Quiz 1

- Do not open this quiz booklet until you are directed to do so.
- This quiz ends at 3:55 P.M.
- When the quiz begins, write your name on the top of every page in this quiz booklet, because the pages will be separated for grading.
- Write your solutions in the space provided. If you need more space, write on the *back* of the sheet containing the problem. Do not put part of the answer to one problem on the back of the sheet for another problem.
- Plan your time wisely. Do not spend too much time on any one problem. Read through all of them first and attack them in the order that allows you to make the most progress.
- 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.
- When describing an algorithm, describe the main idea in English. Use pseudocode only to the extent that it helps clarify the main ideas.
- Good luck!

Problem	Points	Grade
1	24	
2	32	
3	24	
Total	80	

Name: ______________________________

Please circle your TA's name and recitation:
Chris Matt Nati Nitin Rachel Shane

10am 11am 12pm 1pm 2pm

Problem 1. True or False, and Justify [24 points] (6 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

$$T(n) = T(n/99) + T(n/98) + n!$$

is $T(n) = 2^{\Theta(n \log n)}$.

Solution: True. We analyze the recurrence in two steps: first, note that $T(n) \geq T(n/99) + T(n/99) + n! = 2T(n/99) + n!$. By the Master Theorem, this implies $T(n) = \Omega(n!)$. Similarly, $T(n) \leq 2T(n/98) + n!$, so $T(n) = O(n!)$. Combining, we get $T(n) = \Theta(n!)$. (This conclusion was worth two points.)

Next, recall that $\log(n!) = \Theta(n \log n)$, implying $n! = 2^{\Theta(n \log n)}$. Since $T(n) = \Theta(n!)$, $T(n) = 2^{\Theta(n \log n)}$.

Common mistakes: many students wrote $n! = \Theta(n \log n)$, which is wrong; $n!$ is super-exponential!

T F Radix sort works correctly even if we start sorting from the most significant digit.

Solution: False. To prove an algorithm false, it suffices to show a single counterexample. Let's radix-sort the two numbers 19 and 91. After sorting on the most significant digit, their order is 19, 91. After sorting on the next digit, their order is 91, 19. The algorithm terminates, and the numbers are not in sorted order, therefore the algorithm is incorrect.

Common mistakes: many students explained how sorting is possible from the most significant digit, as on the problem set. This is true, but that algorithm is not radix sort. Others said that sorting from the most-significant digit is not "stable" - in fact, stability is a property of count-sort, and refers only to whether equal elements are in the same order both before and after sorting. The stability of count-sort is maintained, but is irrelevant to the question.

T F Sorting 3 elements with a comparison sort requires at least 4 comparisons in the worst case.

Solution: False. We can give an algorithm which will sort 3 elements in only 3 comparisons. To sort 3 elements: compare the first two (first comparison), if the first is larger than the second, swap them. Then compare the larger one with the third element (second comparison), if the third element is larger than we are done. If not, swap it with the second element and compare to the first element (third comparison). If it is smaller swap again and return, otherwise just return.

T F Checking if there is a pair of non-equal elements in an array with n numbers requires $\Omega(n \log n)$ time.

Solution: False. This can be done in linear time by simply comparing the first element to the successive $n-1$ elements in the array. If there is a pair of non-equal elements in the array then the first element is not equal to some other element in the array, if an unequal element is found return true, else return false.

Common Mistakes: Many students believed that you had to sort in order to do solve this problem, which, as shown above, is not true. In addition, many students thought that you could sort in linear time using counting or radix sort. This was not true because we don't have any knowledge of the nature of the elements in the array. They could all be irrational numbers in which case neither radix nor counting sort would work.

T F Every array sorted in decreasing order is also a Max-Heap.

Solution: True. For an array to be a Max-Heap, we require that any parent be greater than its children, i.e., $A[i] \geq A[2i], A[2i+1]$, which is clearly true for an array sorted in decreasing order.

T F If we hash n elements into an array of size m (using a hash function chosen uniformly at random from the set of all functions, as in the lecture), then for each array location the expected number of elements falling into it is n/m.

