Problem 1. Recurrences

Solve the following recurrences by giving tight $\Theta$-notation bounds.

(a) $T(n) = 3T(n/5) + \lg^2 n$

**Solution:** By Case 1 of the Master Method, we have $T(n) = \Theta(n^{\log_5 3})$.

(b) $T(n) = 2T(n/3) + n \lg n$

**Solution:** By Case 3 of the Master Method, we have $T(n) = \Theta(n \lg n)$.

(c) $T(n) = T(n/5) + \lg^2 n$

**Solution:** By Case 2 of the Master Method, we have $T(n) = \Theta(\lg^3 n)$.

(d) $T(n) = 8T(n/2) + n^3$

**Solution:** By Case 2 of the Master Method, we have $T(n) = \Theta(n^2 \log n)$.

(e) $T(n) = 7T(n/2) + n^3$

**Solution:** By Case 3 of the Master Method, we have $T(n) = \Theta(n^3)$.

(f) $T(n) = T(n-2) + \lg n$

**Solution:** $T(n) = \Theta(n \log n)$. This is $\sum_{i=1}^{n/2} \lg 2i = \sum_{i=1}^{n/2} \lg i \geq (n/4)(\lg n/4) = \Omega(n \lg n)$. For the upper bound, note that $T(n) \leq S(n)$, where $S(n) = S(n-1) + \lg n$, which is clearly $O(n \lg n)$. 


Problem 2. True or False, and Justify

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 If \( f(n) \) does not belong to the set \( o(g(n)) \), then \( f(n) = \Omega(g(n)) \).

   Solution: False. For example \( f(n) = n^{\sin(n)} \) and \( g(n) = n^{\cos(n)} \).

(b) T F A set of \( n \) integers in the range \( \{1, 2, \ldots, n\} \) can be sorted by RADIX-SORT in \( O(n) \) time by running COUNTING-SORT on each bit of the binary representation.

   Solution: False. This results in \( \Theta(\log n) \) iterations of counting sort, and thus an overall running time of \( \Theta(n \log n) \).

(c) T F An adversary can construct an input of size \( n \) to force RANDOMIZED-MEDIAN to run in \( \Omega(n^2) \) time.

   Solution: False. The expected running time of RANDOMIZED-MEDIAN is \( \Theta(n) \). This applies to any input.

Problem 3. Short Solution

Give brief, but complete, solutions to the following questions.

(a) Consider any priority queue (supporting INSERT and EXTRACT-MAX operations) in the comparison model. Explain why there must exist a sequence of \( n \) operations such that at least one operation in the sequence requires \( \Omega(\lg n) \) time to execute.

   Solution: Take an array \( A \) of \( n/2 \) elements. Insert the \( n/2 \) elements into the priority queue, and then extract-max \( n/2 \) times. This sorts the array à la Heapsort. If none of the operations in this sequence require \( \Omega(\lg n) \) time, then these \( n \) operations take \( o(n \log n) \) time, which is impossible in the comparison model, since we’ve shown an \( \Omega(n/2 \log(n/2)) = \Omega(n \log n) \) lower bound for any sorting algorithm.

(b) Suppose that an array \( A \) has the property that only adjacent elements might be out of order—i.e., if \( i < j \) and \( A[i] > A[j] \), then \( j = i + 1 \). Which of INSERTION-SORT or MERGE-SORT is a better algorithm to sort the elements of \( A \)? Justify your choice.

   Solution: INSERTION-SORT is a better choice: since its running time depends on the number of inversions in the array. We may swap each element with the element immediately to its left, but that’s all that we do in INSERTION-SORT. So INSERTION-SORT is \( O(n) \) in this case, while MERGE-SORT remains \( \Theta(n \log n) \).
(c) Recall from lecture that for \( n \times n \) matrices \( A, B, \) and \( C, \) we can probabilistically test whether \( AB = C \) by choosing a random 0-1 vector \( x \) and checking whether \( ABx = Cx. \) In lecture we showed a lower bound of \( p \geq 1/2 \) on the probability \( p \) that this test is successful as follows. For a matrix \( M, \) let \( M_{ij} \) denote the \( ij \)th element of \( M. \) We argued that if \((AB)_{ij} \neq C_{ij}, \) then at most one of the two values for \( x_i \) can make the equation
\[
\sum_{k=1}^{n} (AB)_{kj}x_k = \sum_{k=1}^{n} C_{kj}x_k
\]
hold.

