Solution: Quiz 2

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on the top of every page of this quiz booklet.
- You have 90 minutes to earn a maximum of 90 points. Do not spend too much time on any one problem. Skim them all first, and attack them in the order that allows you to make the most progress.
- **You are allowed two double-sided letter-sized sheet with your own notes**. No calculators, cell phones, or other programmable or communication devices are permitted.
- Write your solutions in the space provided. Pages will be scanned and separated for grading. If you need more space, write "Continued on S1" (or S2, S3) and continue your solution on the referenced scratch page at the end of the exam.
- Do not waste time and paper rederiving facts that we have studied in lecture, recitation, or problem sets. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required. Be sure to argue that your **algorithm is correct**, and analyze the **asymptotic running time of your algorithm**. Even if your algorithm does not meet a requested bound, you **may** receive partial credit for inefficient solutions that are correct.
- **Pay close attention to the instructions for each problem**. Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Parts	Points
0: Information	2	2
1: Red vs. Blue	1	16
2: Color Avoidance	1	16
3: Performance Programming	1	18
4: Toting Tea	1	18
5: Placing Posts	1	20
Total		90

Name: ______________________________

School Email: ______________________________

Problem 0. [2 points] **Information** (2 parts)

(a) [1 point] Write your name and email address on the cover page.

Solution: OK!

(b) [1 point] Write your name at the top of each page.

Solution: OK!

Problem 1. [16 points] **Red vs. Blue**

A **red-blue** graph is any dense[1] directed simple[2] graph where:

- every vertex and edge is labeled with a positive integer weight, and
- every vertex is colored either red, blue, or black.

A **red-blue path** is any path in a red-blue graph starting at a red vertex and ending at a blue vertex. The **red-blue weight** of a red-blue path is the sum of the weights of all vertices and edges in the path. Given a red-blue graph G, describe an algorithm to return the minimum red-blue weight of any red-blue path in G, with a running time that is **linear** in the size of G.

Solution: Construct a new graph G' from G by adding an auxiliary vertex s with a directed edge with weight zero to every red vertex. Then re-weight every edge in this graph to be its former weight plus the weight of its destination vertex. Then the red-blue weight of any red-blue path in G corresponds to the weight of some path in G' from s to some blue vertex. G' has only positive weights, so run Dijkstra to find shortest paths from s and return the shortest path weight to any blue vertex. Dijkstra runs in $O(|E| + |V| \log |V|)$ time. Since G is dense, $|E| = \Theta(|V|^2)$, so this running time is $O(|V|^2)$ which is linear in the size of G.

Common Mistakes:

- Not taking into account vertex weights
- Missing initial or final vertex weight
- Constructing a DAG, or running DAG relaxation on a graph with cycles
- Not specifying a start node for a shortest paths algorithm
- Not recognizing that $O(|V|^2)$ is linear in size of dense graph

[1]Recall a graph $G = (V, E)$ is dense if $|E| = \Omega(|V|^2)$.
[2]Recall a graph is simple if it has no self-loops and no multi-edges.

Problem 2. [16 points] **Color Avoidance**

A graph is **vertex-colored** if each vertex has been assigned a color.

- The **chromatic number** of a path in a vertex-colored graph is the number of distinct colors assigned to vertices on the path.
- A **path-monochromatic** graph is any vertex-colored graph where every pair of vertices assigned the same color is connected by a path with chromatic number 1.

Given an undirected connected simple path-monochromatic graph and two distinct graph vertices, describe an **efficient** algorithm to return the smallest chromatic number of any path between them.

Solution: Let G be the input graph; let s and t be the specified vertices; and let $c(a)$ denote the color of vertex a. Because G is path-monochromatic, all vertices of the same color in G induce a subgraph that is connected. So there exists a path with smallest chromatic number that enters the connected component of any color at most once (otherwise we could make such a path by replacing non-monochromatic paths between vertices of the same color with monochromatic ones). Construct a new graph G' with a vertex for each color in G (which can be found in $O(|V|)$-time using a hash table), with an undirected unweighted edge between colors c_1 and c_2 if there is an edge $\{a, b\}$ in G such that $c_1 = c(a)$ and $c_2 = c(b)$ (these edges can also be found in $O(|E|)$-time using a hash table). Then, the smallest chromatic number of any path between s and t will be one more than the length of the shortest unweighted path between $c(s)$ to $c(t)$ in G', which we can find using a breadth-first search from $c(s)$. G' is at most linear in the size of G so breadth-first search also runs in time linear in the size of G.

Common Mistakes:

- Incorrect graph construction
- Modifying breadth-first search in a way that does not find best path
- Returning shortest path length when chromatic number is length $+1$

Problem 3. [18 points] **Performance Programming**

Concert cellist Momo Ya plans to hold a concert in a European city every night of December (from Dec. 1st to the 31st), starting with a concert in Vienna on Dec. 1st.

