Practice Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains five multi-part problems. You have 80 minutes to earn 80 points.
- This quiz booklet contains 11 pages, including this one and an extra sheet of scratch paper, which is included for your convenience.
- This quiz is closed book. You may use one handwritten Letter ($8\frac{1}{2}'' \times 11''$) or A4 crib sheet. No calculators or programmable devices are permitted.
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time and paper re-deriving facts that we have studied. It is sufficient to cite known results.
- Do not spend too much time on any one problem. Read them all through first, and attack them in the order that allows you to make the most progress. Generally, a problem’s point value is an indication of how much time 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. Be neat.
- Good luck!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
<th>Grade</th>
<th>Initials</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>80</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 1. Recurrences [?? points] (3 parts)
For each of the following recurrences, give an asymptotically tight ($\Theta(\cdot)$) bound. Justify your answer by naming the particular case of the Master’s Method, by iterating the recurrences, or by using the substitution method. As usual, assume that for $n \leq 10$, $T(n) = O(1)$.

Example: [0 points] BINARY SEARCH
Recurrence: $T(n) = T(n/2) + c$
Solution by iteration:

$$T(n) = T(n/4) + c + c = \sum_{i=0}^{\log n} c = c \log n = \Theta(\log n)$$

(a) [?? points] $T(n) = 8T(n/2) + \Theta(n)$.

Solution: $T(n) = \Theta(n^3)$ by part 1 of the Master’s Method.

(b) [?? points] $T(n) = 9T(n/9) + \Theta(n^{\sqrt{3}})$.

Solution: $T(n) = \Theta(n^{\sqrt{3}})$ by part 3 of the Master’s Method.
(c) [?? points] \(T(n) = T(\sqrt{n}) + \log n\). (It is fine to assume that \(n\) is of the form \(2^{2k}\) in order to avoid floor and ceiling notation.)

**Solution:** Let \(n = 2^k\). Then the recurrence becomes \(T(2^k) = T(2^{k/2}) + k\), or, if we set \(L(k) = T(2^k)\), \(L(k) = L(k/2) + k\). This solves to \(L(k) = \Theta(k)\) by part 3 of the Master’s Method. Thus, \(T(n) = \Theta(k) = \Theta(\log n)\).

Alternative solution is note that \(T(n) \geq \log n\) and, for the upper bound, iterate the recurrence:

\[
T(n) = T(\sqrt{n}) + \log n = T(n^{1/4}) + \log \sqrt{n} + \log n \leq \sum_{i=0}^{\infty} \log n^{1/2^i} = \log n \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2 \log n.
\]

Many students made mistakes on this problem, but partial credit (2-3 points) was given where solutions were on the right track, but made a math mistake or used the wrong case of the Master Method (e.g. finding \(L(k) = \Theta(k \log k)\) instead of \(L(k) = \Theta(k)\)).
Problem 2. True or False, and Justify [?? points] (4 parts)

Circle T or F for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

(a) **T F** [?? points] \( f(n) = \Theta(g(n)) \) is equivalent to \( g(n) = \Theta(f(n)) \).

**Solution:** True. \( f(n) = \Theta(g(n)) \) means there exists some \( n_0 \) and constants \( c_1, c_2 > 0 \) such that for \( n \geq n_0 \) we have \( c_1 g(n) \leq f(n) \leq c_2 g(n) \). But this is equivalent to \( \frac{1}{c_2} f(n) \leq g(n) \leq \frac{1}{c_1} f(n) \), or \( g(n) = \Theta(f(n)) \).

Many people simply argued that if \( f(n) = \Theta(g(n)) \) then \( f(n) \) “grows asymptotically as fast as” \( g(n) \), which implies that \( g(n) \) also grows as fast as \( f(n) \). This argument is clearly not enough since we expected that you use the definitions, so no points were given for that justification.

Some people also assumed the fact that \( f(n) = O(g(n)) \) implies that \( g(n) = \Omega(f(n)) \) to conclude (which, although it is true, needed a line of justification and so one or two points were taken off in those cases).

(b) **T F** [?? points] There is a deterministic algorithm to sort \( n \) numbers in the comparison model running in time \( \log F_n \), where \( F_n \) is the \( n \)th Fibonacci number. (Recall that the Fibonacci numbers are defined as follows: \( F_0 = 0, F_1 = 1 \) and for \( i > 1, F_i = F_{i-1} + F_{i-2} \).)

**Solution:** False. Remember that \( F_n \leq c^n \), where \( c \) is some constant. Then, \( \log F_n = n \log c = O(n) \). In the comparison model, we know that sorting takes \( \Omega(n \log n) \) time.

