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 120 minutes to earn a maximum of 120 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, S4, S5) 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: Relax!	3	12
2: Negative-Weight Cycles	1	8
3: Directed Graph Radius	1	8
4: Future MIT	1	15
5: Gone to the Dogs	1	15
6: Troll Tolls	1	15
7: Muscle Mono-Toning	1	15
8: Super Lario World	1	15
9: Dealer's Choice	1	15
Total		120

Name: ______________________________

School Email: ______________________________

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

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

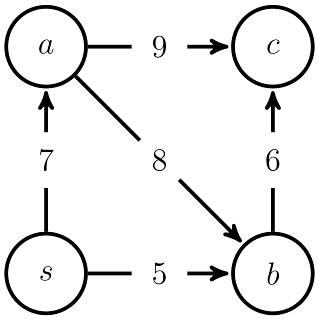
Solution: OK!

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

Solution: OK!

Problem 1. [12 points] **Relax!**

Consider the graph $G = (V, E)$, where $V = \{a, b, c, s\}$, $E = \{(a, b), (a, c), (b, c), (s, a), (s, b)\}$, with weights as shown below. Each of the single-source shortest paths algorithms below repeatedly calls the `relax(..., u, v)` function below on various edges (u, v).



```python
def relax(Adj, w, d, parent, u, v):
    # relax edge (u, v)
    if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w(u, v)
        parent[v] = u
```

For each algorithm below, give the sequence of edges (u, v) on which `relax(..., u, v)` gets called, for some valid execution of the algorithm from source vertex s. Note that an algorithm might call `relax(..., u, v)` on the same edge (u, v) more than once, and a call to `relax(..., u, v)` **might not** decrease the value of `d[v]`.

(a) [4 points] DAG Relaxation

Solution: {(s,a), (s,b)}, {(a,b), (a,c)}, (b,c)

Common Mistakes: Relaxing edges from vertices in DFS order, instead of topological order.

(b) [4 points] Dijkstra

Solution: {(s, a), (s,b)}, (b,c), {(a,b), (a,c)}

Common Mistakes: Not relaxing all edges.

(c) [4 points] Bellman-Ford

Solution:

{(s, a), (s,b), (b,c), (a,b), (a,c)},
{(s, a), (s,b), (b,c), (a,b), (a,c)},
{(s, a), (s,b), (b,c), (a,b), (a,c)}

Common Mistakes: Doing fewer than $|V| - 1 = 3$ rounds of relaxation.

Problem 2. [8 points] **Negative-Weight Cycles**

Given a weighted directed graph, where **every edge** in the graph has **negative weight**, describe an efficient[1] algorithm to determine whether the graph contains a negative-weight cycle.

Solution: Finding a negative-weight cycle is equivalent to finding any cycle in this graph. Run Full-DFS, keeping track of the stack of ancestors visited on the search in a hash table. If the search ever revisits an ancestor stored in the hash table, DFS just traversed a back edge, confirming existence of a cycle. DFS runs in linear time, while hash table updates and lookups take constant time per vertex. Thus this algorithm can detect a negative-weight cycle in linear time.

Common Mistakes:

- In a DFS, encountering an already-visited node doesn't imply a cycle in a *directed* graph. This edge may be a cross edge, not a back edge.
- Bellman-Ford is not wrong, but is slower than necessary.
- Single-Source-DFS may leave part of the graph unexplored, so we need Full-DFS.
- Our topological-order algorithm doesn't throw an error in the presence of cycles. It just returns the node reverse finishing times of DFS. When the graph contains cycles, the returned ordering F won't be a topological order, as in Pset problem 7-1 (cdkx). To test for directed cycles (isn't a DAG) we can check for any edge whose source vertex comes after its destination vertex in F.
- Checking if there are more than $|V| - 1$ edges does not work: a graph with fewer edges can have a cycle, and a *directed* graph with more edges may not have a cycle.

[1] By "efficient", we mean that faster correct algorithms will receive more points than slower ones.

Problem 3. [8 points] **Directed Graph Radius**

Given a weighted graph $G = (V, E, w)$, recall that the **radius** $r(a)$ of a vertex a is the largest weight of any minimum-weight path from a to any other vertex, i.e. $r(a) = \max\{\delta(a, b) \mid b \in V\}$; and the **graph radius** $R(G)$ of G is the smallest radius of any vertex, i.e. $R(G) = \min\{r(a) \mid a \in V\}$. Describe an $O(|V|^3)$-time algorithm to compute the graph radius of any **directed** weighted graph having **possibly negative** edge weights, but no negative-weight cycle.

