Quiz 1 - 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 every page of this quiz booklet.
- You have 120 minutes to earn 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 a 1-page cheat sheet. No calculators or programmable devices are permitted. No cell phones or other communications devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Pages may be 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 unless asked for.
- Pay close attention to the instructions for each problem. Depending on the problem, partial credit may be awarded for incomplete answers.
- Blank answers will receive approximately 25% of the credit for the problem. If your answer shows no understanding of the problem, you will receive less than 25%. So, if you have no idea how to attack a problem, you should leave it blank.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Parts</th>
<th>Points</th>
<th>Grade</th>
<th>Grader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>25</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>15</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>120</td>
<td>120</td>
<td></td>
</tr>
</tbody>
</table>

Name: Alyssa P. Hacker
Problem 1. [20 points] Fun with sorting

(a) [15 points]
Consider the following sorting algorithms on \( n \) integers ranging from 1 to \( n^2 \):

1. Insertion sort
2. Merge sort
3. Heap sort
4. Counting sort
5. Radix sort
6. AVL sort (insert all items into an AVL tree, then read them in order by performing an in-order traversal in \( \Theta(n) \) time)

For each of the following properties, determine which algorithms in the above list have that property. You may refer to them by index. You do not need to justify your answer.

Example Question: This algorithm correctly sorts a list of elements.

Example Solution: 1,2,3,4,5,6

For each of these problems we took off a point for each incorrect entry and for each incorrect omission.

i. This algorithm runs in time \( O(n \log n) \).

2,3,5,6
All of the algorithms run in \( O(n \log n) \), except for insertion sort (which runs in \( O(n^2) \)) and counting sort (which also runs in \( O(n^2) \) since the values can range from 1 to \( n^2 \)).

ii. This algorithm is stable.

1,2,4,5,6
All of the algorithms are stable except for heap sort. Many noted that merge sort and AVL sort are not necessarily stable and got points, but it is trivial to make them stable and both algorithms are stable in the implementations as demonstrated in class and in the book.

iii. This algorithm is in-place.

1,3
Insertion sort and heap sort are the only listed in-place algorithms. Some of the other algorithms can be modified to be in-place, but the modifications are by no means trivial and we did not cover them in class.

iv. For all values of \( n \), there is an array of \( n \) distinct elements which this algorithm sorts in \( O(n) \) time.

1,5
Insertion sort works in \( O(n) \) time if the elements are already sorted. Radix sort works in \( O(n) \) time for all inputs when the values are integers ranging from 1 to \( n^2 \). Many people incorrectly put counting sort in here thinking that it would run in \( O(n) \) time if all the values range from 1 to \( n \). However, counting sort in this case would still be expecting values ranging from 1 to \( n^2 \) and thus would still take \( O(n^2) \) time.
v. This algorithm works in the comparison model (i.e., the only manipulation of the elements, other than data movement, is by pairwise comparisons: $<,>,\leq,\geq,=$).

1,2,3,6

All of the listed algorithms except for counting sort and radix sort work in the comparison model.

(b) [5 points] For each of the following pairs of elements $(x, y)$, state whether $x \geq y$, $x \leq y$, or $x$ and $y$ are incomparable, i.e., not determinable from the information given. You do not need to justify your answer.

Example Question: $x$ is the first element and $y$ is the last element of a sorted list.

Example Solution: $x \leq y$

i. $x$ is the root and $y$ is a leaf of a max-heap.

$x \geq y$

ii. $x$ is the left child and $y$ is the right child of a node $z$ in a max-heap.

$x$ and $y$ are incomparable

iii. $x$ is the root and $y$ is a leaf of an AVL tree.

$x$ and $y$ are incomparable

iv. $x$ is the left child and $y$ is the right child of a node $z$ in an AVL tree.

$x \leq y$

v. Partway through merge-sorting an array $A$, we have recursively sorted the two halves $A_1$ and $A_2$ of $A$, but not yet merged them. Then $x$ is the first element of $A_1$ and $y$ is the first element of $A_2$.

$x$ and $y$ are incomparable
Problem 2. [25 points] Asymptotics and Recurrences

(a) [5 points] Is it always true that \( f(n) + g(n) = \Theta(\min(f(n), g(n))) \)? If so, prove it. If not, find a counterexample and show that this statement is false.