Some people also justified this part by saying that it is impossible to sort 2 numbers using \( 0 = \log 1 = \log F_2 \) comparisons in the comparison model. Although the argument is correct we were expecting an asymptotic argument using the lower bound for sorting in the comparison model. Also, a common mistake was to confuse the magnitude of \( F_n \) with the time needed to compute the \( n \)-th Fibonacci number, which is exponentially smaller.
Problem 3. Hashing

Design a data structure called DISTINCT which maintains a multi-set of integers in the range \(\{1, 2, \ldots, n^2\}\) under insertions and deletions. A multi-set is a set of elements where duplicate keys are allowed (e.g., \(\{1, 2, 1, 3\}\) is a multi-set). At any time, the data structure should provide the current number of distinct elements in the multi-set. For example, the multi-set \(\{1, 2, 1, 3\}\) contains 3 distinct elements.

Design a randomized data structure that supports the above operations in \(O(1)\) expected time. You can assume that at any time the total number of distinct elements in the set is at most \(n\). Moreover, you can assume that you are given an \(O(n)\)-size empty block of memory at the beginning.

Solution: We maintain a hash table of size \(n\) to stored integers in the multiset along with their duplication counts. For each insertion, we lookup the inserted integer in the hash table. If the number is already there, we increase to its duplication count by 1. If it is not, we add that it to the table with duplication count 1. For each deletion, we look up the deleted number in the table, decrease its duplication count by 1 and remove it from the table if the duplication count is 0. We also need to maintain a distinct count for answer queries. This count is increased by 1 whenever new number is added into the table and decreased by 1 whenever a number is removed from the table.

By choosing a hash function \(h\) randomly from a universal hashing family, the expected time for each table lookup is \(O(1)\), and therefore, \(O(1)\) for each operation.
Problem 4. Plural Elements [?? points] (2 parts)
You are given an array of $n$ integers. For each of the questions below, remember to give a brief, clear explanation of why the algorithm works.

(a) [?? points] Suppose there exists an integer that appears more than $n/2$ times in the array. Give a linear time deterministic algorithm to find such an integer.

Solution: Find the median of the array – it will be the majority element. Since, if the median wouldn’t be the majority element, then the majority element would be smaller or larger than the median, which is impossible (e.g., there are only $n/2$ elements smaller than the median).

Many people thought an algorithm using the COUNTING idea, which was used in COUNTING SORT. But, it need an assumption for the range of elements, otherwise it is impossible to implement and it may not run in $O(n)$ time. Also, some people designed an algorithm essentially using the Sorting algorithm, which runs in $\Theta(n \log n)$ time. Both answers got approximately half of full points.

(b) [?? points] Now suppose there exists an integer that appears more than $n/k$ times, where $k > 2$ is a constant (suppose $n$ is divisible by $k$). Give a linear time deterministic algorithm to find all such integers.

Solution: Let’s call a common element an integer that appears more than $n/k$ times. Let $a_1, a_2, \ldots a_k$ be the elements with ranks respectively $n/k, 2n/k, \ldots (k-1)n/k, n$. Then we check each of the elements $a_1, a_2, \ldots a_k$ whether it is a common element (just a linear scan per element).

The correctness of the algorithm follows from the claim that any common element must be among $a_1, a_2, \ldots a_k$. To see that, consider the sorted array. Any common element is a contiguous block of size $> n/k$. Since our “probes” $a_1, a_2, \ldots a_k$ are at distance $n/k$, they must strand the block of a common element.
Problem 5. Min and Max Revisited [?? points] (3 parts) In this problem we will investigate a divide and conquer approach for simultaneously finding the minimum and maximum of an array of \( n \) integers. in the comparison model (i.e., your algorithm is only allowed to compare elements, but not add/subtract/index with them).

Assume for simplicity that \( n \) is a power of 2.

For each of the questions below, remember to give a brief and clear justification.

(a) [?? points] Suppose you compute the maximum separately and the minimum separately, and output both. How many comparisons does this require?

Solution: Computing the maximum takes \( n - 1 \) comparisons and the minimum takes another \( n - 1 \) comparisons. Total number of comparisons is \( 2n - 2 \).

(Full points were given for just saying \( \Theta(n) \). A few students thought we were asking for lower bounds and gave lower bounds of \( \Omega(n) \) or \( \Omega(\log n) \). Both answer got full points as well.)
(b) [?? points] For the rest of this problem, we would like to reduce the leading constant factor in the number of comparisons. We want a result that is not necessarily better asymptotically, but in terms of the exact number of comparisons.

Let’s consider an idea for designing a divide and conquer solution for simultaneously computing the minimum and maximum element. The plan is to break the problem into two sublists, compute the min and max of each sublist and then combine the results. An outline of such an algorithm is given below. In the following page, you will be given the opportunity to fill in some details:

\[
\text{MINMAX}(A, n);
\]

\[
\text{If } n = 2 \text{ then } \langle \ldots \rangle \text{ fill in base case}
\]

\[
\text{else Let } A_1 = \text{left half and } A_2 = \text{right half}
\]

\[
(a, b) = \text{MINMAX}(A_1, n/2)
\]

\[
(c, d) = \text{MINMAX}(A_2, n/2)
\]

\[
\langle \ldots \rangle \text{ fill in combine stage}
\]

(i) Fill in the details for the case \( n = 2 \).


