CS161 Homework 1 **Due:** 15 April 2016, 12 noon
Submit on Gradescope **Handed out:** 8 April 2016

Instructions: Please answer the following questions to the best of your ability. If you are asked to show your work, please include relevant calculations for deriving your answer. If you are asked to explain your answer, give a short ($\sim$ 1 sentence) intuitive description of your answer. If you are asked to prove a result, please write a complete proof at the level of detail and rigor expected in prior CS Theory classes (i.e. 103). When writing proofs, please strive for clarity and brevity (in that order). Cite any sources you reference.

1 Lights On (12 points)

There are n people entering and exiting a room. For each $i \in \{1, \ldots, n\}$, person i enters at time a_i and exits at time b_i (assume $b_i > a_i$ for all i), and all the a_i, b_i are distinct. At the beginning of the day, the lights in the room are switched off, and the first person who enters the room switches them on. In order to conserve electricity, if person i leaves the room at time b_i and there is no one else present in the room at time b_i, then person i will switch the lights off. The next person to enter will then switch them on again. Given the values $(a_1, b_1), (a_2, b_2), \ldots (a_n, b_n)$, we want to find the number of times the lights get switched on.

Design the following algorithms, and prove the correctness and running time of each algorithm.

(a) A $\Theta(n^2)$ algorithm that computes the number of times the lights get switched on.

Solution: (4 points)

Algorithm: For each person i we can determine whether someone is in the room when person i enters at time a_i as follows: For every person i and every person $j \neq i$, check whether $a_j < a_i < b_j$; if so, then person j must be in the room when person i enters the room. If no such j exists, then i must turn the lights on when he/she enters the room. We count the number of people who would turn on the lights, and output it.

Running Time: For each i (there are $\Theta(n)$ of them) we need to check the above inequalities for all $j \neq i$ (there are $\Theta(n)$ of them). Therefore the algorithm runs in time $\Theta(n^2)$.

Correctness: Person i turns the lights on as he/she enters the room if and only if there is no one in the room at time a_i. For some other person j to be in the room at time a_i, person j must have entered the room before a_i ($a_j < a_i$) and stays ($b_j > a_i$). Therefore, person i turns the lights on if and only if no such j exists. Our algorithm correctly counts this: For every i, the algorithm increments the counter if and only if such j exists.

(b) An $O(n \log n)$ algorithm that computes the number of times the lights get switched on. (Hint: start by sorting the entry and exit times.)

Solution: (8 points)

Algorithm: Let L be a list of all entry and exit times altogether, i.e. L has $2n$ integers. Let us sort L in increasing order in time $O(n \log n)$ (say, using MergeSort). We then iterate over L, and we keep a variable that counts the number of people in the room (which is set to 0 to begin with). We also keep a counter (c) that stores the number of times the lights get switched on. Let x be the number of people in the room when we examine time t_i. If $t_i = a_i$ (i.e., person i entres the room), if $x = 0$ then we increment c by one (since person i turns the lights on) and increment x by one (since person i now occupies the room); we then examine the next entry/exit time. If $t_i = b_i$ instead (i.e., person i leaves the room), we decrement x by one (as person i leaves the room); we then examine the next entry/exit time.

Running time: After we examine all $2n$ entry/exit times, we output c as the answer. The examination process runs in time $O(n)$, and the overall algorithm runs in time $O(n \log n)$ (due to sorting).

Correctness: It is clear that if the value stored in x is correct before/after each examination, then the algorithm correctly counts the number of times the lights get switched on. We increment x by one when a person enters the room, and decrement x by one when a person leaves the room. Because all entry/exit times are distinct and they are sorted, this invariant holds throughout the execution of the algorithm.

2 Select with Other Group Sizes (15 points)

In the lecture, you have seen the linear time Algorithm Select that finds the k^{th} smallest element of an array of n elements in $O(n)$ time. The algorithm *mysteriously* divides n numbers into groups of 5 elements and proceeds. In this problem, we analyze Select with different group sizes other than 5.

