Quiz 2 Solutions

- 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. Circle your recitation at the bottom of this page.
- You have 120 minutes to earn a maximum of 120 points. Do not spend too much time on any one problem. Read 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 sheets with your own notes**. No calculators, cell phones, or other programmable or communication devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the scratch pages at the end of the exam, and refer to the scratch pages in the solution space provided. Pages will be scanned and separated for grading.
- Do not waste time and paper rederiving facts that we have studied. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required. But be sure to prove the required bound on **running time** and explain **correctness**. Even if your running time is slower than the requested bound, you will likely receive partial credit if your algorithm and analysis 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	Grade	Grader
0: Name	0	2		
1: True or False, and Justify	12	48		
2: Two Weights, One Edge	1	10		
3: Red and Blue States	1	10		
4: free-fruit@mit.edu	4	15		
5: Algorithmic Adventure Time!	3	20		
6: Double Negative	1	15		
Total		120		

Name: _____

Circle your recitation:	R01 Skanda Koppula	R02 Ludwig Schmidt	R03 Parker Zhao	R04 Gregory Hui	R05 Yonadav Shavit	R06 Atalay Ileri	R07 Anton Anastasov
	R08 Akshay Ravikumar	R09 Aradhana Sinha	R10 Ray Hua Wu	R11 Daniel Zuo	R12 Themistoklis Gouleakis	R13 Adam Hesterberg	R14 Ali Vakilian

Problem 0. [2 points] **What is Your Name?** (2 parts)

(a) [1 point] Flip back to the cover page. Write your name and circle your recitation section.

(b) [1 point] Write your name on top of each page, including the scratch paper!

Problem 1. [48 points] **True or False, and Justify** (12 parts)

Circle T or F to indicate whether the statement is true or false, and explain your answer: if true, sketch a proof; and if false, give a counterexample. Your justification is worth more than your true or false designation.

(a) T F [4 points] Consider a hash table storing n items in m slots, with collisions resolved by chaining. Assume simple uniform hashing. Then SEARCH has expected running time $O(1)$.

Solution: False. Search requires finding the position of the element in the hash table ($O(1)$), and then looking through the length of the chain at that location (under uniform hashing, expected $O(\alpha) = O(\frac{n}{m})$. This yields an overall runtime for search of $O(1 + \frac{n}{m})$. The previous expression only reduces to $O(1)$ if $n = \theta(m)$. For example, if $n = \Theta(m^2)$, then Search is expected-time $O(n)$.

(b) T F [4 points] Consider a hash table storing n keys in m slots, with resolutions resolved by chaining, and hash function

$$h(k) = ((k^2 + 3) \mod 8) \mod m.$$

Then DELETE has expected running time $\Theta(n/m)$.

Solution: False. Because we're taking $\mod 8$, there are a constant number of possible hash values (in particular, there are 3 possible values of $k^2 + 3 \mod 8$). For this reason, the expected number of values in each slot will be $n/3$, which makes the runtime of DELETE $\Theta(n)$.

(c) T F [4 points] Consider an empty ($n = 0$) hash table with $m = 4$ initial slots. Assume that the table maintains a load factor ($\alpha = n/m$) of at most 2 via table doubling (i.e., doubling the table size whenever the load factor is exceeded). Then, after inserting 28 keys into the table (and not deleting any), the hash table must have $m \geq 32$.

Solution: False. People occasionally had the right idea, and marked the wrong true/fale. Arithmetic errors were also common.

(d) T F [4 points] Suppose we dynamically resize a hash table by the following rules:

(i) If the table becomes full (you already have m items, and you try to insert one more), the table doubles in size: $m \to 2m$.

(ii) If the table becomes less than half full ($< \frac{m}{2}$ elements), the table halves in size: $m \to \frac{m}{2}$.

Then the amortized time complexity for resizing the table when adding or removing an element is $O(1)$.