This claim is false. In fact, \( f(n) + g(n) = \Theta(\max(f(n), g(n))) \). A possible counterexample is \( f(n) = 1 \) and \( g(n) = n \). It is clear that \( g(n) \) asymptotically dominates \( f(n) \). Therefore, \( f(n) + g(n) \) should be \( \Theta(g(n)) = \Theta(n) \). However, the claim stipulates that \( f(n) + g(n) = \Theta(\min(f(n), g(n))) = \Theta(1) \), which is incorrect.

(b) [5 points] Is it always true that a real-valued function \( f \) satisfies \( f(n) = O((f(n))^2) \)? If so, prove it. If not, find a counterexample and show that this statement is false.

This claim is false. First, we note that asymptotic relations are only defined over non-negative functions, so \( f(n) \) and \( f(n)^2 \) must both be non-negative for all \( n > 0 \). A possible counterexample is \( f(n) = \frac{1}{n} \). Therefore, \( f(n)^2 = \frac{1}{n^2} \). If \( f(n) = O((f(n))^2) \), then for sufficiently large \( n \), there exists a positive constant \( c \) such that \( f(n) \leq c \cdot f(n)^2 \). This implies that \( \frac{1}{n} \leq \frac{1}{n^2} \), and therefore, \( n \leq c \). Since no such constant \( c \) can exist, the counterexample must hold.

(c) [5 points] Suppose we have a hash table, with collisions resolved by chaining, and we implement table doubling to resize the table but do not change the hash function. Why is this a bad idea?

Table doubling is employed to distribute data over a larger number of slots, and therefore reduce the length of chains. However, if we double the table without modifying the hash function, we will only hash elements to the first half of the table, leaving any existing chains unaffected. In fact, the performance of all future operations will be exactly the same as if the table doubling never happened. Finally, we end up consuming more memory to house hash cells which will never become occupied.
(d) [5 points] Write and solve the recurrence relation for a **three-way mergesort**, which splits the array into three equal-size pieces, recursively sorts each piece, and then merges them into one sorted array. (You do not need to give the algorithmic details, but your analysis must be correct.) Show your work. You may assume that \( n \) is a power of 3.

Assume that performing three-way mergesort on an array of \( n \) elements takes \( T(n) \) time. Splitting the array into three pieces takes constant time. Recursively sorting each of the three pieces takes \( 3T(n/3) \) time. Merging the three sorted subarrays takes linear time. Therefore, we get the recurrence: \( T(n) = 3T(n/3) + \Theta(n) \).

The number of leaves in the recurrence tree is \( n^{\log_3(3)} = n^1 = n \). Since the number of leaves and the work done at the first level are asymptotically equal to within logarithmic factors, we are in case 2 of the Master Theorem. Therefore, \( T(n) = \Theta(n \cdot \log(n)) \).

(e) [5 points] What is the asymptotic runtime of an algorithm with the following recurrence relation? Show your work.

\[
T(n) = T(\sqrt{n}) + \Theta(1)
\]

**Hint:** What happens when you substitute \( n = 2^m \)?

We first substitute \( n = 2^m \) into the given recurrence.

\[
T(2^m) = T(\sqrt{2^m}) + \Theta(1)
= T(2^{m/2}) + \Theta(1)
\]

Next, we define an auxiliary function, \( S \), such that \( S(m) = T(2^m) \). We can rewrite the above equation in terms of \( S \):

\[
S(m) = S(m/2) + \Theta(1)
\]

Now, we can apply the Master Theorem. There are \( m^{\log_{2}(1)} = m^0 = 1 \) leaves in the recurrence tree. Since the number of leaves and the work done at the first level are asymptotically equal to within logarithmic factors, we are in case 2 of the Master Theorem. Therefore, \( S(m) = \Theta(\log(m)) \).

We know from above that \( T(2^m) = S(m) \). Therefore, we can conclude that \( T(2^m) = \Theta(\log(m)) \). Back-substituting \( n = 2^m \), we get \( T(n) = \Theta(\log(\log(n))) \).
Problem 3. [20 points] Dream Team

Story (feel free to skip this paragraph): Ben Bitdiddle has just been appointed as the captain of his school’s BitBall team, an intense game with \( k \) players on a team. He wants to create the ultimate Dream Team and needs your help. Because the game is so focused on teamwork, Ben realizes he should only put players on the team who are friends with another player already on the team (initially Ben is the only player on the team). Unfortunately, potential players are not allowed to talk to Ben until he has already decided whether they are on the team (to prevent bribery and blackmail—getting on the team is a big deal), so he won’t be able to deduce who is friends with whom beforehand. As a result, Ben decides on the following strategy:

To recruit \( k \) players out of \( N \) total players, initially Ben considers himself the only eligible player. At each step, Ben picks the highest-rated eligible player to put on the team, then asks that player for a list of his friends, and considers all of those friends eligible. Note that a player can be on the team only once, so after they are recruited, they cannot become eligible again.

