Quiz 1 Solutions

- These are the quiz solutions
- The mean for this quiz was: 65.3
- The standard deviation was: 13.2

Problem	Points	Grade
1	20	
2	32	
3	22	
4	16	
Total	90	

Name: _____

Problem 1. Recurrences [20 points]

Solve the following recurrences (provide only the $\Theta()$ bounds). You can assume $T(n) = 1$ for n smaller than some constant in all cases. You *do not* have to provide justifications, just write the solutions.

- $T(n) = T(n/7) + 1$

  $T(n) = \theta(\log\ n)$
  Master's Theorem, case 2

- $T(n) = 3T(n/3) + n$

  $T(n) = \theta(n\ \log\ n)$
  Master's Theorem, case 2

- $T(n) = 5T(n/5) + n \log n$

  $T(n) = \theta(n\ \log^2\ n)$
  Extended Master's Theorem, case 2

- $T(n) = 10T(n/3) + n^{1.1}$

  $T(n) = \theta(n^{\log_3 10})$
  Master's Theorem, case 1

Problem 2. True or False, and Justify [32 points] (8 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, respectively. If the statement is correct, briefly state why. If the statement is wrong, explain why. Your justification is worth more points than your true-or-false designation.

T F The solution to the recurrence

$$T(n) = T(n/3) + T(n/6) + n^{\sqrt{\log n}}$$

is $T(n) = \Theta(n^{\sqrt{\log n}})$ (assume $T(n) = 1$ for n smaller than some constant c).

True. First because of the $n^{\sqrt{\log n}}$ term, $T(n) \geq n^{\sqrt{\log n}}$. To obtain an upper bound, one can argue that $T(n)$ can be shown to be non-decreasing and that an upper bound can be found by solving the recurrence $U(n) = 2U(n/3) + n^{\sqrt{\log n}}$, which by the master method leads to $U(n) = O(n^{\sqrt{\log n}})$. Putting the upper and lower bounds together, we get $T(n) = \Theta(n^{\sqrt{\log n}})$.

T F Radix sort works in linear time **only if** the elements to sort are integers in the range $\{1 \ldots cn\}$, for some $c = O(1)$.

False. Radix sort also works in linear time if the elements to sort are integers in the range $\{1, \ldots, n^d\}$ for any constant d.

Note. On page 172 of CLRS, it is shown that radix sort runs in $\Theta(d(n + k))$ to sort n d-digit numbers with each digit in the range 0 to $k - 1$. If we take $k = n$ and $d = O(1)$, the running time is $O(n)$, but with d digits in base n, we get a range of n^d. The main mistake that many people did was to assume that digits were always between 0 and 9.

T F There exists a comparison-based sorting algorithm that can sort any 6-element array using at most 9 comparisons.

False. A decision tree for sorting a 6-element array has to have at least $6! = 720$ leaves, but if we perform at most 9 comparisons, we can get at most $2^9 = 512$ leaves which is less than 720.

Note. Some people said that the number of comparisons is at least $n \log_2 n$, but this is not correct. The lower bound is $\log_2(n!)$ which is not $n \log_2 n$ (asymptotically, they have the same order of growth).

T F Consider an implementation of Quicksort which always chooses the partition element x to be equal to $A[3]$, where $A[1 \ldots n]$ is the array to partition. This implementation of Quicksort runs in $O(n \log n)$ time in the worst case.

False. If we apply it to the array $n, n-1, 1, 2, 3, 4, \cdots, n-2$, we would get a running time of $T(n) = T(n-1) + \Theta(n)$, which is $T(n) = \Theta(n^2)$.

Note. Many people seemed to assume that we were partitioning on the third *smallest* element in the array, while we were partitioning on the third element of the unsorted array.

T F Computing the most frequently occurring element in an array $A[1 \ldots n]$ can be done in $O(n \log n)$ time.

True. The idea is simple. First we sort the array A using a general-purpose sort such as merge-sort. Once we have a sorted list, we need to scan from left to right looking for the longest sequence of the 'same' data. Sorting takes $\Theta(n \log n)$. Going through the data takes $\Theta(n)$. The algorithm is therefore $\Theta(n \log n)$ which is certainly $O(n \log n)$.