- Between concerts, she will take a single **morning bus** to the city she will perform in next. She may play in the same city twice, but she won't play in the same city two nights in a row.
- Ya has a listing of all b one-way morning bus routes available between the c cities in Europe.
- For each city, she has estimated the **expected revenue** she would earn if she were to perform there. However, if at least one other musician has a concert in the same city on the same date, her expected revenue would be halved.

Given the bus listing, her revenue estimates, and a list of the m concerts (a city and date) that other musicians have already scheduled, describe an **efficient** algorithm to determine a city to perform in each night of December that will maximize Ya's expected performance revenue while on tour.

Solution: Let C be the set of cities, B be the set of bus routes, and M be the set of other musician concerts; let $r(i)$ denote the expected revenue of playing alone in city i; and let s correspond to Vienna. Construct a graph G with $31 \cdot c$ vertices: a vertex $v_{i,k}$ for each city $i \in C$ and for each date $k \in \{1, \ldots, 31\}$, corresponding to performing in city i on date k; and for each bus route $(i, j) \in B$ add directed edges from $v_{i,k}$ to $v_{j,k+1}$ for every $k \in \{1, \ldots, 30\}$, correspond to taking a bus on date $k + 1$ to city j from a city i after playing a concert in i on date k. G has $O(c)$ vertices and $O(b)$ edges and can be constructed in $O(c + b)$ time. Then mark each vertex $v_{i,k}$ as red if $(i, k) \in M$, corresponding to another musician playing in city i on day k, which can be done in $O(m)$ time using a hash table. Weight each edge $(v_{i,k}, v_{j,k+1})$ by $-2r(j)$, or $-r(j)$ if $v_{j,k+1}$ is red. Then the cities visited along any shortest path in G from $v_{s,1}$ to any vertex $v_{j,31}$ will correspond to a concert schedule that will maximize expected revenue. G is directed and contains no cycles (since edges always increase in k), so we can find shortest paths from $v_{s,1}$ using DAG relaxation (storing parent pointers to reconstruct the path) in $O(c + b)$ time, leading to an $O(c + b + m)$-time algorithm.

Common Mistakes:

- Running Bellman-Ford (without limiting to constant rounds)
- Running Dijkstra to find longest path
- Not specifying how to filter on musician performances (e.g., hash table)
- Not restricting to a schedule that plays every night of December

Problem 4. [18 points] **Toting Tea**

Commodore Jill Swallow[3] wants to find a home port to base her fleet of s ships. Swallow has a map showing the n **ports** in the West Indies and m two-way **shipping lanes** directly connecting pairs of them (assume every port is reachable from every other port via shipping lanes).

- Exactly $s + 1$ of the ports are marked as **client ports**. Upon choosing a home port (which should be a client port), Swallow will assign each ship in her fleet to a different client port to which the ship must make a round-trip tea delivery from her home port each month.
- Each port p is labeled with a **transit fee** f_p: the positive integer number of coins a ship must pay to transfer there from one shipping lane to another. Swallow is very rich, so you can assume her ships will always carry enough coins to cover transit fees.
- Each shipping lane ℓ is labeled with a **congestion number** k_ℓ: the positive integer number of less-well-armed competitors operating on that lane. Whenever one of Swallow's ships shares a shipping lane with a competitor, the competitor will give her ship a "donation" of 100 coins.

Given Swallow's map, describe an $O(nm)$-time algorithm to determine a port to base her fleet so as to minimize her monthly net operating costs (transit fees minus donations), assuming $s = O(\frac{n}{\log n})$.

Solution: Let P be the set of ports, $C \subset P$ be the set of client ports, and L be the set of shipping lanes. Construct a graph G with a vertex for each port $p \in P$, and for each lane $\{a, b\} \in L$, add a directed edge from port a to port b with weight $100k_{\{a,b\}} - f_b$ and a directed edge from port b to port a with weight $100k_{\{a,b\}} - f_a$. (If there is more than one lane between two ports, only add edges for one with largest $k_{\{a,b\}}$ as it will never be optimal to take a lane with fewer competitors). G has n vertices and m edges which may have positive or negative weight, and the weight of a path from any client port a to another client port b corresponds to f_b less than the net operating cost for one of Swallow's ships to sail that route (assuming a transit fee is not needed at the final client destination or the home port; an alternative assumption is also ok). Use Johnson's Algorithm to compute all-pairs shortest path distances from every client port to every other client port. If a negative-weight cycle is detected by Bellmen-Ford in Johnson's, Swallow may choose any client port as her home, as her net operating costs each month will have no finite lower-bound for any choice of home (since every port is reachable from every other). Alternatively, since only s cities are client ports, Dijkstra only needs to be run s times in Johnson's, once from each client port, leading to a running time of $O(nm + s(m + n \log n))$. Since $s = O(\frac{n}{\log n})$, this running time is $O(nm + n^2)$; and since every port is reachable via shipping lanes, G is strongly connected, so $m = \Omega(n)$ and this modified Johnson's runs in $O(nm)$ time. After Johnson's computes all-pairs shortest paths distances, return a port h for which $\sum_{c \in (C - \{h\})} \delta(h, c) + \delta(c, h) - f_c - f_h$ is minimized, which is a port that minimizes monthly net operating costs by definition. Each sum can be computed naively in $O(s)$ time, while all can be computed in $O(s^2) = O(nm)$ time; so the algorithm runs in $O(nm)$-time as desired.

