Problem 1 (20 points)

Given an array $\{a_1, a_2, \cdots; a_n\}$, 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 run-time of $\mathcal{O}(n\log(n))$ for computing the number of reverses in the array. Your solution to this question needs to include both a written explanation and a Python implementation of your algorithm, including:

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

The run-time of this algorithm has to be $\mathcal{O}(n\log(n))$, and the algorithm has to be a divide-and-conquer algorithm, so I took a hint and decided to build my algorithm off of the Merge-Sort Algorithm (since it has both the run-time requirement and the algorithm type requirement). Thus, I'm going to have use the same reasoning as the Merge-Sort Algorithm. Here is the pseudocode for the Merge-Sort Algorithm (note that this pseudocode was taken from the book):

```
// The Merge-Sort Algorithm
Merge-Sort(A, p, r)
  if p < r
    q = floor((p + r)/2)
    Merge-Sort(A, p, q)
    Merge-Sort(A, q + 1, r)
    Merge(A, p, q, r)
```

I have to modify the above pseudocode for the function `Merge-Sort` to be able to return the number of reverses in the array. I will also modify the pseudocode for the Merge procedure. Here is some pseudocode (also taken from the book) for the Merge procedure:

```
// The merge procedure
Merge(A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1 ... n1+1] and R[1 ... n2+1] be new arrays
  for i = 1 to n1
    L[i] = A[p + i - 1]
  for j = 1 to n2
    R[j] = A[q + j]
  L[n1 + 1] = infinity
  R[n2 + 1] = infinity
  i = 1
  j = 1
  for k = p to r
    if L[i] <= R[j]
      A[k] = L[i]
      i = i + 1
    else
      A[k] = R[j]
      j = j + 1
```

The pseudocode for the `Merge` procedure won't need a lot of changes. All I need to do is add a counter for it to return. I will also be getting rid of lines 8 and 9 of the `Merge` pseudocode. I added a counter to sum up all of the reverses in the current recursive call, and another counter in the "`merge`" function, in the else statement when $L[i] > R[j]$, which is exactly what we want.

(b) *Implement your algorithm in Python.*

Here is my Python code, which is also included in `Letey-John-Final.py`:

```python
## Implementation of Count
def Count(A, p, q, r):
  # Initialize the variable that will hold the number of reverses in
  # the array A[p:r]
  count = 0
  # Define the size of the left half and right half of the array A[p:r]
  n1 = q - p + 1
  n2 = r - q
  # Calculate the left half and the right half of the array A[p:r]
  L = []
  R = []
  for i in range(n1):
    L.append(A[p + i])
  for j in range(n2):
    R.append(A[q + j + 1])
  # Define some indeces
  i, j, k = 0, 0, p
  # Calculate the number of reverses in the array A[p:r]
  while i < n1 and j < n2:
    if L[i] <= R[j]:
      A[k] = L[i]
      i += 1
    else:
      A[k] = R[j]
      j += 1
      count += (n1 - i)
    k += 1
  # Return the number of reverses in the array A[p:r]
  return count
## Implementation of CountReverses
def CountReverses(A, p, r):
  # Initialize the variable that will hold the number of reverses in
  # the array A[p:r]
  count = 0
  # Calculate the number of reverses in the array A[p:r]
  if p < r:
    q = math.floor((p + r)/2)
    count += CountReverses(A, p, q)
    count += CountReverses(A, q+1, r)
    count += Count(A, p, q, r)
  # Return the count of reverses in the array A[p:r]
  return count
```

(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 on how the output demonstrates the correctness of your algorithm.*

For the input `[79, 48, 100, 89, 52, 5, 61, 56, 71, 57, 4, 97, 72, 98, 4, 24, 19, 96, 96, 73, 63, 61, 57, 9, 85, 24, 39, 2, 36, 18, 32, 0, 21, 53, 73, 9, 90, 81, 82, 0, 69, 92, 39, 62, 57, 46, 6, 97, 56, 24, 88, 15, 21, 13, 70, 40, 11, 3, 23, 72,95, 66, 49, 53, 74, 33, 14, 45, 32, 14, 47, 2, 47, 86, 83, 32, 63, 7, 5, 74, 10, 64, 93, 64, 33, 37, 34, 65, 52, 89, 64, 8, 15, 64, 48, 44,45, 8, 6, 48]`, there are 2713 reverses.

Problem 2 (25 points)

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 $v \in V'$.*

My first step to creating my graph is to copy the old graph and add a super source S and a super sink T. Then, I connect each person to the super source with an edge going from the super source to the person with lower bound b_p and capacity t_p. I also connect each issue to the super sink with an edge going from the issue to the super sink with lower bound l_i and capacity u_i. I do not use demand since Rhonda says that that isn't needed.

(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 $e \in \hat{E}$.*

I have already made this a max flow problem by adding the nodes and edges that I did in part a.

(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.*

Here is my Here is my Python code, which is also included in `Letey-John-Final.py`:

```python
## Implementation of calculateHatGraph
def calculateHatGraph(G, bptp, liui):
  # Define another graph to G
  GHat = G
  # Add a super source and a super sink to the graph
```

```
GHat.add_nodes_from(['S', 'T'])
# Define indeces
personIndex, issueIndex = 0, 0
# Add lower bounds
for node in list(GHat.nodes()):
  if node[:5] == 'Issue':
    if issueIndex < len(liui):
      GHat.add_edge(node, 'T', lower=liui[issueIndex][0],
                    capacity=liui[issueIndex][1])
      issueIndex += 1
  else:
    if personIndex < len(bptp):
      GHat.add_edge('S', node, lower=bptp[personIndex][0],
                    capacity=bptp[personIndex][1])
      personIndex += 1
# Return the new graph
return GHat
```

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

As Rhonda said, I called the built in NetworkX function `nx.maximum-flow`.