You may assume that Ben has access to a function \( \text{RECRUITPLAYER}(P) \), which takes the name \( P \) of a player to recruit, and returns a list of the names of that player’s friends. This list is guaranteed to have length at most a known parameter \( m \), and \( \text{RECRUITPLAYER} \) is guaranteed to run in \( O(m) \) time. Ben also has a function \( \text{SKILL}(P) \) that returns the numeric skill rating of the player named \( P \) in constant time.

Using data structures covered in 6.006, describe an algorithm to draft Ben’s Dream Team (or report failure) and briefly analyze its runtime. Your algorithm should start by calling \( \text{RECRUITPLAYER}(\text{"Ben"}) \) to add Ben to the team and get a list of his friends.

Solution: The most common solution to this problem was to use a max-heap to track the highest skill player eligible at every step. The solution works as follows:

1. Call \( \text{RECRUITPLAYER} \) on Ben to get an initial list of eligible players. Use \( \text{BUILDMAXHEAP} \) on this list of players to create a max-heap, \( H \), of eligible players \( \rightarrow O(m) \)
2. Insert all eligible players into a hash table, \( T \). We will use this avoid duplicate player entries in our max-heap (two players may share a friend). \( \rightarrow O(m) \)
3. \( \text{EXTRACTMAX} \) to get the highest skilled eligible player, and add this player to the roster. \( \rightarrow O(\log s) \), where \( s \) is the size of the heap
4. \( \text{RECRUITPLAYER} \) on the newly recruited player, and for every friend, check them against \( T \). If they are not in \( T \), insert them into \( T \) and \( H \) \( \rightarrow O(m \log s) \)
5. Repeat steps 3 and 4 until we have \( k \) players on the team.

In the worst case, our heap has size \( mk \) at the end, so in total the runtime is \( k \times O(m \log mk) = O(mk \log mk) \).

Another equally correct solution was to use an AVL tree instead of a max-heap. This turned out to have no penalty to runtime over the heap solution above.

We also saw a few solutions which did better than the required runtime:
The first was a small optimization on the AVL tree solution to limit our tree size of the best $k$ players (this is okay, because we will only ever have to recruit $k$ players out of the tree). We can do this by discarding the minimum player whenever we add a player to the tree, at no extra asymptotic cost. This reduced the asymptotic cost to $O(mk \log k)$.

A second optimized solution was to make use of a heap-of-heaps data structure. Instead of inserting all additional friends into the large heap, you can build a new heap and insert it into the overall heap as one object, keyed by the maximum node. Picking the maximum of heap-of-heaps is in fact the maximum of all elements, and in this situation we can EXTRACTMAX in $O(\log mk)$ time, and add all friends of a particular player in $O(m + \log k)$ time. This gives us an overall runtime of $O(k \log mk + mk)$. 
Problem 4.  [20 points] **Perfect Pairs**

Alyssa P. Hacker has a large unsorted list $A$ of $n$ numbers, and a target number $k$. In $O(n)$ expected time, she wants to compute the number of pairs of numbers in $A$ that sum to $k$. Assume for this problem that all arithmetical operations between two numbers take $O(1)$ time. Also, note that $A$ may contain repeated numbers.

For example, if $A = [-1, -1, 3.5, 4.5, 9, 4, 4.5, -2]$ and $k = 8$, then the answer would be 4 because $(-1, 9)$ can be made in two ways, $(4, 4)$ and $(3.5, 4.5)$ can be made in one way, and these are the only pairs that sum to 8.

(a)  [5 points] What data structure will best help Alyssa with this task?

The correct answer is a hash table (dictionary). You’ll want to use a hash table that maps keys to the number of times they occur in the array $A$.

It is also possible to use a sorted array for your main data structure, but since the elements of $A$ are arbitrary, we can’t use a linear-time sorting algorithm.

