Quiz 2 Solutions

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains 5 problems. You have 120 minutes to earn 115 points.
- This quiz booklet contains 13 pages, including this one, and a sheet of scratch paper.
- This quiz is closed book. You may use two double-sided letter (8½” × 11”) or A4 crib sheet. No calculators or programmable devices are permitted. Cell phones must be put away.
- Write your solutions in the space provided. If you run out of space, continue your answer on the back of the same sheet and make a notation on the front of the sheet.
- Do not waste time deriving facts that we have studied. Just cite results from class.
- When we ask you to “give an algorithm” in this quiz, describe your algorithm in English or pseudocode, and provide a short argument for correctness and running time. You do not need to provide a diagram or example unless it helps make your explanation clearer.
- Do not spend too much time on any one problem. Generally, a problem’s point value is an indication of how many minutes to spend on it.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Please be neat.
- Do not discuss the quiz with anyone until we give you permission to do so. We still have a handful of people taking make-up quizzes after tonight.
- Good luck!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Title</th>
<th>Points</th>
<th>Parts</th>
<th>Grade</th>
<th>Initials</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Name</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>True or False</td>
<td>24</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Short Answers</td>
<td>40</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4-Colorability</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Priority Queues</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Approximating Knapsack</td>
<td>30</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>115</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name: ____________________________

F10  F11  F11  F12  F12  F1  F1  F2  F2  F3  
R01  R02  R07  R03  R08  R04  R09  R05  R10  R06  
Nathan  Nathan  Adam  Bonny  Adam  Bonny  Casey  Hayden  Christina  Hayden
Problem 0. Name. [1 point] Write your name on every page of this exam booklet! Don’t forget the cover.

Problem 1. True or False. [24 points] (8 parts)
Circle T or F for each of the following statements to indicate whether the statement is true or false and briefly explain why.

(a) T F If a linear program has an optimal solution, then the optimal solution is unique.

Solution: False. There could be an optimal edge.

(b) T F Adding a constraint to a linear program always strictly decreases the size of the feasible region.

Solution: False. A simple counterexample is adding $x_1 \geq 2$ to a linear program that already has the constraint $x_1 \geq 4$.

(c) T F Consider Seidel’s algorithm presented in class for 2D Linear Programming which incrementally adds constraints one by one. Instead of adding the constraints in random order, suppose that a little birdy provides them in the order that minimizes the running time. Assume there is a unique optimal solution. Then the little birdy can ensure that the total running time of the algorithm is always linear.

Solution: True. The little birdy can start by providing the two constraints that define the optimum solution. Then all future constraints will be satisfied by that solution and no further updates will be needed.
(d) T F  Given a universal hash family $H$, every nonempty subset $G \subseteq H$ is also a universal hash family.

**Solution:** False. If $|G| = 1$, then the probability of collision among different elements with the same hash is 1. If $n > m$, then there must exist two different elements with the same hash by the Pigeonhole principle.

(e) T F  When achieving perfect hashing with $\Theta(n)$ space using a two-level hash table, the first and second level hash tables must both avoid collisions.

**Solution:** False. The first level can have collisions, but the second cannot.

(f) T F  Consider the family of hash functions $H = \{f, g, h\}$ where each function maps $\{0, 1, 2\} \rightarrow \{0, 1\}$. In particular:

$$f(0) = 1, f(1) = 0, f(2) = 0$$
$$g(0) = 0, g(1) = 1, g(2) = 0$$
$$h(0) = 0, h(1) = 0, h(2) = 1$$

Then $H$ is a universal hash family.

**Solution:** True. This just follows from the definition. For every $x \neq y \in 0, 1, 2$, the probability that $a(x) = a(y)$ over $a \in H$ is $1/3$, which is less than $1/m$ where $m = 2$. 
(g) T F Finding the maximum independent set on graphs that are binary trees is NP-complete.

Solution: False. We can solve this in polynomial time using a dynamic programming algorithm where each subproblem computes the maximum independent set on the subtree rooted at a particular node. This problem is exactly the same as homework problem 2.2 if we set each node’s value to 1.

(h) T F The LRU paging strategy is $2k$-competitive, where $k$ is the size of the cache.

Solution: True. LRU is k-competitive, and by definition, if it’s k-competitive, it’s also $2k$-competitive.
Problem 2. Short Answers. [40 points] (5 parts)

Please answer the following questions.

(a) [8 points] Consider the $k$-Coloring problem: Given a graph $G = (V, E)$, use at most $k$ colors to color the graph such that no two adjacent nodes get the same color. Assume that you are given a graph that is $k$-colorable. Provide a polytime algorithm that finds a coloring that uses at most $\frac{nk}{\log(n)}$ colors. Note that $k$ is a constant.