Solution: False. Consider a table with size m. Adding an element leads to a table doubling, which takes $O(m)$. Now, deleting two elements leads to a table size of $m - 1 < (2m)/2$, which leads to table shrinking. If we continue adding and removing two elements, we end up taking an average of $O(m)$ per operation, which is larger than $O(1)$.

Common mistakes included reverting to the old doubling rules showin in class.

(e) T F [4 points] In every directed acyclic graph, BFS and DFS visit the vertices in the same order.

Solution: False. Consider any tree (other than line or star) – BFS will visit these in level order, while DFS will probe between levels.

(f) T F [4 points] Recall that a ***simple path*** is a path that visits no vertex more than once. In any (unweighted) undirected graph $G = (V, E)$, the number of simple paths between two vertices is at most $2^{|V|}$.

Solution: False. The number of simple paths can be much higher. Consider a complete graph with n vertices (i.e. any vertex is connected to every other vertex). Then, a path between given vertices u and v can visit any number of vertices in between, in any order. There are $\binom{n}{k}$ paths of length k, and there are $k!$ ways to order each path. Summing this over all values of k results in a number much higher than $2^{|V|}$. In particular, $2^{|V|}$ corresponds with picking which intermediate vertices are on the path, irrespective of order. Some students just considered paths of length $|V|$, for which there are $(|V| - 2)!$ ways to order each path. As long as these students did not mistakenly say that the total number of paths is $(|V| - 2)!$, ignoring the paths of smaller length, this received full credit, because $(|V| - 2)! > 2^{|V|}$ as $|V|$ grows to infinity. Other students mixed out the number of different paths, and the ways to order each path, receiving partial credit.

(g) T F [4 points] Consider the forest (vertex-disjoint set of trees) returned by a complete run of DFS on a **directed** graph. The number of trees in this forest will always be the same, regardless of the order in which you visit nodes in the outermost loop of DFS.

Solution: False. There are many possible counterexamples. Consider a line of n vertices $v_1 \to v_2 \to v_3 \cdots \to v_n$. Running DFS in the order $v_1, v_2, \ldots v_n$ results in one tree, while running DFS in the reverse order results in n different trees.

(h) T F [4 points] Consider the forest (vertex-disjoint set of trees) returned by a complete run of DFS on an **undirected** graph. The number of trees in this forest will always be the same, regardless of the order in which you visit nodes in the outermost loop of DFS.

Solution: True. In an undirected graph, there is always a separate tree for each connected component. This number is the same, regardless of the order in which DFS visits vertices.

(i) T F [4 points] When solving a single-source shortest-paths problem on a directed graph where all edges have weight 1 or 2, Dijkstra's algorithm will explore nodes in the same order that BFS explores them.

Solution: False. Consider a graph with vertices a, b, c, d, e, f, and g, edges (a, b), (b, c), (c, d), and (d, g) of weight 1, and edges (a, e), (e, f), and (f, g) of weight 2. Dijkstra's and BFS will explore d and f in opposite orders.

(j) T F [4 points] Consider a connected, undirected graph with more edges than vertices and only negative edge weights. Then, for any start node, Bellman–Ford necessarily detects a negative-weight cycle.

Solution: True. If there are more edges than vertices, there must be a cycle in the graph (this is not true for directed graphs). Because all edge weights are negative, there must be a negative cycle. Because the graph is connected, Bellman–Ford will detect this cycle no matter where it starts in the graph.

(k) T F [4 points] Recall that Johnson's algorithm reweights edges by creating a new node s, connecting s to every other vertex v with an edge (s, v) of weight 0, and running Bellman–Ford from s. The algorithm would still work if, instead, we picked an arbitrary existing node u, and added an edge (u, v) of weight 0 to every other vertex v for which edge (u, v) did not already exist.