(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 person $m = 1000$;*
- *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 the graph G;*
- *For any person p, denote 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]*

Please run my code located in `Letey-John-Final.py`.

Problem 3 (10 points)

Suppose 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 heights between the height of the man and the height of his spear. That is, if the i^{th} man has height m_i, and his spear has height s_i, then you want to minimize: $\sum_i \mid m_i - s_i \mid$

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

```
findOptimalSolution(spears, soldiers) {
  spears = Merge-Sort(spears) // Sort the spears using mergesort
  soldiers = Merge-Sort(soldiers) // Sort the spears using mergesort
  pairs = new list of length n
  for i = 0 to n {
    pairs[i] = (soldiers[i], spears[i])
  }
  return pairs
}
```

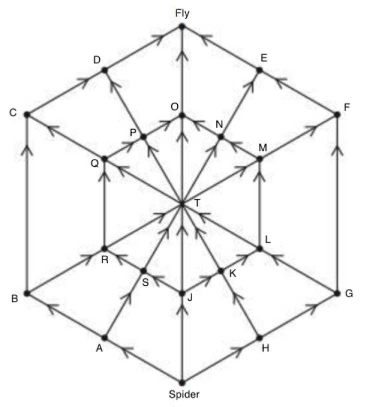
My algorithm finds the near optimal solution for this problem by sorting the two lists of spears, using the Merge-Sort algorithm, and assigns pairs via same indeces in the lists.

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

The brute force solution for this algorithm would have a runtime of $\mathcal{O}(n^2)$, since you'd go through every possible combination. My algorithm on the other hand, is $n\log(n) + n\log(n) + n$, which is $\mathcal{O}(n\log(n))$, which is way better than $\mathcal{O}(n^2)$.

Problem 4 (20 points)

Consider the following spider-web graph that shows a spider siting 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 webs lines in the directions indicated by the arrows?*

I am going to use recursion for this algorithm. My algorithm will start off by getting all of the predecessor nodes to the target (predecessors are nodes where there is an edge from them to the target). If any of those predecessor nodes happen to be the source, my algorithm will return 1, since there's exactly one path that goes from the source to the target. Else, it will go through each of the predecessors and count how many paths there are from the source to that predecessor (thus, recursion). My algorithm will return the sum of the counts that each of the recursive calls returned.

(b) *Implement your algorithm in Python using the NetworkX graph provided as your data structure. You may need to install NetworkX if it isnt 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).*

Here is my Python code, which is also included in `Letey-John-Final.py`:

```python
## Implementation of countPaths
def countPaths(G, s, t):
  # Get the predecessors of the target t
  predecessors = list(G.predecessors(t))
  # Search the list of predecessors for the source s
  for predecessor in predecessors:
    if predecessor == s:
      return 1
  # Define a variable that will hold the count of how many paths there
  # are from s to t
  count = 0
  # Go to each of the predecessors and calculate the number of paths
  for predecessor in predecessors:
    count += countPaths(G, s, predecessor)
  # Return the number of paths
  return count
```

My program reports that there are $\boxed{141}$ paths that the spider can take to get to the fly on the given web.

Problem 5 (25 points)

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 one person not hit by a balloon?*

This is true. Part (c) proves this is true for any odd n by mathematical induction, while my code finds who was 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.*

Here is my Python code, which is also included in `Letey-John-Final.py`:

```python
## Implementation of whoWasHit
def whoWasHit(distances):
  hits = [[] for i in range(len(distances))]
  for i in range(len(distances)):
    Min = float('inf')
    index = 0
    for j in range(len(distances)):
      if distances[i][j] < Min and i != j:
        Min = distances[i][j]
        index = j
    hits[index].append(i+1)
  return hits
```

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

Let's use induction for this proof. Let's first define the predicate $f(m)$ to be the predicate that there is a survivor whenever $2m + 1$ people stand in the field (and each person throws a balloon at their nearest neighbor. To prove this, we'll need to show that the predicate $f(m)$ is true for all integers m that are positive.

Base Case: $m = 1$
When $m = 1$ there are $n = 2m + 1 = 3$ people on the field. Suppose $d(i, j)$ is the distance between people i and j. We can assume, without any loss of generality, that $d(1, 2) < d(1, 3) < d(2, 3)$. Thus persons 1 and 2 both throw to each other, and person 3 throws to person 1. This means that person 3 is never hit.
Inductive Step: Assume that the predicate $f(k)$ is true for an arbitrary integer k. Prove that the predicate $f(k + 1)$ is true.
Suppose that we have $n = 2(k + 1) + 1 = 2k + 3$ people on the field with distinct distances between people. Let's define person 1 and 2 to be two people that will

both throw to each other (they're each other's nearest neighbor). Thus, we have two cases: *Case 1*, where another person throws a balloon at either person 1 or person 2, or *Case 2*, where no other person throws a balloon at either person 1 or person 2

Case 1: Because person 1 and person 2 throw balloons at each other, and someone else (person 3) throws a balloon at either person 1 or person 2, there are 3 balloon thrown in total between persons 1, 2, and 3. This implies at most $2k$ balloons are thrown at the remaining $2k+1$ people, because we had $2k+3$ balloons to start with. Therefore, there must be at least one person remaining because we don't have enough balloons.

Case 2: Other than person 1 and person 2, there are $2k+1$ people on the field. Because the distances are all distinct between pairs of neighbors, we can use the inductive hypothesis to say that there is at least one survivor that wasn't hit.

Thus, $f(k+1)$ also holds.

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

By looking at the above algorithm, you can clearly see that my algorithm has runtime $\boxed{\mathcal{O}(n^2)}$, where n is the number of people on the field.