Name: Natalie Ammerman
Due September 9, 2024

COMP 550
Homework 1 (30 points)

Directions: You may use any resources and collaborate in any manner. However, every student should make an individual submission that exactly follows the `.tex` template provided on Canvas, and every submission will be individually graded. Each homework submission should be a single PDF file.

Every homework problem is worth 5 points, and your score will always be an integer. For each problem, if your submission does not follow the `.tex` template, you will receive at most 1 point. Embedded images (e.g., a hand-drawn graph with labeled vertices) are acceptable, but any substantial amounts of text (e.g., an algorithm, explanation, or analysis) must be typed out in the `.tex` file in LaTeX.

Moreover, for each problem, if you receive at least 2 points, your score for that problem will be rounded up to 5. (This only applies to homework, not quizzes or exams.)

Advice: Try to solve each problem without consulting any resources or people for at least 15 minutes. Many problems (on homework and exams) have solutions that are similar to, or inspired by, solutions in the lectures/notes. Encountering difficulties is normal; I encourage you to attend office hours if it happens.

A relatively simple way to write pseudocode is the following:

```
ALG(A):
  t = 0
  for i = 1, ..., n:
    t += A[i]
  return t
```

Problem 1. Complete the "Punctuality Score" assignment on Gradescope. If you do so by Tue 9/10 at 11:59 pm, you will earn all 5 points for this problem. If not, you will earn 0 points for this problem. (Note that the regular deadline for this homework assignment is Mon 9/9 at 11:59 pm.)

Solution. [This problem has no solution.]

Problem 2. The input is an array A of n positive integers. Our goal is to return indices (i, j) such that $i < j$ and $(j - i) \cdot \min(A[i], A[j])$ is maximized. Describe an $O(n)$-time algorithm for this problem, briefly explain why it's correct, and analyze its running time.

Solution. [Your solution here]

```
ALG(A):
  i, j = 1, len(A)
  res_i, res_j = i, j
  while i < j:
    curr = (res_j - res_i) * min(A[res_i], A[res_j])
    if (j - i) * min(A[i], A[res_j]) > curr:
        res_i = i
    if (j - i) * min(A[res_i], A[j]) > curr:
        res_j = j
    i += 1
    j -= 1
  return res_i, res_j
```

Algorithm: Initialized two pointers, placed one at the start of the array (i) and one at the end (j). Looped while i<j. Calculated (j-i)*min(A[i], A[j]) for every pair (A[i], A[j]), and if the value was larger than the curr, curr was updated to store the new indices. Moved the pointer with the smaller value inward as to keep (j - i)*min(A[i], A[j]) maxed. Return the indices res_i and res_j.

Correctness: ALG maxes min(A[i], A[j]) while maintaining that j > i and there is a large gap (j - i). Ensured that min(A[i], A[j]) increased by only moving the pointer with the smaller value in the array.

Running Time: O(n) as number of iterations is n and each comparison and calculation is O(1).

Problem 3. The input is an array A of n distinct integers sorted in increasing order. Our goal is to return an array B such that B contains the squares of the elements of A, and B is sorted in non-decreasing order. Describe an $O(n)$-time algorithm for this problem and analyze its running time. **Example:** If $A = [-1, 0, 1, 2]$, the algorithm should return $B = [0, 1, 1, 4]$.

Solution. [Your solution here]

```
ALG(A):
  n = len(A)
  B = [0] * n
  i, j = 0, n - 1
  k = n - 1
  while i <= j:
    if abs(A[i]) > abs(A[j]):
        B[k] = A[i] * A[i]
        i += 1
    else:
        B[k] = A[j] * A[j]
        j -= 1
    k -= 1
  return B
```

Algorithm: Initialized two pointers, placed one at the start of the array (i) and one at the end (j). Initialized an array B of 0s at n vertices. Used a pointer (k) to track the current position in B, starting at the end. Looped while i was less than or equal to j to ensure they did not pass each other. Calculated the absolute value of A[i] and A[j] to make sure the square of the integer at i wouldn't be larger than that of j in case of negative ints. Added the square of whichever integer had the larger absolute value to the array B at k, decrementing k each time something was added. Return the sorted array B.

Correctness: The ALG fills B from largest to smallest square via comparing abs values at both ends of the inputted array, which ensured B was in non-decreasing order.

Running Time: O(n) as each element is only processed once.

Problem 4. The input is a connected, undirected graph $G = (V, E)$. An edge e is a *bridge* if the graph $(V, E \setminus \{e\})$ is not connected. Describe an $O(m^2)$-time algorithm that returns a list of the bridges in G, briefly explain why it's correct, and analyze its running time.

Solution. [Your solution here]

```
def findBridges(G):
   bridges = []
   n = len(G)
   for u in range(n):
      for v in G[u][:]:
         if u < v - 1:
            G[u].remove(v)
            G[v-1].remove(u+1)
            if not is_connected(G, n):
               bridges.append((u+1, v))
            G[u].append(v)
            G[v-1].append(u+1)
   return bridges

def is_connected(G, n):
   visited = [False] * n
   start = 0
   DFS(G, start, visited)
   return all(visited)

def DFS(G, node, visited):
   visited[node] = True
   for neighbor in G[node]:
      if not visited[neighbor-1]:
         DFS(G, neighbor-1, visited)
```

Algorithm: Initialized a new empty list bridges to store the result. Looped through each edge in the graph and temp. removed the edge from each vertex. Then checked whether the graph was still connected by running a DFS, if it was not connected, the edge was added to the bridges list. Restored the edge back to the graph afterward. Returned list of bridges.