Solution: False. If the selected node u was part of a negative weight cycle in the original graph, then running Bellman-Ford from u will not be able to find finite node potentials for the Johnson's transformation, and thus we will not be able to run Dijkstra's from every node. Unfortunately, the wording of the statement makes it unclear whether or not the added 0 weight edges are removed from the graph once the node potentials are calculated, so answers which stated that adding 0 weight edges does not preserve the original graph's shortest paths were awarded full credit as well.

(l) T F [4 points] Recall that the Floyd–Warshall algorithm involves three nested *for* loops, iterating through intermediate, start, and destination vertices, respectively. If we remove the middle loop and just use a fixed start vertex, then we obtain an $O(V^2)$-time single-source shortest-paths algorithm.

Solution: False. If we fix the start vertex, then we will never update paths starting from an intermediate vertex. For this reason, we won't find the shortest paths.

Problem 2. [10 points] **Two Weights, One Edge** (1 part)

Given an edge-weighted undirected graph $G = (V, E, w)$ where every edge e has an integer weight $w(e)$ equal to 0 or 1, design an algorithm to solve the single-source shortest-paths problem in $O(V + E)$ time.

Solution:

Because every edge $e \in E$ has nonnegative weight, we can use Dijkstra's algorithm to solve this single-source shortest-paths problem in $O(V \lg V + E)$-time. Unfortunately, this is not as efficient as we want. There are two ways to achieve the desired time bound.

The simplest solution is a modified form of breadth-first search, which alternates between advancing along paths of weight-0 edges and single weight-1 edges. Let G_0 denote the subgraph of G with only the weight-0 edges, and let G_1 denote the subgraph of G with only the weight-1 edges. We start with $\{s\}$ in our frontier, and with a G_0 round. In a G_0 round, we run DFS in G_0 starting from all nodes in the frontier, without repeating any nodes visited in any DFSs from the frontier (i.e., without resetting the parent pointers). However, we are in the right direction. We just need one crucial observation. In a G_1 round, we run a single round of BFS, advancing the frontier to the nodes reachable by one edge in G_1 from the previous frontier.

Another solution is based on modifying Dijkstra. Note that at any point throughout the execution of Dijkstra's algorithm on this particular graph G, the priority queue we maintain will contain **at most 3** different path weights: k, $k + 1$ and ∞. At initialization, all keys are ∞ except for $s.key$, which is 0. Then, after we consider all neighbours $Adj[s]$ of s, the priority queue will contain keys among the set $\{0, 1, \infty\}$, because every edge e has an integer weight $w(e)$ equal to either 0, or 1. The property that all keys in the priority queue are among $\{k, k + 1, \infty\}$ for some $k \in \mathbb{N}$ is an invariant throughout the execution of the algorithm.

With the above observation in mind, we can improve the runtime of Dijkstra's algorithm on this specific graph G to $O(V + E)$ by substituting the priority queue with one that only maintains two different possible keys: k and $k + 1$ (we do not need to maintain the ∞ keys explicitly). To that end, there are at least two different options: use $|V|$ queues, or use a deque.

- Any shortest path p in G will have $w(p) \leq |V| - 1$ because all edges have weights of either 0 or 1. With that in mind, instead of using a Fibonacci heap for a priority queue, we can use a list of $|V|$ queues where each queue Q_k for $k = 0, 1, \ldots, |V|-1$ will only store keys equal to k. With this idea in mind, we can implement all priority queue operations necessary for Dijkstra's algorithm in $O(1)$ worst case, thus leading to a $O(V + E)$-time algorith overall.
- Alternatively, we can use double-ended queue, usually called a *deque*, that allows for consuming or adding elements at either of its ends in constant time. Updating the estimate for the shortest path to a given node results in adding a new element in the deque: if the last edge traversed has 0 weight, then it suffices to add the new key to the front of the deque; otherwise, we add the new key to the back of the deque. EXTRACT-MIN always consumes the element from the front of the deque. Finally, using this data structure results in a $O(V + E)$-time algorithm for the problem at hand.