Solution: This problem can be solved by computing all-pairs shortest-path weights $\delta(u, v)$ for all $u, v \in V$. Johnson's algorithm can compute these values in $O(|V|^2 \log |V| + |V||E|) = O(|V|^3)$ time, because $E = O(|V|^2)$. For each $a \in V$, we can then compute $r(a)$ (the maximum of $|V|$ values) in $O(|V|)$ time, for a total of $O(|V|^2)$ time. Finally, we can compute $R(G)$ (the minimum of $|V|$ values) in $O(|V|)$ time, for a total of $O(|V|^3)$ time.

Common Mistakes:

- Using $|V|$ rounds of Bellman-Ford is slower than Johnson's (which is the point of Johnson's)
- Johnson's does create a graph with nonnegative weights, but shortest path weights in the new graph are different than what you want for radius.
- We cannot assume $|E| = O(|V|)$.
- Absence of negative-weight cycles does not imply DAG.
- Radius isn't looking for *longest* paths; it's looking for the longest *shortest* path. So we shouldn't negate edge weights to find longest paths.
- Using Floyd-Warshall is not wrong, but we haven't discussed it. We did allow it. It's never better that Johnson asymptotically, but it's good enough for this problem.

Problem 4. [15 points] **Future MIT**

In the future, MIT has taken over all of Cambridge. The campus now consists of b buildings and t underground indoor tunnels, with each tunnel connecting a pair of buildings. In addition to the underground tunnel network, every pair of buildings is connected via an outside route. Describe an efficient[1] algorithm to determine the minimum number of times you must go outside in order to visit the inside of every building at least once, starting inside building 6006.

Solution: Construct a graph G where future MIT buildings are vertices, with an undirected edge from building a to building b if there is an indoor tunnel between them. Once inside a building b, it is free to go to any other building connected to b via a sequence of tunnels. You only need to go outside when two buildings are not connected by tunnels, i.e. are in different connected components. Thus we can use Full-DFS or Full-BFS to count connected components of graph G in $O(t + b)$ time, while the number of times you must go outside will be one less than the number of connected components (since you start inside).

Common Mistakes:

- Incorrect graph construction, e.g., including outdoor paths so they're indistinguishable from indoor ones
- Thinking this was a shortest paths problem.
- Inefficient graph construction, e.g., b-fold graph duplication (one level per outdoor path taken)
- Using o or O as a variable for "outdoor edges". This is not wrong, but is quite challenging to distinguish from 0 (zero), especially in handwriting. :(

[1] By "efficient", we mean that faster correct algorithms will receive more points than slower ones.

Problem 5. [15 points] **Gone to the Dogs**

Most residents in Barkindog Town love dogs. In fact, by law, every stretch of road between intersections must be home to at least one dog (though many roads contain more than one). Resident Catnip Everdeen hates dogs. She wants to walk along roads from her home to the pet store (both located at road intersections) while minimizing canine interaction. Catnip has a map of all roads in Barkindog Town, marked with how many dogs live along each stretch of road. Given that k dogs live in Barkingdog Town, describe an $O(k)$-time algorithm to find a route from Catnip's house to the pet store that minimizes the number of dogs passed along the way.

Solution: Construct a graph G with a vertex for each intersection, with a chain of x unweighted edges from intersection a to intersection b if there is a stretch of road from a to b containing $x \geq 1$ dogs (since each road contains at least one dog). Graph G is an unweighted graph on k edges (one for each dog) and $O(k)$ vertices (each vertex is connected to at least one edge, as an intersection must be attached to a road). Then breadth-first search can find a path traversing the fewest edges (passing the fewest dogs) from Catnip's home to the pet store in $O(k + k) = O(k)$ time.

Common Mistakes:

- The graph is not a DAG.
- Dijkstra works but is slower. Many students claimed $O(V \log V + E) = O(k)$, which is false. Dijkstra with a different priority queue implementation (instead of Fibonacci Heap, use a DAA with a careful amortized analysis) can get $O(k)$, but no student tried to argue this method.

Problem 6. [15 points] **Troll Tolls**

Balbo Biggins is in a mountaintop metropolis consisting of m mountain villages connected by b two-way rope bridges, where each bridge is patrolled by a troll. In order to cross a bridge, Balbo must pay a **toll**: some **positive integer number** of gold pieces, where each bridge troll may demand a different toll. Whenever Balbo arrives at a mountain village via a bridge, a village elder welcomes him with a gift of a single gold piece (even if he has been there before). Balbo has a map listing each mountain village and its corresponding height, and each bridge and its corresponding toll. Balbo wants to reach the unique highest village from his starting location, and starts with a fixed amount of gold, large enough to reach any mountain village. Describe an efficient[1] algorithm to determine the maximum amount of gold Balbo can have when entering the highest village.

