the spear is replaced and the original is added back to the array. The worst case for this algorithm is if the spear for each soldier had to be replaced every iteration. Again, there are $n = 5$ spears and soldiers. The spears are length $s_i = 5, 4, 3, 2, 1$ respectively and soldier are height $m_i = 1, 100, 200, 300, 400$. I know those are some huge soldiers but for the sake of the example, just go with it. The function would begin by comparing spear 1 (length 5) to each soldier, resulting in giving it to soldier 1. Then spear 2 (length 4) would be compared to each soldier, resulting in giving to soldier 1 and putting spear 1 back into the array. This process would continue until spear 5 (length 1) is given to soldier 1. This sequence results in n^2 operations (or 25 in this example), but the algorithm would have to continue to compare each soldier with the remaining spears, resulting in 16 operations at the next level, then 9, and 4, and 1. From this pattern we can gather:

$$25 + 16 + 9 + 4 + 1 = \sum_{i=1}^{5} \left(i^2\right)$$

Which can be generalized to:

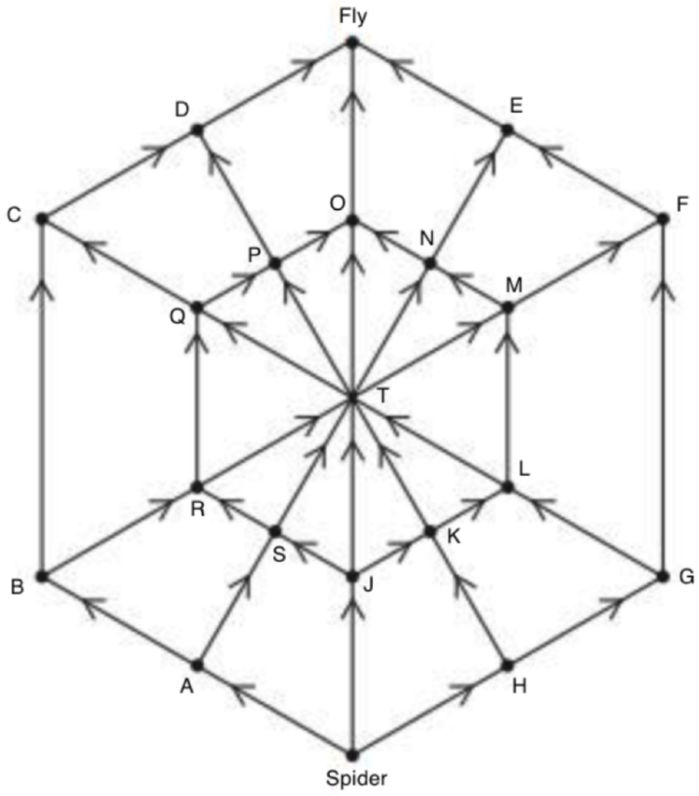
$$\sum_{i=1}^{n} \left(i^2\right) = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$

Therefor the runtime of the brute force method is $\mathcal{O}(n^3)$.

In the average case, our algorithm runs at $\mathcal{O}(n \log n)$ which is faster than the brute force method's runtime of $\mathcal{O}(n^3)$ by a factor of $\frac{n^2}{\log n}$. In the best case, the brute force algorithm would never need to replace a spear on a soldier and would therefor run at $\mathcal{O}(n^2)$, which is still slower than our algorithm.

Problem 4

(20 pts) Consider the following spider-web graph that shows a spider sitting at the bottom of its web, and a fly sitting at the top. On moodle, there is a file called graphExample.py that implements the graph using a library called NetworkX.



(a) *Write an algorithm to determine how many different ways can the spider reach the fly by moving along the web's lines in the directions indicated by the arrows?*

```
findTotalPaths(G, start, end)
stack = []  //Initialize an empty stack
stack.append(start)    // Add starting point
while stack.not_empty() // Iterates through all nodes in the graph
    temp = stack.pop()
    for i in G.temp.sinks // Adds every node succeeding temp
        stack.append(i) // to the stack
    if G.temp == start  // Sets the paths to the source node to 1
```

```
        G.temp.paths = 1
    else            // Else sets the number of paths to the node
        PathsToNode = 0 // as the sum of the paths
        for i in G.temp.sources // to the proceeding nodes
            PathsToNode = PathsToNode + i.paths
        G.temp.paths = PathsToNodes
print G.end.paths
```

The findTotalPaths() algorithm uses a dynamic programming solution to find the total number of paths between two nodes by finding the sum of the paths to all of the proceeding nodes. The function creates an empty stack and adds the initial node to it. Then the algorithm begins a while loop where it iterates through all of the nodes in a Breadth First Search. From the stack, the function pops the top node to add all succeeding nodes to the stack. The algorithm then calculates the number of paths to the popped node by finding the sum of paths of all of the proceeding nodes. This processes continues until the stack is empty, meaning the entire graph has been traversed because the graph is acyclic. Note that unlike a traditional BFS, the algorithm does not assign a "visited" value to the nodes. This is because when the sum of the proceeding paths is initially calculated, it may not be correct because some of the proceeding nodes may not yet have been calculated. This would be as if the value for node T was calculated before node R, causing an incorrect solution. By adding all succeeding nodes to the stack after a node has been popped, it guarantees that all of the paths to the proceeding nodes will be calculated the final time the number of paths to the desired node is calculated and is correct.

