CS 141, Spring 2019 Midterm 1/B

Problem 1. (15 points [writing/solving recurrence relations])

Solve exactly (that is, without using any asymptotic notation) the following recurrence relation by iterative substitutions

$$T(n) = \begin{cases} 1 & n = 1 \\ T\left(\dfrac{n}{9}\right) + \sqrt{n} & n > 1 \end{cases}$$

Answer: We have

$$\begin{aligned}
T(n) &= T(n/9) + \sqrt{n} \\
&= T(n/9^2) + \sqrt{n}\,(1/3 + 1) \\
&= T(n/9^3) + \sqrt{n}\,(1/9 + 1/3 + 1) \\
&\dots \\
&= T(n/9^i) + \sqrt{n}\left(1/3^{i-1} + 1/3^{i-2} + \ldots + 1/3^1 + 1/3^0\right) \\
&= T(n/9^i) + (3/2)\sqrt{n}\left(1 - 1/3^i\right)
\end{aligned}$$

now we set $n/9^i = 1$ which is $i = \log_9 n$ and we get

$$\begin{aligned}
T(n) &= T(1) + (3/2)\sqrt{n}\left(1 - 1/3^{\log_9 n}\right) \\
&= 1 + (3/2)\sqrt{n}\,(1 - 1/\sqrt{n}) \\
&= \frac{3\sqrt{n} - 1}{2}
\end{aligned}$$

Problem 2. (15 points [writing/solving recurrence relations])

Using the Master method, give an asymptotic tight bound for $T(n)$ in the following recurrence relation

$$T(n) = \begin{cases} 1 & n = 1 \\ 2\,T\left(\dfrac{n}{4}\right) + \sqrt{n}\log n & n > 1 \end{cases}$$

Answer: We have $a = 2$, $b = 4$, $f(n) = \sqrt{n}\log n$. We also have $n^{\log_b a} = n^{\log_4 2} = \sqrt{n}$. Clearly, $f(n) \in \Theta(\sqrt{n}\log^k n)$ for $k = 1$. The second case of the Master Theorem applies, hence $T(n) \in \Theta(\sqrt{n}\log^2 n)$.

Problem 3. (15 points [writing/solving recurrence relations])

In the algorithm SELECT described in class (linear-time selection), the input elements are divided into $\lceil n/5 \rceil$ groups of 5. Suppose you modify the algorithm to divide the input elements into $\lceil n/3 \rceil$

groups of 3 instead. Let $T(n)$ denote the worst-case running time of the modified algorithm as a function of the input size n. Write a recurrence relation for $T(n)$, but do NOT solve it.
Answer: For groups of 3, the number of elements greater than x (and the number of element less than x) is at least

$$2\left(\lceil\frac{1}{2}\lceil\frac{n}{3}\rceil\rceil - 2\right) \geq \frac{n}{3} - 4$$

and the recurrence relation becomes

$$T_3(n) \leq T_3\left(\lceil n/3\rceil\right) + T_3(2n/3 + 4) + O(n)$$

Problem 4. (20 points [divide & conquer])

An array A is said to have a *majority element* if half or more than half of the entries in A are exactly the same. Given an unsorted array $A[1\ldots n]$ of n items (where n is a power of two) we want to determine whether A has a majority element, and if so, return such an item. Consider the following divide and conquer algorithm for this problem.

Algorithm FIND-MAJORITY (A : **array**)
$n \leftarrow |A|$
if $n \leq 1$ **then return** $A[1]$
else
  $a_1 \leftarrow$ FIND-MAJORITY($A[1,\ldots,n/2]$)
  $a_2 \leftarrow$ FIND-MAJORITY($A[n/2+1,\ldots,n]$)
  if the number of times a_1 appears in A is $\geq n/2$ **then return** a_1
  else if the number of times a_2 appears in A is $\geq n/2$ **then return** a_2
  else return NULL

Is this algorithm correct i.e., does FIND-MAJORITY always return the majority element, if A has one in it (or NULL otherwise)? Give a counterexample if your answer is "No", a brief argument of correctness (e.g., proof by induction) if your answer is "Yes". You can assume n to be a power of 2.
Answer: The algorithm is incorrect. The simplest counter-example is $A = \{1, 2, 3, 2\}$, $n = 4$. From the recursive call on the first half of the array $\{1, 2\}$, we will get $a_1 = 1$. From the recursive call on the second half of the array $\{3, 2\}$, we will get $a_2 = 3$. Neither a_1 or a_2 occur at least $n/2 = 2$ times, so the algorithm will return NULL which is incorrect, since number 2 is a majority element in A.

Problem 5. (15 points [divide & conquer])

Write the pseudo-code for the $O(n \log n)$-time algorithm for closest pair that we described in class.
Answer: See slides.

Problem 6. (20 points [divide & conquer])

Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements. Describe a divide-and-conquer algorithm that takes $O(kn \log k)$ time. Make sure you explain why your algorithm runs in $O(kn \log k)$ time.

Hint: Use as a sub-routine ("black-box") the MERGE algorithm we described in class to merge two sorted arrays with n elements in $O(n)$-time.

1. Divide the arrays into two sets of $k/2$ arrays, where each array is of size n.
2. Recursively merge the first set of arrays, giving an array of size $nk/2$.
3. Recursively merge the second set of arrays, giving another array of size $nk/2$.
4. Merge the two resulting arrays (of size $nk/2$ each) to get the result.

The total time $T(n, k)$ satisfies the recurrence $T(n, k) = 2T(n, k/2) + nk$, and $T(n, 1) = 1$.

In the recursive calls, n does not change.

At the ith level in the recursion tree, there are 2^i subproblems, each with a set of $k/2^i$ arrays, each taking time $nk/2^i$. The total time at level i is $2^i nk/2^i = nk$.

There are $\log_2 k$ levels. Thus, the total time is $O(nk \log k)$.