**Solution:** Modify the algorithm we used in class for a $\frac{n}{\log(n)}$ approximation of Clique. Take the set of vertices $V$, and partition them into $\frac{n}{\log(n)}$ groups. For each group, there is at most $\log(n)$ vertices. Simply brute force over all possible assignment of $k$ colors to this graph, and find a valid $k$-coloring. This takes at most $O(k^{\log(n)}) = O(nk)$ time for each group. We use a set of $k$ new colors for each group, and thus in total we use a maximum of $nk/\log(n)$ colors in this valid coloring.

Common mistake: The algorithm of “pick a node of high degree and $k$-color its neighbors” does not work since we don’t know how to $k$-color the neighbors in poly time. The algorithm we did in class only needed to 2-color, which we know how to solve in polytime.

(b) [8 points] Given a graph with maximum degree $d$, give a polynomial time algorithm to find an independent set of size at least $\frac{n}{(d + 1)}$.

**Solution:** Naively pick any vertex, put it in the independent set and remove it and its neighbors (which is at most $d + 1$). In each iteration, the algorithm removes at most $d + 1$ vertices, thus after choosing $k$ vertices, there are at least $n - k(d + 1)$ vertices left. Thus if $k$ is strictly less than $\frac{n}{(d + 1)}$, after removing any $k$ vertices, there will still be remaining vertices that are not adjacent to any of the vertices chosen in the independent set. Thus, this algorithm always gives an independent set of size at least $\frac{n}{(d + 1)}$. 
(c) [8 points] The MAXEKSAT problem is similar to KSAT, but attempts to maximize the total number of clauses which are satisfied. It also has the additional constraint that each clause has exactly $k$ distinct literals. Consider a randomized algorithm for MAXEKSAT which simply chooses a random assignment for each variable. What is the expected number of clauses satisfied by this algorithm?

**Solution:** Let $m$ be the number of clauses. Any given clause is false with probability $(1/2^k)$, and therefore true with probability

$$1 - \frac{1}{2^k}.$$ 

By linearity of expectation, the expected number of clauses satisfied is $(1 - \frac{1}{2^k}) m$.

(d) [8 points] Consider a cow facing a fence. The cow wants to get around the fence. It knows that there is an opening $k$ steps away, but it does not know in which direction the opening is. Give a 3-competitive algorithm for the cow to find the opening in the fence.

**Solution:** Begin by walking $k$ steps to the right. If you haven’t found the hole, walk $2k$ steps to the left. By then you must have found the opening. It took a total of $3k$ steps and the optimal offline algorithm would take $k$ steps, so the algorithm is $k$-competitive.
(e) [8 points] If $H$ is a universal family of hash functions $h : \{1 \ldots u\} \rightarrow \{1 \ldots 7\}$, show that for any set $K$ of 4 keys there is always $h \in H$ that is perfect (no collisions) on $K$.

Solution: The expected number of collisions is $\binom{4}{2}/7 = 6/7 < 1$. By Markov property, the probability that a randomly chosen $h \in H$ is perfect on $K$ is strictly greater than zero. Thus there is at least one hash function with zero collisions.
Problem 3. 4-Colorability [10 points]

Show that 4-colorability is NP-Complete. Recall that the 4-colorability problem takes as input a graph $G = (V, E)$, and outputs whether there exists an assignment of colors to the vertices such that for every edge, the two endpoints have different colors, and at most 4 colors are used.

*Hint: Reduce from 3-colorability.*

**Solution:** Firstly, 4-colorability is in NP because our certificate can just be a valid 4-coloring of the graph.

For the reduction, let $G = (V, E)$ be the graph we want to test for 3-colorability. Add a new vertex $v$ and draw $|V|$ new edges connecting $v$ to every other vertex in $V$. Given a valid coloring of this new graph, $v$ must have a different color from all the other vertices, so if the new graph is 4-colorable, then $G$ must be 3-colorable. Conversely, if $G$ is 3-colorable, then the new graph is 4-colorable because you can assign a fourth color to the one vertex you added without violating any constraints. Therefore 4-coloring is NP-hard.

**Grader’s Notes.** There were several common mistakes that people made on this problem. One of the most common mistakes was performing the reduction in the wrong direction. Reducing from 3-colorability to 4-colorability means that you must show how to solve 3-colorability *by using 4-colorability as a black box*, not the other way around.