(b) *Implement your algorithm in Python using the NetworkX graph provided as your data structure. You may need to install NetworkX if it isn't a part of your Python installation. Do not use any of the NetworkX features that would make this problem trivial as part of your solution. However, you can use anything in NetworkX to verify your solution. Your algorithm should return an answer to the question in part (a).*

There is more information about NetworkX available here:
https://networkx.github.io/documentation/networkx-1.10/index.html

Theere are 141 paths in the graph above. The code for this is on the submitted .py file.

Problem 5

(25 pts) There are $n \geq 3$ people positioned on a field (Euclidean plane) so that each has a unique nearest neighbor. Each person has a water balloon. At a signal, everybody hurls his or her balloon at the nearest neighbor. Assume that n is odd and that nobody can miss his or her target.

(a) *Write an algorithm to answer the question: Is it true or false that there always remains at least 1 person not hit by a balloon?*

```
//Initial n*n matrix of distances to each person
oneLeftStanding(matrix, n)
    //Initialize empty matrix of people hit
    hitPeople = []
    //Iterates through entire matrix
    for i in n
        //Iterate through entire column to find minimum distance
        for j in n
            dist = machine_max
            //Would be distance 0 because same person
            if i != j
                if matrix[i][j] < dist
                    dist = matrix[i][j]
                    //Set person i to throw at person j
                    hitPeople[i] = j
    //Sort the array because everyone should be hit once if false
    //so if there is an index that doesn't match its element, then
    //there must be a person who was not hit and another that was
    //hit twice.
    sort(hitPeople)
    for i in n
        if array[i] != i
            return true
    return false
```

The algorithm iterates through the entire matrix to find the closest person in each i^{th} column. Then it stores the closest person in an array at the i^{th} position. After the algorithm has found the least distance for each person, then we sort the array. This is because if every person is hit once, then every element in the array should be the same as its index. If not, then there will be at least one element in the array that will be different from its index meaning that person

was not hit.

(b) *Implement your algorithm in Python such that it takes a data structure of people and distances and produces a data structure of who was hit by a balloon.*

Code is in the submitted .py file. The algorithm takes a matrix of distances between a person and every other person on the field. Then it iterates through the distances of each i^{th} person to find the closest person (minimum distance) and adds them to an array at the i^{th} location. The function then does this for every person in the matrix. The final returned array represents who threw at who. For example, if the element at the 0^{th} position is 2, then that means the first person threw at the third person. If the value at the 2^{nd} position is not 0, then there was another person closer to the 2^{nd} person than the 0^{th} person. This final array can be searched to find if there are any values that do not appear, meaning that there was a person who was not hit, which is always true.

(c) *Prove that your algorithm is correct. Your proof needs to include specific features of your algorithm.*

Loop Invariant:
Every i^{th} element of the subarray [0...i] of the array contains the minimum distance from the i^{th} column of the matrix.
Initialization:
The subarray starts with the first element in the matrix and the nested loop assigns the second element as their target.
Maintenance:
Every iteration of the second loop compares the current minimum distance to the j^{th} element in the column, where $0 \leq j \leq n$. There are 2 possible outcomes:

1. The current distance is less than the distance to the j^{th} element. No change is made to the subarray and j increments. The value at the i^{th} position is still the minimum distance from $0 \leq j$.
2. The current distance is greater than the distance to the j^{th} element. Distance is set to the distance to j and j becomes the new value at the i^{th} position in the subarray. The value at the i^{th} position is the minimum distance from $0 \leq j$.

Note: All of the minimum distances are unique so the algorithm does not need to consider the occurance where two distances are the same.

When the algorithm exits the nested loop, the value at the i^{th} position is the minimum distance from $0 \leq j \leq n$ in the column. The subarray contains all of the minimum distances from $0 \leq i$, which corresponds to who will target whom.

Termination:
The loop terminates when i has reached the last element in the row, which means every element in the array will contain the corresponding element with the minimum distance between them. The array contains all of the targets for every i^{th} person.

(d) *Analyze the runtime behavior of your algorithm.*

The main process of the algorithm is the nested for loop to create the array of targets. There are two nested for loops that iterate from 0 to n with some atomic functions in between. Therefor the runtime would be $T(n) = n^2 + c$, so the complexity would be $\mathcal{O}(n^2)$.
If the algorithm also finds if there are any people who are not hit, then there are two more independent functions that it must perform:

1. Sort the array
2. Iterate through the sorted array to find missing elements.

Assuming we use mergeSort to sort the array, it can be sorted in $n \log n$ time. Iterating through the sorted array involves a single loop and takes n time. The total runtime for this would be $T(n) = n^2 + n \log n + n + c$, which results in the same complexity of $\mathcal{O}(n^2)$.