Note that the problem does NOT state that the given numbers in the array are integers, or that they are positive. The given element $k$ could also be 0, negative, or an arbitrary real number.

(b)  [20 points] Describe an $O(n)$ expected time algorithm to solve this problem using the data structure you specified in (a). Partial credit will be given to algorithms that work properly under the assumption that $A$ contains no repeats and/or that $A$ does not contain $k/2$.

The intended answer runs as follows:

1. Initialize dictionary (hash table) $D$ to empty table
2. count = 0
3. For each entry $x$ in A:
   (a) increase count by $D.get(k - x, 0)$
   (b) if $D.has_key(x)$:
       $D[x] += 1$
   else:
       $D[x] = 1$

This approach takes care of repeated elements and elements equal to $k/2$ automatically, since the counts at each increment are equal to the number of previous elements that sum to $k$ with the current element $x$.

Another approach is to fill the hash table first, and then go through and do the counting; this requires more care to get the handling of repeated elements and the handling of elements equal to $k/2$ right.
Problem 5.  [20 points] **Data Structure Mashup**

Recall the following operations on a data structure storing a dynamic set $S$ of $n$ distinct numbers:

- **INSERT($k$):** Add $k$ to $S$.
- **SEARCH($k$):** Return **TRUE** if $k \in S$; otherwise return **FALSE**.
- **SUCCESSOR($k$):** Return the smallest value $x \in S$ with $x > k$, or $\infty$ if none exists.

(a) [10 points] Fill in the running times of the operations above for each of the data structures below. For hashing, assume a table size of $m = \Theta(n)$, assume hashing with chaining, and state expected time bounds.

<table>
<thead>
<tr>
<th>Data structure</th>
<th>INSERT</th>
<th>SEARCH</th>
<th>SUCCESSOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(\lg n)$</td>
</tr>
<tr>
<td>AVL tree</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(\lg n)$</td>
</tr>
<tr>
<td>Hash table</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Heap</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

(We have not directly discussed arrays as data structures. The unsorted array models Python’s **list** data structure, where insert maps to **list.append** and Search/Successor involve linear search. The sorted array enables binary search for Search/Successor but requires a linear-time shift to insert.)
(b) [10 points] Describe a data structure achieving the following running times. You should give explicit algorithms for each of the three operations. You may use any algorithms covered in class as a black box. Hint: Combine some data structures from the list above.

<table>
<thead>
<tr>
<th>Data structure</th>
<th>INSERT</th>
<th>SEARCH</th>
<th>SUCCESSOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

**Solution:** We apply a general transformation for combining two data structures, here AVL trees and hash tables. The queries (Search/Successor) will run in the minimum of the two times and the updates (Insert) will run in the maximum of the two times. Specifically, we maintain the two data structures on the same dynamic set. Ultra’s Insert operation inserts into both data structures, for a cost of $O(\log n) + O(1) = O(\log n)$. Ultra’s Search operation calls Search just on the hash table, which runs in $O(1)$ time. Ultra’s Successor operation calls Successor just on the AVL tree, which we will show can be done in $O(\log n)$ time.

One subtlety (missed by most students) is that we have not seen how to implement an actual Successor operation in binary search trees. We have seen a *next-largest* operation, but its input is a *node* of the tree, while Successor’s input is a *key* (which needs to be found). One trick (found by one student) is to implement Successor by temporarily inserting the key (if it is not already present), then calling next-largest, before deleting the key (if it was inserted). A more direct solution is to search for the key (in the usual way) and maintain the largest key seen along the path. In detail:

```plaintext
Successor(k)
    node = root of tree
    next = ∞
    while node:
        if k < node.key:
            next = node.key
            node = node.left
        else:
            node = node.right
    return next
```
Problem 6. [15 points] Are We Related?

Suppose that you have an AVL tree $T$ with $n$ elements already inserted into it. Describe an efficient algorithm that, given pointers to two nodes $x$ and $y$ in $T$, finds the lowest common ancestor of both $x$ and $y$. For partial credit, achieve a running time of $O(\log n)$. For full credit, achieve a running time of $O(d)$ where $d$ is the distance between nodes $x$ and $y$ in the tree.

For example, if $T$ were the following tree, and $x$ pointed to the node with value 2 and $y$ pointed to the node with value 7, then your algorithm should return the node with value 4.

