Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- **For this quiz, you *need not* provide rigorous proofs of correctness. Instead, give informal arguments for why you believe your algorithms are correct. Pseudocode is only required when explicitly indicated, but you may include it if it clarifies your answers.**
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains 5 multi-part problems. You have 80 minutes to earn 80 points.
- This quiz booklet contains 11 pages, including this one. Two extra sheets of scratch paper are attached. Please detach them before turning in your quiz.
- This quiz is closed book. You may use one handwritten A4 or $8\frac{1}{2}'' \times 11''$ crib sheet. No calculators or programmable 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. Do not put part of the answer to one problem on the back of the sheet for another problem, since the pages may be separated for grading.
- Do not waste time and paper rederiving 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.
- 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!

Problem	Points	Grade	Initials
1	12		
2	12		
3	30		
4	13		
5	13		
Total	80		

Name: Solutions _____

Circle the name of your recitation instructor:

Moses Jen Steve

Problem 1. Recurrences [12 points]

Solve the following recurrences. Give tight, i.e. $\Theta(\cdot)$, bounds.

(a) $T_1(n) = 5\,T_1(n/2) + \sqrt{n}$

Solution: We use the Master Theorem: note that $\log_2 5 > 1/2 + \epsilon$, so the solution is $T_1(n) = \Theta(n^{\log_2 5})$.

(b) $T_2(n) = 64\,T_2(n/4) + 8^{\lg n}$

Solution: First, notice that $8^{\lg n} = n^{\lg 8}$ (this can be seen by taking $\lg$ of both sides). Since $\log_4 64 = \lg 8 = 3$, we use case (ii) of the Master Theorem: $T_2(n) = \Theta(n^3 \log n)$.

(c) $T_3(n) = 2\,T_3(3n/8) + T_3(n/4) + 6n$

Solution: We solve this recurrence by Akra-Bazzi, with $p = 1$. Then we get

$$T_3(n) = \Theta(n\left(1 + \int_1^n \frac{6x}{x^2}\,dx\right)) = \Theta(n(1 + 6\ln n)) = \Theta(n \log n).$$

Problem 2. Short Answer [12 points]

Give *brief*, but complete, answers to the following questions.

(a) Briefly describe the difference between a ***deterministic*** and a ***randomized*** algorithm, and name two examples of algorithms that are not deterministic.

Solution: On identical inputs, a deterministic algorithm always performs exactly the same computations and returns the same output. A randomized algorithm is one which "flips coins," i.e. one which makes random choices that may cause it to perform different computations, even on the same input. RANDOMIZED-QUICKSORT and RANDOMIZED-SELECT are two examples of non-deterministic algorithms.

(b) Describe the difference between ***average-case*** and ***worst-case*** analysis of deterministic algorithms, and give an example of a deterministic algorithm whose average-case running time is different from its worst-case running time.

Solution: An average-case analysis assumes some distribution over the inputs (e.g., uniform), and computes the expected (average) running time of an algorithm subject to that distribution. A worst-case analysis considers those inputs which force an algorithm to run for the longest amount of time, and computes the running time under those inputs. QUICKSORT has a worst-case running time of $\Theta(n^2)$ (on an already-sorted or reverse-sorted array), but has an average-case running time of $\Theta(n \log n)$ (assuming all input permutations are equally likely).

(c) If you can multiply 4-by-4 matrices using 48 scalar multiplications, can you multiply $n \times n$ matrices asymptotically faster than Strassen's algorithm (which runs in $O(n^{\lg 7})$ time)? Explain your answer.

Solution: Our algorithm breaks an $n \times n$ matrix into a 4-block by 4-block matrix (where each block is $n/4 \times n/4$). It then multiplies the appropriate blocks using 48 recursive calls (corresponding to the scalar multiplications) and combines their products. Our new algorithm's running time is $T(n) = 48T(n/4) + \Theta(n^2)$, which is $O(n^{\log_4 48})$ by the Master Theorem. For comparison with Strassen's algorithm, note that $\log_2 7 = \log_{2^2} 7^2 = \log_4 49$. Therefore our algorithm is asymptotically better.