A few solutions said “Return \(\text{min}(A[1], A[2]), \text{max}(A[1], A[2])\)” . Since the goal is to minimize actual number of comparisons this is wasteful. Usually (depending on clarifications in Part (c)), two to four points were taken off for this error.

(ii) Fill in the details for the combine step here.

**Solution:** If \( a < c \) let \( a' = a \), else \( a' = c \). If \( b < d \) then \( b' = d \) else \( b' = b \). Return \((a', b')\)
(c) [?? points]

Let $T(n) = c_1 n - c_2$ be the number of comparisons used by the algorithm. Find an exact expression for $T(n)$ (i.e., the best values of the constants $c_1$ and $c_2$). (You may do this by writing out and solving the recurrence for $T(n)$ using the substitution method.) Is this really better than the naive algorithm?

**Solution:** The recurrence for the number of comparisons is $T(n) = 2T(n/2) + 2$ with the base case $T(2) = 1$.

We guess that the solution is of the form $T(n) = c_1 n - c_2$. Then we have $c_1 n - c_2 = 2(c_1 n/2 - c_2) + 2$, giving $c_2 = 2$. Now plugging in $T(2) = 2c_1 - 2 = 1$, we get $c_1 = 3/2$.

Thus the number of comparisons is $(3/2)n - 2$, which has a better leading constant than the naive solution.

Alternate solutions expanded the recurrence as $2+4+\cdots+n/2+n/2T(2) = 3/2n-2$, using $T(2) = 1$. Such a solution also received full credit.

Common mistakes in this part were to ignore the base case, or to say $T(n) = 2T(n/2) + \Theta(1)$. 
Problem 6. Matrix Multiplication [?? points] (2 parts)

In this problem we are given three matrices and the question is to compute their product. For each of the two cases below, report the fastest algorithm you know to compute the product of the given three matrices. You need to prove correctness of the algorithm and argue its running time.

(a) [?? points] Given matrices $A, B, C$, each of dimension $n \times n$, give a fast algorithm to compute $A \cdot B \cdot C$? What is its asymptotic running time? (You may use algorithms mentioned in lectures without describing them.)

Solution: The solution depends on what’s the fastest way to compute the product of just two matrices. Suppose it takes $T$ time to compute the product of two $n \times n$ matrices. Then, computing $A \cdot B \cdot C$ takes $2T$ time – just compute $A \cdot B$ first, and then multiply the result, also an $n \times n$ matrix, by $C$.

What is the best known $T$? Strassen’s algorithm gives a $T = O(n^{\log_2 7})$ time algorithm for computing the product of two matrices. The currently best known algorithm has $T = O(n^{2.376})$, and is due to Coppersmith and Winograd. We gave full points for the answer $O(n^{\log_2 7})$, and four points for knowing there is a faster algorithm than the straightforward $O(n^3)$ algorithm, but not remembering the runtime.

Several solutions forgot to mention the runtime, two to three points were taken off for this type of error.

(b) [5 points] Now, suppose $C$ is of dimension $n \times 1$ (i.e., $C$ is a vertical vector), and $A$ and $B$ are still of dimension $n \times n$. Give a fast algorithm for computing $A \cdot B \cdot C$? What is its asymptotic running time?

Solution: Here the trick is to perform the computations in the right order, using the fact that multiplying an $n \times n$ matrix by an $n \times 1$ vector can be done in $O(n^2)$ time with the straight-forward algorithm. First compute $B \cdot C$, which takes $O(n^2)$ time. Then compute $A \cdot (B \cdot C)$, which is also a multiplication of an $n \times n$ matrix by a $n \times 1$ vector, and takes $O(n^2)$ times. So, the total running time is $O(n^2)$.

Some people thought that matrix multiplication is commutative, please check to convince yourself that it is not. Three to four points were given for the right algorithm, but wrong runtime analysis.
Problem 7. Joining and Splitting 2-3-4 Trees

The JOIN operator takes as input two 2-3-4 trees, $T_1$ and $T_2$, and an element $x$ such that for any $y_1 \in T_1$ and $y_2 \in T_2$, we have $\text{key}[y_1] < \text{key}[x] < \text{key}[y_2]$. As output JOIN returns a 2-3-4 tree $T$ containing the node $x$ and all the elements of $T_1$ and $T_2$.