Solution: Construct a graph G where mountain villages are vertices, with an undirected edge with weight $t - 1$ from village a to village b if there is a troll bridge between a and b requiring a toll of t gold pieces. We subtract one from the toll as entering a village will yield Balbo one more piece of gold. As each toll is a positive integer, subtracting one from t remains non-negative. Use Dijkstra to find the weight of a shortest path from Balbo's starting location to the highest village. Graph G has m vertices and b edges. The highest village can be found in $O(m)$ time by looking at each village, and Dijkstra will run in $O(m \log m + b)$ time (assuming the use of a Fibonacci heap for Dijkstra's priority queue).

Common Mistakes:

- Not accounting for gold received
- Balbo's path need not be monotone in altitude, so a DAG by height isn't sufficient
- Bellman-Ford is not wrong, but is slower than necessary
- Trying to modify Dijkstra relaxation without rearguing correctness
- Runtime analysis should be given in terms of the given variables, m and b.

[1] By "efficient", we mean that faster correct algorithms will receive more points than slower ones.

Problem 7. [15 points] **Muscle Mono-Toning**

Bain Usolt wants to do a running workout on his morning commute to work. He has measured the length of all r stretches of road in the city and the **unique** elevation of each of the city's n intersections, including the locations of his home and office. His running route follows a revolutionary muscle-toning technique called **Mono-Toning**: starting at home, he wants to only increase his elevation at each successive road intersection until he stops for a water break at some intersection. After the break, he wants to only decrease his elevation at each successive intersection all the way to his office. Describe an efficient[1] algorithm to help Bain find a Mono-Toning route from home to work that **maximizes** the length of his run.

Solution: Construct a graph G where each road intersection x corresponds to two vertices x_1 and x_2, and for each road between intersections a and b with length w, there is a directed edge with weight $-w$: from a_1 to b_1 if the elevation of a is smaller than that of b; and from a_2 to b_2 if the elevation of b is smaller than that of a. Further, for each intersection x, add a directed edge from x_1 to x_2 with zero weight. Graph G has $2n$ vertices and $n + r$ edges, and G is acyclic because of the elevation restriction. Let s and t be the intersections corresponding respectively to his home and office. Then a shortest path from s_1 to t_2 will be a Mono-Toning route that maximizes the length of Bain's run. So use DAG Relaxation to find such a shortest path in $O(n + r)$ time.

Common Mistakes:

- Bellman-Ford to find longest paths (e.g., by negating edge weights) is not wrong, but is slower than necessary because the graph is a DAG (if set up correctly).
- Dijkstra cannot be used to find longest paths. Negating edge weights would leave *negative* weights, which Dijkstra can't handle.
- Some students ran SSSP from all possible water-break locations, which amounts to solving APSP. This is more (and slower) than necessary, since there are only two sources we care about: home and office.
- When running SSSP from the office, you want the graph of *uphill* edges, not *downhill* edges, so you can gain elevation from the office to the water break. The final route will run this path backwards.

[1]By "efficient", we mean that faster correct algorithms will receive more points than slower ones.

Problem 8. [15 points] **Super Lario World**

In a classic video game, Lario Muigi must traverse g game levels. At the end of each level are at most p pipes; Lario can choose to enter any one pipe which will transport him deterministically to the start of some other level. One pipe is **winning**: if Lario enters that pipe, he will win the game! Each level contains a positive number of **coins** that Lario can collect. In addition, some levels contain either a mushroom or a boss, but not both. Eating a **mushroom** will make Lario big (Lario may be either big or small). In a **boss** level, Lario must fight the boss which will decrease his size from big to small and cause Lario to lose 100 coins (Lario is allowed a negative number of coins); Lario will **lose the game** if he tries to fight a boss while small. You know which pipes connect to which levels, and the locations of all coins, mushrooms, and bosses. Re-entering a level will regenerate everything in that level. Assuming Lario begins the game with no coins and small size, describe an efficient[1] algorithm to determine the maximum number of coins (possibly negative) that Lario can have when he enters the winning pipe.