Suppose that instead we choose a random vector \( x \) where each element of \( x \) is chosen independently at random from the set \( \{0, 1, 2, 3\} \) instead of from \( \{0, 1\}, \) and perform the same test. What lower bound on \( p \) can be deduced in this case?

**Solution:** The probability of success was 1/2 if \( AB \neq C, \) since at most one of the two values for \( x_i \) can make \((AB)^{(j)}x = C^{(j)}x \) if \((AB)_{i,j} \neq C_{i,j}. \) So, similarly, at most one of the four values for \( y_i \) can do the same, so the same analysis shows that the probability of success is at least 3/4. In either case, if \( AB = C, \) then we will always produce the correct solution.
Problem 4. 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 $T^m_i = T[i \ldots i + m - 1]$ of $T$, we say that $T^m_i$ is an anagram of $P$ if there is a way of permuting symbols in $T^m_i$ so that the resulting array is equal to $P$.

(a) Give an algorithm that, given an index $i$ (and $m$), determines whether $T^m_i$ 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.

Solution: The two $m$-element sequences $P$ and $T^m_i$ 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 $T^m_i$, 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 $\Theta(m)$.

(b) Design an algorithm, which given $T$ and $P$ as an input, reports all $i$’s such that $T^m_i$ 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.

Solution: Start with $i = 1$. Let us count the number of occurrences of each letter in $P$ and in $T^m_i$, 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 occurrences of each letter in $T^m_i$. Construction of the arrays $C$ and $D$ will consume $\Theta(m)$ time, as it requires a single linear scan over $P$ and $T^m_i$. 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 $T^m_i$.

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 $T^m_i$ 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 $T^m_{i+1}$) and by incrementing $D[T[i + m]]$ (since $T[i + m]$ will now be present in $T^m_{i+1}$). 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 $\Theta(m + n)$. 

Problem 5. Mean 6.042 instructors

There are two types of professors who have taught 6.042: nice professors and mean professors. The nice professors assign A’s to all of their students, and the mean professors assign A’s to exactly 75% of their students and B’s to the remaining 25% of the students. For example, for $n = 8$, the arrays $[A A A B A B A A]$ and $[A B B A A A A A]$ represent grades assigned by mean professors, and the array $[A A A A A A A A]$ represents grades assigned by a nice professor. Given an array $G[1 . . n]$ of grades from 6.042, we wish to decide whether the professor who assigned the grades was nice or mean.

Give an efficient randomized algorithm to decide whether a given array $G$ represents grades assigned by a mean or nice professor. Your algorithm should be correct with probability at least 51%.

Solution: Here’s the algorithm:

1. Repeat the following $t$ times:
   (a) choose a random index $i \in \{1, \ldots, n\}$.
   (b) if $G[i] = B$, then return “mean”

2. Return “nice”.

First, note that with probability one, a nice professor will be noted as such, since there are no $B$’s in $G$ in this case. The probability that a mean professor makes it through an iteration of the loop is $3/4$, since $1/4$ of the entries of array are $B$’s. The probability that the mean professor makes it through $t$ iterations is $(3/4)^t$. If we take $t$ to be 3, then $(3/4)^3 < 0.49$. Thus the probability of giving a correct solution is at least 51%.

The running time is obviously $O(1)$, since we do a constant number of iterations of constant-time loop.

Comment: The above arguments show that

- If a professor is nice, the probability of an incorrect solution is 0
- If a professor is mean, the probability of an incorrect solution is $\leq 49$

However, in several cases, people were making “reverse” statements, e.g.:

"If we get an A, then the probability that the professor is mean is ..."

Although intuitive, a closer inspection reveals that such a statement does not have any real meaning. This is because there is no probability distribution determining if the professor is mean or not. In case of randomized algorithms, the input is not random. Instead, for any input, the algorithm should report the correct solution with at least certain probability.