Common Mistake: The most common mistake was to substitute every edge of weight 1 with two edges of weight 0, and subsequently make the graph unweighted, which allows for running BFS

in $O(V + E)$ time. However, this transformation does not preserve shortest paths. In particular, consider two paths – one with one edge of weight 1, and one with 3 edges of weight 0. Under the transformation, the path with one edge is transformed into a path of two edges, whereas the path of 3 edges of weight 0 is transformed into a path of 3 edges. Therefore, we see that under the transformation that path of weight 1 is shorter than the path of weight 0, which shows that the transformation does not preserve shortest paths.

Problem 3. [10 points] **Red and Blue States**[1] (1 part)

Let $G = (V, E)$ be an (unweighted) undirected graph. We want to color each vertex either red or blue in such a way that no vertices with the same color are connected by an edge. Design an $O(V + E)$-time algorithm that determines whether G has such a coloring.

Solution: Run BFS (DFS with proper modification also works) and color the layers alternatively with blue and red. Moreover, once you visit a node check that it does not participate in any mono-choromatic edge (it would be over already colored edges only). If there exists a mono-choromatic edge at any point of the run of algorithm return FALSE; otherwise return the coloring.

Common Mistake: Assuming that a coloring of nodes is given and your task is to check whether it is feasible or not!

[1] Too soon?

Problem 4. [15 points] **`free-fruit@mit.edu`** (4 parts)

Whole Shaw's Star Market Basket offers several different varieties of fruit, given by the set F. On day $i \in \{0, 1, \ldots, n-1\}$, their dumpster contains a subset $F_i \subseteq F$ of fruits (which you can predict via stock analysis). Unfortunately, your band of dumpster divers will let you eat **exactly one** fruit per day (and you cannot save fruit for later days, as it would go bad).

Each fruit $f \in F$ contains a number $c(f)$ of calories, and your goal is to maximize the number of calories you consume. The catch is that, in every window of three consecutive days, you must consume a total of between $c_{\min}$ and $c_{\max}$ calories (so that you do not die of starvation or overconsumption).

You and your freegan 6.006 TA decide to use dynamic programming to solve this problem—choosing exactly one fruit $f_i \in F_i$ for each day i to maximize total calorie consumption $\sum_i c(f_i)$ while satisfying the three-day-window min/max constraints.

(a) [3 points] Your TA defined the following subproblems: $T(i, d_2, d_1)$ is the maximum number of calories you can consume from the ith day through the final day, supposing that you ate fruit $d_2 \in F$ two days ago and you ate fruit $d_1 \in F$ one day ago (using special value None when we are in the first one or two days).

How many subproblems are there, as a function of n and $|F|$?

Solution: $\Theta(nF^2)$ subproblems

(b) [2 points] Supposing you have solutions to all subproblems $T(i, d_2, d_1)$, how can you solve the original problem?

Solution: $T(0, \text{None}, \text{None})$.

(c) [7 points] Your TA wrote on a napkin the following recurrence for solving $T(i, d_2, d_1)$, but he failed to fill in some blanks (drawn here with boxes):

$$T(i, d_2, d_1) = \max\{\ \boxed{A} + T(\boxed{B}, \boxed{C}, \boxed{D}) \mid f \in F_i, \text{ where } \boxed{E} \text{ holds } \}$$

where $\max$ is defined to return $-\infty$ (meaning "impossible to survive") if the given set is empty.

Unfortunately, your TA is away on vacation, so you need to fill in the blanks yourself. Provide the expressions for $\boxed{A}$, $\boxed{B}$, $\boxed{C}$, and $\boxed{D}$ in terms of f and i. Also describe the condition $\boxed{E}$ to check your calorie consumption bounds.

Solution:
$\boxed{A}$ = $c(f)$
$\boxed{B}$ = $i + 1$
$\boxed{C}$ = d_1
$\boxed{D}$ = f
$\boxed{E}$ = "$c_{\min} \leq c(f) + c(d_1) + c(d_2) \leq c_{\max}$"

(d) [3 points] What is the running time of the resulting dynamic program, as a function of n and $|F|$? Assume that, for any $f \in F$, $c(f)$ can be computed in constant time.