Solution: Define level t to be the state after entering the winning pipe, and let s be the starting level. Construct a graph G in the following way. For each level v add two vertices v_S and v_B to G, corresponding to Lario entering level v as small or big respectively. Then for each pipe in level a which can transport Lario to level b, add either one or two edges to G. If level a does not contain a mushroom or boss, add directed edges (a_S, b_S) and (a_B, b_B) weighted by the negative of the number of coins in level a. If level a contains a mushroom, add directed edges (a_S, b_B) and (a_B, b_B) (as it is always better to eat the mushroom and be big), again weighted by the negative of coins in level a. If level a contains a boss, add directed edge (a_B, b_S) weighted by 100 minus the coins in level a. Graph G has $2(g+1)$ vertices and at most $2(g+1)p$ edges, might contain negative weights, and might contain cycles. Then the minimum weight of a shortest path from s_S to either t_S or t_B will be a winning gameplay that will maximize the number of coins. So use Bellman-Ford to compute this maximum in $O(g^2 p)$ time; in particular, if a negative cycle in G can be reached along a path from s_S to either t_S or t_B, Bellman-Ford can return that the maximum will be infinite.

Common Mistakes:

- There are up to p pipes *per room*, so $|E| = O(pg)$, not $O(p)$.
- Runtime analysis should be given in terms of the given variables, p and g.
- Not a DAG and can have negative weight edges, so DAG relaxation and Dijkstra don't apply.
- There can be levels without mushrooms or bosses.
- Boss levels might have coins, so the edge weight is $100 -$ (number of coins), not just 100.

[1]By "efficient", we mean that faster correct algorithms will receive more points than slower ones.

Problem 9. [15 points] **Dealer's Choice**

Annie Doeshun plays a card game against a single opponent using a deck of $2n$ cards, where each card has a positive integer value. At the beginning of the game, Annie deals n cards to each player. After the deal, Annie's **advantage** will be the total value of cards dealt to her, minus the total value of cards dealt to her opponent. Annie has some flexibility when dealing cards: she first deals either **one** or **two** cards to her opponent from the top of the deck, immediately followed by dealing **the same number** of cards to herself. She repeats this procedure (dealing both players either one or two cards, opponent first) until all cards are evenly dealt. Given the values of the $2n$ cards in their deck order at the start of the deal, describe an $O(n)$-time **dynamic programming** algorithm to determine the largest advantage Annie can have at the end of the deal.

Solution:

1. Subproblems
   - Let the card values be $C_1, C_2, \ldots, C_{2n}$ from top to the bottom
   - Let $x(2i)$ be the maximum advantage possible after dealing a total of $2i$ cards from the top of the deck, i to each player, in the specified manner
2. Relate
   - Either Annie deals her opponent one card or two (Guess!)
   - If she deals one, her opponent gets the next card, and she gets the one after
   - If she deals two, her opponent gets the next two cards, and she gets the two after
   - $x(2i) = \max\{x(2i-2) - C_{2i-1} + C_{2i},\ x(2i-4) - (C_{2i-3} + C_{2i-2}) + (C_{2i-1} + C_{2i})\}$
   - Subproblems $x(2i)$ only depend on strictly smaller i, so acyclic
3. Base
   - $x(0) = 0$ (nothing dealt yet)
   - $x(2) = -C_0 + C_1$ (two cards dealt)
4. Solution
   - Solve subproblems $x(2i)$ in order of increasing i for $2 \leq i \leq n$
   - Solution to problem is subproblem $x(2n)$, maximizing the entire $2n$-card deck
5. Time
   - # Subproblems is $n + 1$
   - Work per subproblem is $O(1)$
   - Total running time is $O(n)$

Alternative (sketch): Another approach uses deck *suffixes* instead of prefixes: define subproblem $y(2i)$ as the greatest possible advantage from a balanced deal on *only* the last $2i$ cards. Then $y(2i)$ can be computed in terms of $y(2i+2)$ and $y(2i+4)$, as opposed to $x(2i)$ above being computed from $x(2i-2)$ and $x(2i-4)$.

Continued on S1...

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.

Common Mistakes:

- The subproblems “maximum after dealing $2i$ cards” and “maximum after dealing i cards to each player” are different! E.g., after dealing 4 cards, Annie might have 3 of them. Many students meant the latter but wrote the former.
- Many students described the “deck prefix” strategy $x(2i)$ in words but wrote formulas for the “deck suffix” strategy $y(2i)$, or vice-versa. Make sure your “Subproblem” and “Relate” steps agree with each other.
- The recurrence above for $x(2i)$ relies on $x(2i-2)$ and $x(2i-4)$, so it only applies to $x(4)$ and higher. This means $x(0)$ **and $\boldsymbol{x(2)}$** are both base cases. Setting $x(i) = -\infty$ for $i < 0$ is an alternatively acceptable base case.
- Make sure you specify what the final return value is! In this case, it is $x(2n)$.

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.

SCRATCH PAPER 4. 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 S4" on the problem statement's page.

SCRATCH PAPER 5. 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 S5" on the problem statement's page.