The scan for the most frequently occurring element can be done by having a counter that keeps track of the 'longest sequence seen so far' and the 'item in that sequence'. Then, simply compare it to the new sequence you've found, and if it's longer, assign it to the counter.

Note: A lot of people suggested using a 'counting' approach. The idea was to use the 'counting sort' algorithm to count the occurrences of each value, and then spit out the one with the most occurrences. The problem with this is that counting only works when you know a simple small range for your elements (k). If the range is large (say $k = n^2$) then the algorithm really runs in k time, which can be higher than $n \log n$. Furthermore, if the data came from all possible reals, k is infinite, which means counting just can't be done at all.

T F Consider a set S of n two-dimensional points $(x_1, y_1), \ldots (x_n, y_n)$, n even. Assume that all coordinates x_i, y_j are different. A vertical line L is a *separator* for S if the number of points of on the left of L is equal to the number of points on the right of L.

A separator for S can be computed in $O(n)$ time.

True. First of all, since we're looking for a vertical separator, we know we can ignore the y values. Now we just want to find a value of x that is right in the middle of all the values. This translates to finding the median (specifically, since there are an even number of points, we need to find the $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 1$ elements, and put the line somewhere in between).

We were shown in class algorithms to do this that ran in $O(n)$ time (notably RANDOMIZED-SELECT and SELECT).

Therefore, we can do this in $O(n)$ time.

Note: Again people assumed we can sort the data using a linear-time sort (such as radix or counting) and then picking the middle element. However, since we do NOT know anything about the data (it could be badly-distributed real values), we can't sort it in linear time.

T F Consider a n-digit decimal number $x_1x_2\cdots x_n$ (i.e. each x_i is between 0 and 9). Is the hash function $f(x) = \sum_{i=1}^{n} a_i x_i \pmod 7$ universal where the a_i's are independently and uniformly selected in $\{0, 1, 2, 3, 4, 5, 6\}$?

False. Consider a key k_1 that has the digits 7, 8, or 9 in some positions. Construct k_2 that toggles some of these digits to be 0, 1, or 2, respectively. Regardless of the choices of a_i, the keys k_1 and k_2 will hash to the same position. Thus, a single digit counterexample stating that the keys 8 and 1 collide with probability 1 is good enough for an answer.

Note: Some people argued that this is true because this is the same function as was given in class. However, they missed the subtlety of the fact that the example in class asserted that the input is broken into digits whose values are less than m (in this case $m = 7$). However, here the digits are possibly bigger than m.

T F Is your name written on every page of this booklet? (Yes, we will check it.)

Unclear. It turns out that the above isn't a boolean question, and with the errata in class it was not grammatically correct. However, those who marked 'true' were expected to have their names on every page. Those who marked 'false' were expected NOT to have it. Those who didn't choose an option got no credit.

For justification, we had proofs varying from induction to references to large tomes on the topic. And you thought WE were bad....

Problem 3. Anagram pattern matching [22 points]

Assume you are given a *text* array $T[1 \ldots n]$ containing letters from the standard Latin alphabet. In other words, $T[i] \in \{a, b, \ldots z\}$ for $i = 1 \ldots n$. In addition, you are given a *pattern* array $P[1 \ldots m]$, $m < n$, which also contains letters from the Latin alphabet. For any sub-array $T_i^m = T[i \ldots i + m - 1]$ of T, we say that T_i^m is an *anagram* of P if there is a way of permuting symbols in T_i^m so that the resulting array is equal to P.

(a) (8 points) Give an algorithm that, given an index i (and m), determines whether T_i^m is an *anagram* of P. Try to give an algorithm that is as efficient as possible. However, partial credit will also be given for less efficient solutions.

The two m-element sequences P and T_i^m will be anagrams of each-other if and only if their sorted orderings are equal. Based on this observation, our algorithm is to sort both P and T_i^m, and to compare the resulting sorted orderings. Since the elements from these sequences come from a constant-size alphabet, we can use counting sort to sort both of them in $\Theta(m)$ time. Afterwards, it takes $\Theta(m)$ time compare the sorted orderings. The total running time is therefore $\Theta(m)$.