(See S1 for an improvement and common mistakes)

[3] Swallow is an innocent tea merchant and definitely not a pirate.

Problem 5. [20 points] **Placing Posts**

Misty the Bear, tasked with boredom prevention, wants to build a network of one-way zip-lines to help visitors travel within her local state park.

- There are p **park locations** where posts may be built to support zip-lines.
- Each location ℓ has a known **elevation** $e(\ell)$: an integer number of feet above sea level.
- A **zip-line**, designated by ordered pair (a, b), must start atop a post at location a and end atop of a post at location b. For a zip-line (a, b) to **function**, it must drop by at least 10 feet[4].
- Each location supports one post, and any zip-line connected to a post must connect at its top.

Given a proposed set of z zip-lines and the elevation of each of the p park locations, describe an $O(z + p)$-time algorithm to return a positive integer post height for each location, that will allow every proposed zip-line to function, or return that no such height assignment is possible.

Solution: Construct a graph G with a vertex for each of the p park location and directed edge from location a to b for each of the z proposed zip-lines (a, b). G has $O(z+p)$ size and can be constructed in as much time. If G has any directed cycles, no heights can make every zip-line function (because each zip-line must decrease in height), which we can detect using depth-first search in $O(z + p)$ time. Alternatively, G is acyclic, so we can find a topological sort order of the locations using depth-first search, also in $O(z + p)$ time. Then for each location a in the reverse topological order, we can assign it a top of post elevation $e'(a)$ equal to the max of $e(a) + 1$ (posts must have positive height) and the max of $e'(b) + 10$ over all out-going edges (a, b). Each time we assign a value for $e'(a)$ for a location a, we ensure that all out-going zip-lines from it function, so all zip-lines will be functional after processing all vertices. For vertices with no outgoing edges, this computation takes $O(1)$ time, and since we process the locations in reverse topological order, each $e'(a)$ can be computed in $deg(a)$ time, so this process also takes $O(z+p)$ time. Then returning $h_a = e'(a)-e(a)$ for each location a provides positive post heights that satisfy the desired properties.

The above algorithm returns the lowest possible post heights. However, since we are not given a limit on the height of poles, an easier valid solution could be to add pole height to every location so that the top of all poles are at one more than the maximum elevation, and then add $10i$ on top of the pole in the i^{th} location in the reverse topological order.

Common Mistakes:

- Running a shortest-path algorithm incorrectly or without justification
- Not taking into account elevation when computing heights
- Determining feasibility checking for negative weight cycles, not cycles in general
- Removing zip-lines if elevations without posts go up-hill
- Returning top post elevations rather than post heights

[4]i.e., two posts of heights h_a and h_b must be built at a and b respectively such that $h_a + e(a) \geq h_b + e(b) + 10$.

SCRATCH PAPER 1. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S1" on the problem statement's page.

Solution: Problem 4, Improved solution
It is actually possible to improve on the requested running time of $O(nm)$. Instead, make **undirected** graph G on the ports, with edge $\{a, b\}$ having weight $f_a + f_b - 200k_{\{a,b\}}$ for each lane. Now the weight of a path from a client port a to another client port b corresponds to $f_a + f_b$ more than twice the net operating cost for one of Swallow's ships to sail that route. In particular, operating costs can still be computed from the APSP distances in G. However, we can now compute APSP by simply running Dijkstra s times without using Bellman-Ford first. If all edge weights are positive, then Dijkstra will succeed; but if any edge weight is negative, then there is a negative cycle and we can abort and resort to piracy. The running time of this algorithm is $O(sm + sn \log n) \subset O(sm + n^2)$. In particular, if $n = o(m)$, then $O(sm + n^2) \subset o(nm)$, which is an improvement.

Common Mistakes:

- Running Bellman-Ford from a supernode to all ports
- Running Dijkstra from all ports instead of just client ports
- Not accounting for round trips
- Not accounting for negative cycles

SCRATCH PAPER 2. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S2" on the problem statement's page.

SCRATCH PAPER 3. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S3” on the problem statement’s page.