Solution: $\Theta(nF^2)$ subproblems, each costing $\Theta(F)$, for a total of $\Theta(nF^3)$.

Problem 5. [20 points] **Algorithmic Adventure Time!** (3 parts)

Finn and Jake are trying to travel through the wonderful country of Dijkstan, which consists of ***cities*** $V = \{v_1, v_2, \ldots, v_{|V|}\}$ and one-way ***flights*** $E = \{e_1, e_2, \ldots, e_{|E|}\}$ connecting ordered pairs of cities. Each flight $e \in E$ has a positive integer ***time*** $t(e)$ and positive integer ***cost*** $c(e)$ required to traverse it (measured in minutes and dollars, respectively).

You may assume that there exists a path between every pair of cities, and that there is at most one flight between any ordered pair of cities.

Answer the following questions to help Finn and Jake plan our their schedules. In the following parts, partial credit will be awarded for correct solutions, but the amount of credit depends on the efficiency of your algorithm. **Clearly specify your running time.**

(a) [5 points] Finn wants to travel from city s to city t using the path p that minimizes $a \cdot t(p) + b \cdot c(p)$, where $t(p)$ and $c(p)$ are the total time and cost of the edges (flights) along path p, respectively, and a and b are positive real numbers. Design an $O(V \log V + E)$-time algorithm to compute the optimal such route.

Solution: Weight each edge e by $w(e) = a \cdot t(e) + b \cdot c(e)$. Because every edge weight is positive, we can run Dijkstra's algorithm to find the shortest path from s to t, which runs in $O(V \log V + E)$.

(b) [10 points] Jake wants to travel from city s to city t using the minimum time, subject to not spending more than his positive integer budget of B dollars, Design an $O(BV \log(BV) + BE)$-time algorithm to compute the optimal such route.

Solution: Create $B + 1$ different graphs, each one representing reachable states that cost $0, 1, 2, \ldots B$ dollars. If v was a vertex in the original graph, we now have $B + 1$ vertices $(v, 0), (v, 1), \ldots (v, B)$. Now, for edge $e = (u, v)$ in the original graph, draw an edge from (u, x) to $(u, x + c(e))$ for $0 \leq x + c(e) \leq B$, and let each of these edges have weight $t(e)$. To account for the case where we use less than B dollars, draw a weight-0 edge from (t, x) to $(t, x + 1)$ for $0 \leq x \leq B - 1$. Now, we can run Dijkstra from $(s, 0)$ to (t, B). Because our new graph has BV vertices and $\leq BE$ edges, the runtime of our algorithm is $O(BV \log BV + BE)$.

(c) [5 points] Princess Bubblegum is considering removing some flights from Dijkstan. Call a flight $e \in E$ ***useless*** if e is not used by any optimal path, according to Finn's objective from part (a), between any pair (s, t) of cities. Design an $O(V^2 \log V + VE)$-time algorithm to find all useless flights.

Solution: First, weight the edges as in part (a), and compute all-pairs shortest paths by running Dijkstra with every vertex as source. (Alternatively, run Johnson's algorithm, but reweighting is unnecessary here because all edge weights are positive.) The running time so far is $O(V^2 \log V + VE)$.