Correctness: ALG correctly identifies an edge as a bridge via checking if removing the edge would make the graph disconnected. Running the DFS indicated whether all nodes were reachable, and if they were not the graph would no longer be connected and the edge that was removed is a bridge.

Running Time: For each edge we removed the edge and restored the edge which would take O(m*n) total and ran the DFS which would take O(m(n+m)). RT is O(m^2) because we checked every edge, meaning the RT for all edges was O(m(n+m)), which is equivalent to O(m^2).

Problem 5. The input is an undirected graph G. We say that G is *bipartite* if there exists an array c such that $c[u] \in \{1, 2\}$ for every $u \in V$, and for every $\{u, v\} \in E$, $c[u] \neq c[v]$. (Intuitively, G is bipartite if we can color each vertex red or blue such that every edge has a red endpoint and a blue endpoint.) Describe an $O(m + n)$-time algorithm that either returns the array c (if G is bipartite) or nothing (if G is not bipartite), briefly explain why it's correct, and analyze its running time.

Solution. [Your solution here]

```
def ALG(G):
   n = len(G)
   c = [-1] * n

   def BFS(start):
      queue = deque([start])
      c[start] = 1

      while queue:
         u = queue.popleft()
         for v in G[u]:
            v -= 1
            if c[v] == -1:
               c[v] = 3 - c[u]
               queue.append(v)
            elif c[v] == c[u]:
               return False
      return True

   for i in range(n):
      if c[i] == -1:
         if not BFS(i):
            return None

   return c
```

Algorithm: Initialized an array c of n vertices of -1 value. Ran BFS on each vertex if the value was still -1, meaning it hadn't been visited yet. In the BFS, created a queue with only the vertex in it and changed the value at c[vertex] to 1 to show it was visited. Looped while queue was not empty to pop from the queue and set u to this value, looped through each edge for that vertex and if the second vertex had not been visited yet, changed the value of c[second vertex] to the opposite of c[first vertex], and appended the second vertex to the queue. If the value of c[u] == c[v] for any edge, returned false because that graph would not be bipartite.

Correctness: The BFS makes sure that adjacent vertices got opposing values, and if they had the same value it indicated the graph was not bipartite. If the BFS completed without any issues, the graph was bipartite.

Running Time: O(m+n) because visiting and processing each vertex once in the BFS takes O(m+n), and the initial loop over each vertex only takes O(n).

Problem 6. Suppose there is a cooking competition with n dishes and k judges. Each judge ranks exactly ℓ of the n dishes from tastiest to least tasty (in their opinion). Our goal is to return a ranking R of the n dishes such that every judge's ranking is a subsequence of R. (Judges don't care about dishes other than the ℓ they ranked.) If this task is not possible (due to disagreement among the judges), the algorithm should return 0.

More formally, the input is an integer n and an array A of size k, where $A[i]$ is a list containing the ℓ dishes ranked by Judge i from tastiest to least tasty. Each dish is represented by a number in $\{1, \ldots, n\}$. The output should either be a list R of the n dishes (from tastiest to least tasty) that contains every judge's list as a subsequence, or 0 (if no such ranking exists). Describe an $O(k\ell + n)$-time algorithm for this problem, briefly explain why it's correct, and analyze its running time.

Example: Suppose there are $n = 4$ dishes, $k = 2$ judges, and each judge ranks $\ell = 3$ dishes. If $A = [[4, 2, 1], [2, 3, 1]]$, then the algorithm can output $R = [4, 2, 3, 1]$ because each $A[i]$ is a subsequence of R. If $A = [[4, 2, 1], [1, 2, 3]]$, then the algorithm must output 0, because Judge 1 thinks Dish 2 is tastier than Dish 1 but Judge 1 thinks otherwise.

Solution. [Your solution here]

```
ALG(A, n):
    G = [] * n
    in_deg = [0] * n
    for i = 1,...,n:
        G[i] = []

    for judge in A:
        for i = 1,...,len(judge):
            u = judge[i]
            v = judge[i+1]
            if v not in G[u]:
                G[u].append(v)
                in_deg[v] += 1

    queue = empty queue
    for i = 1,...,n:
        if in_deg[i] == 0:
            queue.enqueue(i)

    R = []
    while queue is not empty:
        dish = queue.dequeue()
        R.append(dish)
        for neighbor in G[dish]:
            in_deg[neighbor] -= 1
            if in_deg[neighbor] == 0:
                queue.enqueue(neighbor)

    if len(R) == n:
        return R
    else:
        return 0
```

Algorithm: Created an adjacency list to store the judges rankings for each dish and an in-degree array of n vertices of 0 value to track the judges that ranked each dish. Looped through each judge in the array and each dish the judge ranked, adding an edge to the adjacency list from one tasty dish to the next in order of tastiest to least tasty. Then checked to see if the adjacency list had any cycles by running a DFS, and returned 0 if there was one. Then ran TOPO sort on to see if it contained all the dishes and if it didn't it'd return 0.

Correctness: The ALG makes a graph where the directed edges indicate the rankings of each judge. It checks the graph for cycles, and if it has cycles, it would indicate the rankings of different judges were in direct conflict with one another. As long as no cycle is found, the topo sort would ensure that the final returned ranking would represent all of the judge's rankings as subsequences.

Running Time: In building the graph, we iterate through every judge and visit every dish in their rankings, taking O(kl) time. Then while looking for the cycles with the DFS as well as the topo sort, the worst case would be visiting each edge as well as each node once, taking O(n + kl) time. Checking the length of the result list would only take O(n) time, so the time complexity of the whole algorithm would be O(kl + n).