Solution: True. The probability of a key x getting mapped to location i is $\frac{1}{m}$ for this family (there are m^n functions in total and exactly m^{n-1} of them map x to i). Let X be the random variable representing the number of keys hashed to location i. Then $X = \sum_x X_x$ where X_x is an indicator random variable that is 1 if x gets hashed to i and 0 otherwise. It follows from the linearity of expectation that $E[X] = \sum_x E[X_x] = n/m$. Less formal solutions, relying on arguements of symmetry, got full credit as well.

Common mistakes: Some students tried to prove/disprove this fact using the notion of universality, which is not relevant for this problem. Arguements to the effect that this family is not universal and hence the statement is false got no credit. Students who saw that the family was universal but tried to use the universality condition to prove the statement got partial credit. Another common mistake was to argue that a "bad" hash function could hash everything to a single slot and so the statement was false. Remember we are talking about the expectation taken over all functions in the family, not any single function.

Problem 2. Median of a Read-Only Array [32 points]

Assume you are given an array $A[1, \ldots, n]$ of **distinct** numbers, and your goal is to find the median of those numbers. However, the array A is stored in read-only memory and you cannot modify it. At the same time, your computer is an old ZX 80 with only *constant* (i.e. $O(1)$) number of read/write memory cells. The aim of this problem is to show that there is a randomized algorithm that always finds the median of A and, with probability $> \frac{1}{2}$, terminates in time $O(n \log n)$.

You can use your $O(1)$ memory cells to store array indices, array elements and so on. Also, can use any facts/theorems from the lecture notes without reproving them.

The problem is broken into several parts. Your answers may assume the algorithm as required in previous parts (even if you were not able to find them).

(a) (8 points) Give a linear time algorithm that, given numbers x and y with $x < y$, counts the number of elements of A that are between x and y (i.e., number of indices i such that $x \leq A[i] \leq y$). Your algorithm may only use $O(1)$-space.

Solution 1:

Maintain a counter "count" and an iteration counter i. Run through the array A (i.e., for i = 1 to n do), checking if $x \leq A[i] \leq y$, and if so increment count. Output count at the end.

Obviously, the algorithm runs in $O(n)$ time and uses only 2 extra counters and thus runs in $O(1)$ space.

Scoring: If your solution uses $O(n)$ space, then you get zero points. Most other solutions were correct, though many included pseudocode only, with no final English summary. One point was taken off for lack of "clarity" in such cases.

(b) (8 points) Give a linear time algorithm that, given numbers x and y with $x < y$, picks a uniformly random element of A with value between x and y.

(E.g., if $A = \langle 7, 3, 2, 5, 1, 6 \rangle$ and $x = 2$ and $y = 5$ then each of the numbers 2, 3, 5 should be output with equal probability.)

There were several correct solutions in this case.

Solution 1: Count the number of elements between x and y using Part (a). Call this number k. Pick r at random between 1 and k. Now run through the array and output the rth element of A that lies between x and y. Pseudocode below:

```
k ← Count-Between[A,x,y];
r ← Random[1 .. k];
j ← 0;
For i ← 1 to n do
        If x ≤ A[i] ≤ y then j++;
        If j = r then Return(A[i]);
Endfor
```

Note the solution does output a uniformly random element, and runs in $O(n)$ time with $O(1)$ space, as required.

Solution 2: A more complicated implementation of the above was given by some, who didn't pick r at random initially, but instead maintained a current winner and then looked for the next (the jth) element between x and y. When such a new element was found, a new contest was played between the old winner and the newly found element, with the probability of the old winner continuing being $j - 1/j$ and the new one winning being $1/j$. Finally the algorithm returned the winner at the end.

Solution 3: An extremely elegant solution to this problem is: Pick a random element of the array and output it if it lies between x and y. This solution is corrrect and runs in expected linear time, provided we first check Between[x,y] to ensure at least one element exists between x and y. To analyse the expected running time, note that any one iteration takes $O(1)$ time (say c), and has a probability of success at least $1/n$. Thus if X denotes the expected running time, then

$$X = 1/n.c + (1 - 1/n)(c + X).$$

Solving the above gives $X = cn$.