Problem 3. True or False, and Justify [30 points]

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 Every comparison-based sort uses at most $O(n \log n)$ comparisons in the worst case.

Solution: False. INSERTION-SORT, for example, uses $\Theta(n^2) \neq O(n \log n)$ comparisons in the worst-case (a reverse-sorted array). The statement would be true if it read "*. . . at least* $\Omega(n \log n)$ comparisons in the worst case."

(b) T F RADIX-SORT is stable if its auxiliary sorting routine is stable.

Solution: True. If two numbers are equal, then they have the same digits. Each intermediate sort is stable, so the two equal numbers never change relative positions.

(c) T F It is possible to compute the smallest $\sqrt{n}$ elements of an n-element array, in sorted order, in $O(n)$ time.

Solution: True. We can SELECT the $\sqrt{n}$th smallest element and partition around it, then sort those $\sqrt{n}$ elements in $O(n)$ time. Alternately, we can build a min-heap in $O(n)$ time and call EXTRACT-MIN $\sqrt{n}$ times, for a total runtime of $O(n + \sqrt{n} \log n) = O(n)$.

Some incorrect solutions amounted to: "we must do $\sqrt{n}$ order statistic queries, each of which take $O(n)$ time, for a total running time of $O(n\sqrt{n})$." However, this argument does not preclude us from coming up with a more clever algorithm (like the one above) that is more efficient. In fact, a similar argument would "prove" that sorting must take $\Omega(n^2)$ time (despite the existence of MERGESORT etc.), because we must do n order statistic queries!

(d) T F Consider hashing the universe $U = \{0, \ldots, 2^r - 1\}$, $r > 2$, into the hash table $\{0, 1\}$. Consider the family of hash functions $\mathcal{H} = \{h_1, \ldots, h_r\}$, where $h_i(x)$ is the ith bit of the binary representation of x. Then $\mathcal{H}$ is universal.

Solution: False. Take $x = 0, y = 1$. Then all of x's binary digits are the same as y's, except for the least significant one. Thus $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = (r-1)/r > 1/2$. If $\mathcal{H}$ were universal, the probability would be at most $1/2$.
Some incorrect solutions said that the probability of a collision, taken over random choices of h, x, and y is 1/2. This is true, but the universality condition demands something stronger: it says that for *every* fixed (distinct) pair x, y, their probability of collision over the random choice of h must be at most 1/2.

(e) T F RANDOMIZED-SELECT can be forced to run in $\Omega(n \log n)$ time by choosing a bad input array.

Solution: False. RANDOMIZED-SELECT runs in expected $O(n)$ time; the only way it can take longer is if its random choices of pivots are unlucky. The input array cannot force these unlucky choices.

(f) T F For every two functions $f(n)$ and $g(n)$, either $f(n) = O(g(n))$ or $g(n) = O(f(n))$.

Solution: False. Let $f(n) = \sin n$ and $g(n) = \cos n$; then neither case holds. Another example is $f(n) = \sqrt{n}$ and $g(n) = n^{\sin n}$. Finally, one could let $f(n)$ and $g(n)$ be any strictly-negative functions; by a technical condition of the definition, $f(n)$ must be at least 0 to be $O(g(n))$.
Many incorrect answers argued that one of the statements $f(n) \leq cg(n)$, $f(n) = cg(n)$, or $g(n) \leq cf(n)$ must be true. This is correct for any *particular* value of n, but it doesn't mean that the *same* statement is true for *all* sufficiently large values of n, which is the condition needed in the definition of big-O.

(g) T F Suppose that we have a hash table with $2n$ slots, with collisions resolved by chaining, and suppose that $n/2$ keys are inserted into the table. Each key is equally likely to be hashed into each slot (***simple uniform hashing***). Then the expected number of keys for each slot is $1/4$.