Notes: Many students provided correct algorithms which were less efficient, for exmaple with running time $\Theta(m^2)$. Such solutions generally received at most half of the possible points. Some students also confused n with m.

(b) (14 points) Design an algorithm, which given T and P as an input, reports all i's such that T_i^m is an anagram of P. Ideally, your algorithm should run in $O(n+m)$ time. However, partial credit will also be given for less efficient solutions.

Start with $i = 1$. Let us count the number of occurences of each letter in P and in T_i^m, just as is done in the first stage of the counting sort algorithm. Specifically, we will scan through the elements of P and construct an array $C[a \ldots z]$ of counts, such that $C[l]$ gives the number of times the letter l appears in the array P. Similarly, we create an array $D[a \ldots z]$ which contains the number of occurences of each letter in T_i^m. Construction of the arrays C and D will consume $\Theta(m)$ time, as it requires a single linear scan over P and T_i^m. Note that we can compare the arrays C and D in $\Theta(1)$ time, as the arrays have constant size. If $C = D$, then P will be an anagram of T_i^m.

Now let us loop over all values of i. For each value of i, we check, in $\Theta(1)$ time, if $C = D$, and if so, we output that value of i since T_i^m will be an anagram of P. When we move from i to $i+1$, we will update the array D by decrementing $D[T[i]]$ (since the letter $T[i]$ will no longer be present in T_{i+1}^m) and by incrementing $D[T[i+m]]$ (since $T[i+m]$ will now be present in T_{i+1}^m). In total, we spend $\Theta(1)$ time per iteration of our loop over i, for a total running time of $\Theta(n)$ plus the initial cost $\Theta(m)$ of constructing C and D for $i = 1$. Therefore, total running time is $\Theta(m+n)$.

Notes: A common solution to this problem was to apply the solution of part (a) for every index i, which results in a running time of $\Theta(mn)$. This solution, and similar other inefficient solutions, were generally awarded at most 6 points.

Problem 4. Mode finding [16 points]

Assume that you are given an array $A[1 \ldots n]$ of distinct numbers. You are told that the sequence of numbers in the array is *unimodal*, i.e., there is an index i such that the sequence $A[1 \ldots i]$ is increasing (i.e. $A[j] < A[j+1]$ for $1 \leq j < i-1$) and the sequence $A[i \ldots n]$ is decreasing. The index i is called the *mode* of A.

Show that the mode of A can be found in $O(\log n)$ time.

The most common solution was to modify binary search to look at the *pair* of successive middle elements of the array.

If the array is only a single element, that element is the mode. If the successive elements are increasing, we know that our elements are in the increasing portion of the array. Thus, the mode will be found to the right of the pair, and we recurse on that half of the array. Otherwise, we know the pair is in the decreasing portion of the array, and the mode will be found to the left. (Assuming indices increase from left to right.)

```
FIND_MODE(A)
1  if (length(A) = 1) then return 1
2  mid ← ⌊length(A)/2⌋
3  if A[mid] > A[mid + 1]
4     then return FIND_MODE(A[1 ... mid])
5     else  return mid + FIND_MODE(A[mid + 1 ... length(A)])
```

To compute the running time we note FIND_MODE employs a divide and conquer strategy. The divide step requires constant work to compute the middle index of the array. The combine step require constant work to compute the return value. The conquer step recurses on half the array, yielding the recurrence $T(n) = T(n/2) + \Theta(1)$, which implies $T(n) = \Theta(\log n)$, as needed.

Note:

By far the most common error, was sloppiness in the base case. Many people compared three elements to find the mode in the base case. Two problems with this solution were common. First, there was no guarantee the array would divide such that the base case had exactly three elements. Second, the test often failed in the case that the mode was the first (or last) element of the array. Note that the problem does not prohibit the case where the increasing (or decreasing) portion of the array is empty. Some students wanted to find two equal elements in the base case. Unfortunately, the problem specification implies all the elements in the array are unique. A few enterprising students selected the index of the pair of elements to compare at random. Unfortunately, even if analyzed correctly, this only gives *expected* time of $\Theta(\log n)$ and we wanted *worst case* time $\Theta(\log n)$. Another small group of students submitted correct but linear time solutions.