There were also several approaches that some people tried that did not work. One example was adding a vertex $u$ for every set of three vertices $v_1, v_2, v_3$ that formed a clique and adding edges $(u, v_1), (u, v_2), (u, v_3)$, and asserting that the original graph is 3-colorable if and only if the transformed graph is 4-colorable. This would not work on the graph shown below, with the original graph on the left and the transformed graph on the right:

![Diagram](attachment:image.png)

The transformed graph is 4-colorable, but the original graph is not 3-colorable.
Another common mistake was to simply add a new vertex $v'_i$ for each $v_i \in V$ and an edge $(v_i, v'_i)$. In order for this to work you would need all the new vertices to be the same color, but this transformation does not constrain them in any way. You could make this work by adding three additional vertices $u_1, u_2, u_3$ with edges $(u_1, u_2), (u_1, u_3), (u_2, u_3)$ and connecting each $v'_i$ to each of the $u_j$'s. Since the $u_j$'s will use 3 different colors, each $v'_i$ must take on the fourth color, which means that the original graph needs to be 3-colorable.

A third mistake was to split each edge into two edges and create a vertex in between. In other words, transform $G = (V, E)$ into $G' = (V', E')$ where $V' = \{v_1, \ldots, v_n, u_1, \ldots, u_m\}$ (where $n = |V|$ and $m = |E|$) and $E' = \{(v_i, u_j) \mid \text{edge } j \text{ is incident to vertex } i \text{ in the original graph } G\}$. The assertion was that $G$ is 3-colorable if and only if $G'$ is 4-colorable. However, regardless of the structure of $G$, $G'$ is always 2-colorable, since you can color each $v_i$ one color and each $u_j$ a second color.
Problem 4. Priority Queues [10 points]

Consider a priority queue $P$ which supports the following operations:

1. $\text{INSERT}(P, x)$ in $O(\log n)$
2. $\text{EXTRACT-MIN}(P)$ in $O(\log n)$
3. $\text{MERGE}(P_1, P_2)$ in $O(\log n)$
4. $\text{MAX-HEAPIFY}(P)$ in $O(n)$

Say that we construct a new priority queue $P^*$ which consists of a normal priority queue $P$ and a linked-list $L$. It implements its operations as follows:

1. $\text{INSERT}(P^*, x)$: Append $x$ to the end of $L$.
2. $\text{EXTRACT-MIN}(P^*)$:
   - Make $L$ into a heap (via $\text{MAX-HEAPIFY}(L)$)
   - Merge $P$ and $L$ into a new priority queue $Q$ (via $\text{MERGE}(P, L)$)
   - Extract the minimum from $Q$ (via $\text{EXTRACT-MIN}(P)$).

Consider the potential function $\phi = |L|$. Show that for any sequence of INSERTs and EXTRACT-MINS, the amortized cost of an INSERT operation is $O(1)$ and the amortized cost of an EXTRACT-MIN operation is $O(\log n)$.

Solution: The real cost of insertion is $O(1)$ because we just have to append an element to the end of a linked list. $\Delta \phi$ is 1, because we increase the size of $L$ by 1. So the real cost is bounded above by $O(1)$.

Let $s = |L|$. The real cost of EXTRACT-MIN is $O(\log n + s)$. Calling $\text{MAX-HEAPIFY}(L)$ takes $O(s)$, merging $P$ and $L$ takes $O(\log n)$, and extracting the minimum of the new heap takes $O(\log n)$. $\Delta \phi$ is $-s$, since we remove all the elements from $L$. So then the real cost is bounded above by $O(\log n + s - s) = O(\log n)$.
Problem 5. Approximating Knapsack [30 points] (4 parts)

Consider the Knapsack optimization problem we saw in recitation. We introduce a modification such that there is only a single object per item type. Given $n$ items such that item $i$ has value $v_i$ and weight $w_i$, and given a bag that can hold a total weight at most $W$, we want to choose a subset of these items which fits in the bag and has maximum value. Assume that $v_i / w_i$ is distinct for all items $i$, and that $w_i \leq W$ for all items $i$.

(a) [5 points] One obvious algorithm to try is a greedy algorithm: sort the objects in decreasing order of value per unit weight $(v_i / w_i)$ and go through the list in order, taking an object if it still fits in the bag. Show that there are inputs (choices of $v_i, w_i, W$) on which this algorithm might give less than or equal to $1/10$ of the value of the optimal subset of items.

Solution: Suppose we have $W = 20, v_1 = 20, w_1 = 20, v_2 = 2, w_2 = 1$. The greedy algorithm only takes object 2, for a total value of 2, when we could have had value 20. So we get only a $1/10$ fraction of the optimal total worth.

(b) [5 points] The following is an integer linear program for solving Knapsack. Fill in the missing constraint.

\[
\begin{align*}
\text{maximize} & \quad \sum_{k=1}^{n} v_k x_k \\
\text{subject to} & \quad (\text{missing constraint}) \\
& \quad x_k \in \{0, 1\} \quad \forall \, k
\end{align*}
\]

Solution: $\sum_{k=1}^{n} w_k x_k \leq W$
(c) [10 points] Consider the relaxed Linear Program, where we replace $x_k \in \{0, 1\}$ by $x_k \in [0, 1]$. Show that the unique optimal solution of the relaxed LP is given by the greedy fractional algorithm depicted below.