Solution: True. Define X_j (for $j = 1, \ldots, n/2$) to be the indicator which is 1 if element j hashes to slot i, and 0 otherwise. Then $E[X_j] = \Pr[X_j = 1] = 1/2n$. Then the expected number of elements in slot i is $E[\sum_{j=1}^{n/2} X_j] = \sum_{j=1}^{n/2} E[X_j] = n/4n = 1/4$ by linearity of expectation.

(h) T F The following array A is a max-heap:

$$30 \;\; 25 \;\; 7 \;\; 18 \;\; 24 \;\; 8 \;\; 4 \;\; 9 \;\; 12 \;\; 22 \;\; 5$$

Solution: False. See that $7 = A[3] < A[2 \cdot 3] = 8$, which is a violation of the max-heap property.

(i) T F Suppose we use HEAPSORT instead of INSERTION-SORT as a subroutine of BUCKET-SORT to sort n elements. Then BUCKET-SORT still runs in average-case linear time, but its worst-case running time is now $O(n \log n)$.

Solution: True. Even if all the elements land in the same bucket (the worst-case input), HEAPSORT sorts them in $O(n \log n)$ time.

(j) T F If memory is limited, one would prefer to sort using HEAPSORT instead of MERGESORT.

Solution: True. MERGESORT is not in-place, which means it requires an auxiliary array as big as the input. HEAPSORT is in-place, which means it only uses $O(1)$ auxiliary space.

Problem 4. Perfect Powers [13 points]

You have just discovered a breakthrough result in number theory that will quickly become world-famous, but there is one step left for you to complete. You need to find a fast algorithm to tell whether a number X is a perfect power. That is, you want a fast algorithm which, on an input integer X that is n bits long, finds whether there exist integers $B \geq 2$ and $e \geq 2$ such that $X = B^e$. If so, your algorithm should output the values of B and e.

(a) If $X = B^e$ is a perfect power, how large can e be? Express your answer as a function of n.

Solution: (First, some trivia: the scenario from this problem is entirely real! The ground-breaking deterministic primality-testing algorithm by Agrawal et al, discovered with great fanfare this past summer, performs the perfect-power test as one of its first steps.)
[3 points] Note that $n \geq \lg X = e \lg B$, and since $B \geq 2$, $e \leq n$.
A common error was to write only $e \leq \frac{n}{\lg B}$, which is insufficient because we asked for e *as a function of* n, and also because the value of B (if any) is not known in advance.

(b) Suppose that your computer had an $O(1)$-time operation $\textsc{Root}(M, r)$ that returns the rth root of an integer M, if that root is an integer (and returns $\perp$ otherwise). Give an algorithm to solve the perfect-power problem and analyze its running time.

Solution: [5 points] We simply test all values of e up to n: for $e = 1, \ldots, n$, if $\textsc{Root}(X, e) = B$ where B is an integer, then return the pair (B, e). If no such root is an integer, return $\perp$. The running time of this procedure is $O(n)$.
Some solutions looped over all possible values of *both* B and e, and only returned if $B = \textsc{Root}(X, e)$. This solution is wasteful, because $\textsc{Root}$ returns the proper base, so there is no need to guess it in advance. This solution is also too inefficient, because B could be as large as $\sqrt{X} \approx 2^{n/2}$, so an exponential number of iterations would be peformed.

(c) In reality, there is no such $O(1)$-time ROOT procedure. Still it is possible to solve the perfect power problem in $O(n^2 \log n)$ by using a suitable algorithm for ROOT. Describe such an algorithm and analyze its running time. You may assume that multiplying two integers takes $O(1)$ time (no matter how large they are).

Solution: [5 points] We will implement a ROOT procedure that runs in time $O(n \log n)$ when used in the above algorithm, so that we solve the perfect power problem in $O(n^2 \log n)$ time. On input (M, r), our ROOT procedure performs a binary search for the rth root of M in the range $[2 \,..\, M]$. When testing the midpoint m of the range, if $m^r < M$ then we recursively search the upper half of the range; if $m^r > M$ then we search the lower half; if $m^r = M$ then we return m as the rth root of M.