This problem has numerous possible solutions. We give the most common $O(d)$ solutions here.

Solution 1: Recall that nodes in AVL trees are augmented with 'height', i.e. the longest path from a node to a leaf when following child pointers. Observe that the height of the Least Common Ancestor (LCA) of $x$ and $y$ has to be greater than or equal to the maximum of the heights of $x$ and $y$. This motivates the following algorithm:

\[
\text{LCA}(x, y) \quad \text{while } x \neq y \quad \text{do} \\
\quad \text{if } (x.\text{height} < y.\text{height}) \\
\quad \quad \text{then} \\
\quad \quad \quad x = x.\text{parent} \\
\quad \quad \text{else} \\
\quad \quad \quad y = y.\text{parent} \\
\quad \text{return } x
\]

At each step, we check which node has a smaller height (we arbitrarily break ties in favor of $y$), and set its pointer to its parent. Note that both pointers will eventually reach the LCA, at which point the loop will break. We will never step past the LCA because its height must be strictly greater than the heights of all of its descendants, so as long as one pointer has reached the LCA and the other has not, only the lower one will step up.

This loop runs exactly $d$ times, since the distance from $x$ to $y$ is exactly the distance from $x$ to the LCA plus the distance from $y$ to the LCA. The body of the loop can be run in $O(1)$ time, so the total running time is $O(d)$. 

Solution 2:

Consider the paths from $x$ to the root and $y$ to the root that we obtain by following parent pointers. Observe that the LCA is the first node at which these paths intersect (after this point they are identical). Efficiently detecting the intersection point takes some care, but it can be done with a hash table as in the following algorithm:

\[
\text{LCA}(x, y) \\
1 \quad \text{HashTable } Hx, Hy \leftarrow \emptyset \\
2 \quad \text{while } True \\
3 \quad \quad \text{do} \\
4 \quad \quad \quad \text{if } (Hy.find(x)) \\
5 \quad \quad \quad \quad \text{then return } x \\
6 \quad \quad \quad \text{elseif } x.\text{HAS\textsc{PARENT}}() \\
7 \quad \quad \quad \quad \text{then} \\
8 \quad \quad \quad \quad \quad Hx.\text{INSERT}(x) \\
9 \quad \quad \quad \quad \quad x = x.\text{parent} \\
10 \quad \quad \text{end} \\
11 \quad \quad \text{if } (Hx\text{.find}(y)) \\
12 \quad \quad \quad \text{then return } y \\
13 \quad \quad \text{elseif } y.\text{HAS\textsc{PARENT}}() \\
14 \quad \quad \quad \text{then} \\
15 \quad \quad \quad \quad Hy.\text{INSERT}(y) \\
16 \quad \quad \quad y = y.\text{parent} \\
\]

At each step, we check if the current pointer for $x$ is in the path from $y$ to the root, and if so it must be the LCA and we return it. Otherwise, we insert $x$ into its hash table and step it up to its parent. We do symmetric operations on $y$. Note that if one of the pointers reaches the root, it must still allow the other pointer to finish its traversal up the tree.

This loop runs a number of times exactly equal to the maximum of the distance from $x$ to the LCA or from $y$ to the LCA. This is at most $d$ times. Furthermore, the body of the loop can be executed in $O(1)$ time with a good hash table implementation. Therefore, the total running time is $O(d)$.

A note on $O(\log n)$ solutions:

Several unrelated $O(\log n)$ solutions were possible. For example, we can traverse down the search path for both $x$ and $y$ starting from the root. As long as the two search paths match, we are above the LCA. The first node at which the paths diverge is the LCA and we may return it.

A simpler variation on the approach using height could be done with depth instead. We can compute the depth of each node in $O(\log n)$ time, step the deeper node up the tree (following parent pointers) until we reach the same height as the other node. Now, we can step the nodes up in lock-step until they meet. This works for depth but not height because the depth of a node’s parent is always exactly one greater than the depth of the node. Note that we cannot augment AVL trees with depth as we can with height because depth cannot be maintained in $O(1)$ time through rotations.
Grading Note: Many solutions assumed that a node is not an ancestor of itself. In fact, the conventional definition of 'ancestor' does allow this to be the case (so that LCA(\(x, y\)) = \(x\) if \(y\) is a descendant of \(x\)) but no points were taken off for solutions using the alternative definition.