HW 1: Data structures, recurrences, divide and conquer

Disclaimer. These are not fully detailed solutions, but they are meant to give you a good enough explanation.

Question 1

The main idea is to store an additional count at each node of the tree. In the standard prefix tree data structure, we have nodes with parent and children pointers, along with a binary isWord parameter that is 1 if the string from the root to the node is a word and 0 otherwise. Now, we have an additional parameter count at each node. It will keep a count of the number of words in the sub-tree rooted at that node (including itself). The procedures are modified as follows:

- ADD(w): we increment the count parameter of all the nodes on the way. If we create new nodes, we initialize the count values to 1. Time complexity is thus $O(|w|)$.
- DELETE(w): we go down the tree as in the usual implementation to see if the word exists. If so, we set isWord to zero, and then go back up the tree and decrement all the count values by 1. The time taken is still $O(|w|)$.
- QUERY(w): no counts are changed; the procedure is the same as before (thus time is $O(|w|)$).
- COUNT-PREFIX(w): we follow the letters of w (as in the other procedures), and simply return the count value at the last node. If we are unable to follow the letters of w all the way to the end (because no words had w as a prefix and thus that portion of the tree wasn’t added), then we return 0. The time taken is clearly also $O(|w|)$.

Question 2

(a) We have

$$T(n) = 2T(n^{1/2}) + 4 = 4T(n^{1/4}) + 8 + 4 = 8T(n^{1/8}) + 16 + 8 + 4 = \ldots$$

In general, we have

$$T(n) = 2^r T(n^{1/2^r}) + 2^{r+1} + 2^r + \ldots + 2^2,$$

for any $r \geq 2$. This can be proved by induction. Now, observing that $n^{1/2^r} = 2^{\log n/2^r}$, which is 2 if $r = \log \log n$, we have (by plugging in this r into the formula above),

$$T(n) = 2^r T(2) + 2^{r+1} + 2^r + \ldots + 2^2 = O(2^r) = O(\log n).$$

(b) First, note that $T(n) \geq \sqrt{n}$ (it’s one of the terms on the RHS, after all). It’s also easy to see that $T(n) = O(n)$ (easy to see by induction). We thus use the “guess and check by induction” way of proof. Say $T(n) = An^c$, for some $c$ between 1/2 and 1. Then, consider proving the bound by induction. We would want to have

$$A(n/2)^c + A(n/3)^c + n^{1/2} \leq An^c.$$
Now, if $c$ is any constant s.t. $(1/2)^c + (1/3)^c < 1$, then we can make the above hold for an appropriately large $A$.

The value of $c$ such that $(1/2)^c + (1/3)^c = 1$ is $0.787885$ (using Wolfram Alpha). Thus we have $T(n) = O(n^{0.788})$.

(c) We have the recurrence $T(n) = 3T(n/2) + g(n)$. This can be simplified as:

$$T(n) = 3T(n/2) + g(n) = 9T(n/4) + 3g(n/2) + g(n) = \ldots = 3^rT(n/2^r) + 3^{r-1}g\left(\frac{n}{2^{r-1}}\right) + 3^{r-2}g\left(\frac{n}{2^{r-2}}\right) + \ldots g(n),$$

for any $r \geq 2$. We set $r = \log_2 n$. The first term thus becomes $O(n^{\log_2 3})$. The rest of the sum, if $g(n) = n^{1.8}$, can be seen to converge to $\Theta(n^{1.8})$ (essentially because $2^{1.8} > 3$). Thus we have $T(n) = O(n^{1.8})$. If $g(n) = n^{1.5}$, the sum is dominated by the $3^{r-1}g(n/2^{r-1})$ term, which is essentially $n^{\log_2 3} < n^{1.8}$. Thus it makes sense to improve the $g(n)$. Finally, if $g(n) = n \log n$, still the sum is dominated by the $3^{r-1}...$ term, and thus there is no asymptotic improvement.

**Question 3**

(a) Consider the array: $n, 1, 2, 3, \ldots, n - 1$. (I.e., just the first term is out of order.) In this case, bubble sort will bubble it all the way to the end, and end after one iteration of the loop. This takes time $O(n)$.

(b) Same rough idea as above. Suppose we have the array:

$$(n - \sqrt{n}), (n - \sqrt{n} + 1), \ldots, (n - 1), n, 1, 2, 3, \ldots, (n - \sqrt{n} - 1).$$

In the first iteration, the $n$ bubbles all the way to the end (rest are fixed), in the next, the $(n - 1)$ bubbles to its right position, and so on. Overall, each iteration takes $\Theta(n)$ time, and there are $\sqrt{n}$ iterations. Thus the overall time is $\Theta(n^{3/2})$.

*(Moral: a simple algorithm like bubble sort can have pretty varying behavior depending on the input.)*

**Question 4**

(a) We can prove this incrementally, placing one pigeon at a time. We start with just one pigeon in the row. Suppose at time $i$, we have the row having pigeons $p_1, p_2, \ldots, p_{i-1}$ (in that order), and we are placing the pigeon $q$. If $q$ is pecked by $p_1$, we are done, as we can place $q$ in the left-most spot. Likewise, if $q$ pecks $p_{i-1}$, we are done, as we can place it to the right of all the current pigeons.

Now, let us suppose neither of the above happens. This means that $q$ pecks $p_1$, and $q$ is pecked by $p_{i-1}$. Let us now consider $p_2, p_3, \ldots, p_{i-1}$. There must be some $j$ with the property that $q$ pecks $p_{j-1}$ but $q$ is pecked by $p_j$ (this is because we know that we move from pecking
Problem 5

(a) By definition, $A = p(10^{n/3})$ and $B = q(10^{n/3})$, and thus we have $AB = r(10^{n/3})$.

(b) This is also clear, as $r(z) = C_1 z^4 + C_2 z^3 + \ldots$

(c) Let us take $r(2)$. By definition, this is $p(2)q(2)$. Each of these terms has at most \( n/3 + 1 \) digits (because $A_i$ and $B_i$ are $n/3$ digits each and we are multiplying by small constants and adding up). Thus computing each $r(z)$ for $z = \{-2, -1, 0, 1, 2\}$ takes $T(n/3 + 1) + O(n)$ time. As $T(n) \leq n^2$, this is $T(n/3) + O(n)$.

(d) We have a system of linear equations, $Mx = b$, where $x$ is the vector $[C_1 C_2 C_3 C_4 C_5]$, $b$ is the vector $[r(-2) r(-1) r(0) r(1) r(2)]$, and $M$ is the so-called Vandermonde matrix. The inverse of $M$ can be computed (it will have all entries being $O(1)$). We thus use $x = M^{-1}b$ to find the $C_i$. Computing each coordinate will take $O(n)$ time, and there are only five coordinates.