In our usage of ROOT from the previous part, $M = X \leq 2^n$ and $r \leq n$. Computing each m^r takes $O(\log r) = O(\log n)$ mulitiplications by the repeated-squaring technique from class, and the binary search does $O(\log M) = O(n)$ iterations, each with one exponentiation, for the claimed runtime of $O(n \log n)$.

Many solutions did a brute-force search for B, but this is too inefficient. In the worst case, $r = 2$, so all values of B from 1 to $\sqrt{M} = \sqrt{X} = 2^{n/2}$ would have to be checked, which is an exponential number of tests. All exponential-time algorithms received no points for this part.

Other incorrect solutions made mathematical errors: assuming that the rth root of M must be smaller than $\log_r M$, or returning $\log_r M$ by repeatedly dividing M by r or multiplying r by itself (this treats r as the base, instead of its proper role as the exponent).

Problem 5. Assigning Grades [13 points]

It is the not-too-distant-future, and you are a computer science professor at a prestigious north-eastern technical institute. After teaching your course, "6.66: Algorithms from Hell," you have to assign a letter grade to each student based on his or her unique total score. (Scores can only be compared to each other.) You are grading on a curve, and there are a total of k different grades possible. You want to rearrange the students into k equal-sized groups, such that everybody in the top group has a higher score than everybody in the second group, etc. However, you don't care how the students are ordered within each group (because they will all receive the same grade).

(a) Describe and analyze a simple algorithm that takes an unsorted n-element array A of scores and an integer k, and divides A into k equal-sized groups, as described above. Your algorithm should run in time $O(nk)$. (If you find a faster algorithm, see part **(c)**.) You may assume that n is divisible by k. *Note*: k is an input to the algorithm, not a fixed constant.

Solution: [5 points] Our algorithm first uses SELECT to find the n/kth order statistic, then partitions around it. At this point, the first n/k elements of the array form the bottom group. Then it uses SELECT to find the n/kth order statistic of the remainder of the array, and partitions around it, etc., until all the groups have been separated. Each SELECT and PARTITION requires linear time in the number of remaining elements, which is at most n, so the running time is $O(nk)$.

(b) In the case that $k = n$, prove that any algorithm to solve this problem must run in time $\Omega(n \log k)$ in the worst case. Recall that we are only considering comparison-based algorithms, i.e., algorithms that only compare scores to each other as a way of finding information about the input. *Hint:* There is a very short proof.

Solution: [3 points] Any algorithm for this problem can fully sort an array of n elements if we provide it with an input where $k = n$. Since sorting requires $\Omega(n \lg n)$ comparisons in the worst case, the algorithm must run in time $\Omega(n \lg n) = \Omega(n \lg k)$ in the worst case.

(c) Now describe and analyze an algorithm for this problem that runs in time $O(n \log k)$. You may also assume that k is a power of 2, in addition to assuming that n is divisible by k.

Solution: [5 points] We use a recursive algorithm GROUP, which takes an array and a value k, and works as follows: if $k = 1$, return. Otherwise, SELECT and PARTITION around the median of the array. Then call GROUP on the lower half of the array with $k/2$, and again on the top half with $k/2$.

To see that this works, note that after partitioning, all grades in the upper half of the array are greater than those in the lower half. By induction, the two recursive calls divide each half into $k/2$ groups, for a total of k groups. Finally, note that the base case satisfies the problem statement.

We now analyze the running time: the recurrence describing the algorithm's running time is $T(n, k) = 2T(n/2, k/2) + \Theta(n)$ because SELECT and PARTITION are linear-time. The base case of the recurrence is $T(n, 1) = \Theta(1)$ for any n. Therefore the recurrence tree does $\Theta(n)$ work at each level, and has $\lg k$ levels, for a total running time of $\Theta(n \log k)$.

Some students correctly observed that this solution is essentially an "early quitting" QUICKSORT, where the pivot is always chosen to be the median, and the algorithm terminates once the recursion depth reaches $\lg k$.

SCRATCH PAPER — Please detach this page before handing in your quiz.

SCRATCH PAPER — Please detach this page before handing in your quiz.