The SPLIT operator is like an “inverse” JOIN: given a 2-3-4 tree $T$ and an element $x \in T$, SPLIT creates a tree $T_1$ consisting of all elements in $T - \{x\}$ whose keys are less than $\text{key}[x]$, and a tree $T_2$ consisting of all elements in $T - \{x\}$ whose keys are greater than $\text{key}[x]$.

In this problem, we will efficiently implement JOIN and SPLIT. For convenience, you may assume that all elements have unique keys.

(a) Suppose that in every node $x$ of the 2-3-4 tree there is a new field $\text{height}[x]$ that stores the height of the subtree rooted at $x$. Show how to modify INSERT and DELETE to maintain the height of each node while still running in $O(\log n)$ time. Remember that all leaves in a 2-3-4 tree have the same depth.

Solution: Let leaf nodes have a height of 1 and internal nodes have $\text{height}[x] = 1 + \text{height}[	ext{child}(x)]$. A node affected by INSERT or DELETE operations will simply recalculate their height value by looking at the height of their children. In both INSERT and DELETE, at most $O(\log n)$ nodes positions will be affected and each of their height values can be updated in $O(1)$ time. Therefore, the added calculation cost of maintaining height fields is $O(\log n)$.

A slight caveat in using this method is that we must ensure that heights are calculated from the bottom-up, otherwise there could be a case where a parent computes its height from an out of date child. Fortunately, both INSERT and DELETE recursively work from the bottom of the tree upward, so this is not an issue.

(b) Using part (a), give an $O(1 + |h_1 - h_2|)$-time JOIN algorithm, where $h_1$ and $h_2$ are the heights of the two input 2-3-4 trees.

Solution: Find the heights of $T_1$ and $T_2$. If $h_1 > h_2$, find the node $z$ with depth $h_1 - h_2$ on the rightmost path of $T_1$ and insert $x$ into $z$. If $z$ is full, it will be split with a key floating up as in INSERT. Set the rightmost child of the node containing $x$ to be the root of $T_2$. Now every leaf in the resulting 2-3-4 Tree has depth $h_1$, and the branching constraint is obeyed. The case for $h_1 < h_2$ is similar. If $h_1 = h_2$, merge the two root nodes along with $x$ into a “fat” node, and split the node if it is overloaded.

It takes $O(1 + |h_1 - h_2|)$ time to find the node $z$ and insert $x$ into $z$, and $O(1)$ time to join the smaller tree to the larger tree. Therefore the total running time is $O(1 + |h_1 - h_2|)$. 
(c) Give an $O(\log n)$-time SPLIT algorithm. Your algorithm will take a 2-3-4 tree $T$ and key $k$ as input. To write your SPLIT algorithm, you should take advantage of the search path from $T$’s root to the node that would contain $k$. This path will consist of a set of keys $\{k_1, \ldots, k_m\}$. Consider the left and right subtrees of each key $k_i$ and their relationship to $k$. You may use your JOIN procedure from part (b) in your solution.

Solution:

1. Initialize two empty trees $T_1$ and $T_2$.
2. Search for the element $k$ in the tree $T$.
3. If the search path at node $k_i$ traverses right, INSERT $k_i$ into $T_1$ and JOIN $k_i$’s left subtrees with $T_1$.
4. If the search path at node $k_i$ traverses left, INSERT $k_i$ into $T_2$ and JOIN $k_i$’s right subtrees with $T_2$.
5. If $k$ is found JOIN $k$’s left child with $T_1$ and its right child with $T_2$.
6. If a leaf node is encountered, insert any remaining elements into their appropriate tree.

Let $k_l$ be some key less than $k$. Then $k_l$ will either: (1) be a node which the search path turned right on, (2) be less than some node that the search path turned right on, or (3) be a left child of $k$. In all three cases, $k_l$ will be INSERTed or JOINed into $T_1$. Similarly, any nodes greater than $k$ will be placed in $T_2$.

The algorithm joins the subtrees as it walks down the 2-3-4 tree along the search path. Therefore the height of subtrees never increases. In other words, we have $\text{height}[T_{i-1}] \geq \text{height}[T_i]$. Searching for $k$ takes $O(\log n)$ time. Let $h_i$ denote the height of subtree $T_i$. The running time for the iterative JOIN takes:

$$O\left(\sum_{i=1}^{m} (1 + |h_{i-1} - h_i|)\right) = O\left(\sum_{i=1}^{m} (1 + h_{i-1} - h_i)\right)$$

$$= O(m + h_0 - h_m)$$

$$= O(m + \log n).$$

Since a 2-3-4 tree has at most 4 branches, the algorithm can join at most 3 subtrees before the search path goes down 1 level in the 2-3-4 tree. Therefore, the number of subtrees $m$ joined is at most 3 times the depth of the key $k$. Therefore $m = O(\log n)$ and the time complexity of the SPLIT operation is $O(\log n)$. 