In words, sort the items in decreasing order by $v_i/w_i$. If the item fits in the bag, take it. Let $j$ denote the first item that does not fully fit in the bag. Take the largest fraction of item $j$ to completely fill the remaining space in the bag. Thus the optimal solution is to set $x_i = 1$ for all items considered before item $j$, to set $x_j$ equal to the remaining weight divided by $w_j$, and to set $x_i = 0$ for all items considered after item $j$.

Solution: Basic idea: Prove by contradiction. Given another solution, it must take less of the first $j$ items (since the optimal solution maximizes this), and it must take some weight from a less valuable item (value per unit weight). Thus we can just shift the assignment to a more valuable item and increase the objective function.

(Guilio Gueltrini's solution - good job!) Assume that in the best solution there exists some $i, j$ such that $x_i < x_j$ and

$$\frac{v_i}{w_i} > \frac{v_j}{w_j}.$$  

This assumption is equivalent to assuming that the greedy algorithm is not optimal. We define $w^*$ to be the space occupied by elements $i$ and $j$ in our knapsack, and $v^*$ to be the value of the fraction of the two objects in the knapsack.

$$w^* = x_i w_i + x_j w_j.$$  

$$v^* = x_i v_i + x_j v_j.$$  

Now we consider a new solution where $x'_i w_i + x'_j w_j = w^*$, $x'_i > x_i$, and $x'_j < x_j$. We can always find such a $x'_i, x'_j$, since $x_i < x_j$, so $x_i < 1$, and $x_j > 0$. Now we have found a new solution which still satisfies the constraints given by the weights, but has a strictly higher value. The change in the value of items in the knapsack is

$$(x'_i v_i + x'_j v_j) - (x_i v_i + x_j v_j) > 0.$$  

This contradicts the assumption that $x$ was an optimal solution, therefore the initial assumption is wrong, and the greedy solution must be optimal.

A common mistake was to state that proceeding with the greedy algorithm would maximize the total value. This does not actually constitute a proof of optimality.
(d) [10 points] Consider the following approximation algorithm: Solve the relaxed LP for knapsack. From the previous part, we know that there is at most one item which has a fractional value for \(x_i\). Denote this item to be \(j\). Let \(A\) denote the set of items such that \(x_i = 1\). Compare the total value of the set \(A\) with the value of the single item \(j\), and choose the more valuable set. Show that this is a 2-approximation algorithm.

**Solution:** Let \(f_{OPT}\) be the value of the true optimal solution of Knapsack. Let \(f_{LP}\) be the value of the solution to the relaxed LP. Let \(f_{APX}\) be the value of the approximation algorithm. We know that \(f_{LP} \geq f_{OPT}\) because the relaxed LP optimizes over a strictly larger feasible region. Then observe that \(\text{value}(A) + v_j\) is greater than \(f_{LP}\), since it takes the whole item \(j\). Since the approximation algorithm picks the largest of \(\text{value}(A)\) and \(v_j\),

\[
 f_{APX} = \max(\text{value}(A), v_j) \geq \frac{1}{2}(\text{value}(A) + v_j) \geq \frac{1}{2}f_{LP} \geq \frac{1}{2}f_{OPT}.
\]

Therefore this algorithm gives a 2-approximation for knapsack.

Common mistakes: Many people did not really compare to the knapsack solution, but rather stopped at claiming that this approximation is at least half of the optimal for the relaxed LP. Note that we do not need to find an approximation algorithm for the relaxed LP since we know how to solve it exactly. The goal is to approximate knapsack (which is exactly equivalent to the integer LP).

Many people also assumed that the optimal solution somehow contains a subset of items \(A \cup j\), and thus tried to argue that by choosing \(A\) or \(j\) you could not add more valuable items to the set. The optimal solution in fact does not even need to include any of the items in \(A \cup j\). For example consider a problem with three items: item 1 has weight 6, value 6.5; item 2 has weight 6, value 6; and item 3 has weight 10, value 9. The knapsack has size 10. Then set \(A\) consists of item 1. Item \(j\) denotes item 2. The approximation algorithm will choose item 1. However the optimal set is item 3, which is not even included in \(A \cup j\).

Also people made assumptions or conclusions about the value of \(x_j\) or the weight of the final approximation output. Note that \(x_j\) can be anything within 0 or 1. There is nothing about the problem that implies it is greater or less than \(\frac{1}{2}\). Also the final set chosen does not need to have weight at least \(\frac{1}{2}\). For example the weight of all items in set \(A\) can be very small compared to \(W\), yet the value of \(A\) can still be larger than \(v_j\).