Scoring: People who gave just the pseudocode of Solution 1 (or 2) above or the English text got eight points. If you gave solution 3, but did not give some explanantion for the expected running time you got about 4 points.

(c) (12 points) Using the algorithms from the previous parts as subroutines, give an algorithm to find the median of A using constant additional space.

Solution: This is a simple divide and conquer strategy.

First compute Max and Min of the array A in linear time and $O(1)$ space. Then maintain limits L and U for the median. Initially $L =$ Min and $U =$ Max. Then iterate as follows: Pick a random element R between U and L. Count-Between[Min,R] and if this count is floor(n/2) then return R, else if this count is less than floor(n/2) then repeat with L set to R, else repeat with U set to R.

Scoring: If you got the basic idea correct and remembered to make recursive calls properly then you get full points. Many people didn't track Min and L separately as required above.

An alternate solution was to keep track of the rank of the element one wishes to find and to modify this rank when setting L to R in the above iterations.

(d) (4 points) Briefly justify correctness and running time.

Solution: When the algorithm does terminate, it terminates outputting the median. The space usage is also only $O(1)$. The tricky part is arguing the expected running time bound.

First note that every iteration runs in linear time. What we wish to prove is that the expected number of iterations is $O(\log n)$.

The main idea is to use the notion of "lucky" splits as in lectures. We call an iteration lucky if the number of elements between L and U at the end of the iteration is at most $3/4$ times the number of elements between L and U at the beginning of the iteration. As shown in lecture, the probability of a lucky split is at least $1/4$. Thus (again from lecture) the probability that, say $10 \log n$, iteration don't lead to at least $\log_{4/3} n$ lucky splits can be upper bounded by $1/2$. Thus in $O(\log n)$ iterations we expect to find the median with probability at least $1/2$.

Scoring: 1 point for correctness justification (almost everyone with a correct algorithm got this). 3 points for the expected running time analysis.

Incidentally several people wrote

"$T(n) \leq T(n/2) + O(n) \quad \Rightarrow \quad T(n) = O(n \log n)$".

The statement is perfectly correct and highly inappropriate!!

Problem 3. Set Sum [24 points]

Design an algorithm which, given a set $S \subset \{1 \ldots m\}$, enumerates all numbers of the form $x + y$ for $x, y \in S$, in time $O(m^{\log_2 3})$.

Hint: use polynomial multiplication.

Solution: Given an array $S[1..n]$ of numbers in the range $\{1 \ldots m\}$, create a polynomial $P(t) = \sum_i t^{S[i]}$. Multiply the polynomial by itself (using the divide and conquer algorithm described in class) to get the polynomial $Q = P^2$. Output exponents that have non-zero coefficients in Q.

Correctness: The basic idea is that when multiplying polynomials the exponents add up. If $x, y \in S$ then P will have terms t^x and t^y, which when multiplied together yield t^{x+y}. Since all the coefficients of P are non-negative, t^{x+y} will have a coefficient of at least 1 in Q (the coefficient might be higher, if there are other ways to get the same sum—in fact, if $x \neq y$ it will be at least two). Similarly, if the term ct^z appears in Q, for a non-zero c, there must have been two terms that multiplied out to t^z, and so P must have some term t^x and t^y such that $z = x + y$, and S contains these x and y.

Running time: Creating a polynomial from the set requires one pass over the set, and so linear time. Extracting exponents with non-zero coefficients also requires one pass, and linear time. All the work is done by the polynomial multiplication. Since the elements in S are between 1 and m, P will be of degree at most m. Multiplying polynomials of degree at most m takes $O(m^{\log_2 3})$ time.

Scoring: The correctness proof need not be as detailed as given here. Most people who got the basic idea, weather they wrote it in English or pseudo-code, got all the points. In some cases a few points were taken off for a "confused" solution that did not explicitly specify the algorithm. Another reason for loosing a few points was extracting only exponents with a coefficient of one (as discussed above, the coefficients will generally be larger that one).

People who understood the need to represent S as a polynomial, but tried a representation that doesn't work (e.g. $P(t) = \sum_i S[i]t^i$) got 3–6 points. A common mistake was to represent the complete set $\{1, \ldots, m\}$, ignoring the input S.