Second, to compute the useless edges from this information, there are two types of solutions we accepted as correct and efficient:

1. For every flight (u, v), check whether the shortest-path weight $\delta(u, v)$ is strictly smaller than the edge weight $w(e)$. If so, the flight (u, v) is useless: any shortest path would never use the edge (u, v), instead opting for the shortest path from u to v which, being strictly shorter, does not include the edge (u, v). Otherwise, $\delta(u, v) = w(u, v)$, so the flight (u, v) is on a shortest, for example, a one-edge shortest path from u to v. Finding all useless flights in this way costs $O(E)$ time.
2. For each source s, mark the flights that form the shortest-path tree from s, i.e., $\{(parent(v), v) : v \in V\}$ where $parent$ is one of the arrays returned by Dijkstra run from source s. (Alternatively, many students modified Dijkstra's algorithm to mark the flight $(parent(v), v)$ when v was extracted from the priority queue, and thus $d[v]$ had reached its final value. This is correct, but unnecessarily complicated; it is easier to use Dijkstra as a black box.) Because the shortest-path tree contains all the edges in the shortest paths from s to all vertices v, a flight e is useless if and only if it is unmarked after running all sources s. Finding all useless flights in this way costs $O(VE)$ time.

(In fact, the two solutions return different solutions when shortest paths are not unique. The first solution gives the intended answer: useless flights are not used by *any* shortest path between any vertex pair (s, t). The second solution solves a slightly different problem: a flight e is useless if, for every vertex pair (s, t), there is *some* shortest path that does not use e. The latter may mark more edges as useless than the former. But we did not penalize this difference in grading.)

The most common inefficient solution was to list all shortest paths from each source s (by, for each vertex v, following the parent pointers to reconstruct the shortest path from s to v), and mark the edges in these paths; and at the end, return all unmarked edges. This algorithm returns the same results as the second solution above, but fails to exploit that shortest paths from s tend to share long prefixes in the form of the shortest-path tree. The running time is $O(V^3)$—$|V|$ for the loop over s, times $|V|$ for the loop over v, times $|V|$ for enumerating the edges in the shortest path from s to v—which can be a factor of $\Theta(V/\log V)$ worse than the requested time bound.

Problem 6. [15 points] **Double Negative** (1 part)

Suppose you are given an edge-weighted directed graph $G = (V, E, w)$ which has exactly **two** negative-weight edges $e_1, e_2 \in E$. Design an $O(V \log V + E)$-time algorithm to detect whether G has a negative-weight cycle.

Partial credit will be awarded for answers that solve the problem for exactly one negative edge.

Solution: Let (u_1, v_1) and (u_2, v_2) be the negative weight edges. If there exists a negative cycle C in G, then at least one of (u_1, v_1) or (u_2, v_2) is in C. Consider the cycles $\mathcal{C}_1$ that only contains (u_1, v_1). For $C \in \mathcal{C}_1$, $w(C) = w(u_1, v_1) + w(P_{v_1,u_1})$ where P_{v_1,u_1} denotes a path from v_1 to u_1. Let $P^*_{s,t}$ be the shortest path from s to t that does not use (u_1, v_1) and (u_2, v_2). Then $w(C) \leq w(u_1, v_1) + w(P^*_{v_1,u_1})$. Similarly, let $\mathcal{C}_2$ denote all cycles that contain (u_2, v_2) but not (u_1, v_1). By the same argument, for all $C \in \mathcal{C}_2$, $w(C) \leq w(u_2v_2) + w(P^*_{v_2u_2})$.

The last case we need to consider is the set of cycles that contain both (u_1, v_1) and (u_2, v_2). For each such cycle C, we have $w(C) \leq w(u_1, v_1) + w(P^*_{v_1,u_2}) + w(u_2, v_2) + w(P^*_{v_2,u_1})$.

Thus, after computing $w(P^*_{v_1,u_1}), w(P^*_{v_2,u_2}), w(P^*_{v_1,u_2})$, and $w(P^*_{v_2,u_1})$ we can decide whether G has a negative weight cycle. Because $G^* = (V, E \setminus \{(u_1, v_1), (u_2, v_2)\})$ does not have any negative weight edges, we can perform Dijkstra algorithms twice (from v_1 and v_2) to get all the required shortest paths. The total runtime is $O(V \lg V + E)$.

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 refer to the scratch paper next to the question itself. Also be sure to write your name on the scratch paper!

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 refer to the scratch paper next to the question itself. Also be sure to write your name on the scratch paper!