In Parts (a) - (c), we consider groups of size $2\ell + 1$ for $\ell \geq 1$. In Part (d), we consider groups of size 6. You may assume that n given elements are distinct. You may also ignore all the floor and ceiling functions whenever they pop up.

(a) Assuming we divide n elements into groups of $2\ell + 1$, derive an upper bound on the number of elements greater than the median of medians; this number will be a function of ℓ. Do the same for the number of elements smaller than the median of medians.

Solution: (2 points)

The number of elements smaller than the median of medians is at least

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

Then, the number of elements greater than the median of medians is at most

$$n - \left(\frac{l+1}{4l+2}n - 2(l+1)\right) \leq \frac{3l+1}{4l+2}n + 2(l+1) \ .$$

By a similar reasoning, we get that the number of elements smaller than the median of medians is at most

$$\frac{3l+1}{4l+2}n + 2(l+1) \ .$$

(b) Derive a recurrence for the worst-case running time $T(n)$ in terms of ℓ. You can assume that the median of each group of size $2\ell + 1$ is found in $O(\ell^2)$ time.

Solution: (3 points)

For the n elements, we do an $O(n)$ amount of work to sort the groups and find their medians and $T\left(\lceil \frac{n}{2l+1} \rceil\right)$ amount of work, via recursion, to find the median of medians. Then, we recurse over either the elements greater than or those smaller than the median of medians. By Part (a), the part we are recursing over has the number of elements at most $\frac{3l+1}{4l+2}n + 2(l+1)$ in both cases. This leads to the following recurrence:

$$T(n) \leq \begin{cases} O(1) & \text{if } n < n_0, \\ T\left(\lceil \frac{n}{2l+1} \rceil\right) + T\left(\frac{3l+1}{4l+2}n + 2(l+1)\right) + O(n) & \text{if } n \geq n_0. \end{cases}$$

Note n_0 is some constant to be determined in Part (c).

(c) Using the substitution method, solve the recurrence from Part (b) to obtain $T(n) = O(n)$. Note that the substitution method will not work for all possible values of ℓ. State a sufficient condition on ℓ under which the substitution method works. What is the running time of the algorithm for the cases when the substitution method does not give $O(n)$?

Solution: (5 points)

Using the substitution method, we show that $T(n) = O(n)$. More specifically, we show that $T(n) \le cn$ for all $n \ge n_0$ for some constants c and n_0. Let a be a constant such that the $O(n)$ term in the recurrence in Part (b) is upper bounded by an for $n > 0$. Assuming that $T(n) \le cn$ for $n \ge n_0$, the recurrence reduces to the following:

$$\begin{aligned} T(n) &\le c\left\lceil \frac{n}{2l+1} \right\rceil + c\left(\frac{3l+1}{4l+2}n + 2(l+1)\right) + an \\ &\le \frac{1}{2l+1}cn + c + \frac{3l+1}{4l+2}cn + 2(l+1)c + an \\ &= \frac{3l+3}{4l+2}cn + (2l+3)c + an \\ &= cn + \left(-\left(1 - \frac{3l+3}{4l+2}\right)cn + (2l+3)c + an\right) \\ &= cn + \left(-\frac{l-1}{4l+2}cn + (2l+3)c + an\right). \end{aligned}$$

If $\frac{l-1}{4l+2} > 0$, we can choose c and n_0 such that $-\frac{l-1}{4l+2}cn + (2l+3)c + an \le 0$ for all $n > n_0$. For instance, $c = \frac{2a}{p}$ and $n_0 = \frac{2(2l+3)}{p}$ satisfy this property. (To see this, note that $-pcn + (2l+3)c + an \ge 0$ is equivalent to $\frac{an}{p(n-(2l+3)/p)} \le c$ where $p = \frac{l-1}{4l+2}$.) Then the last expression is less than or equal to cn and we can conclude that $T(n) \le cn$. Note this is possible if $\frac{l-1}{4l+2} > 0$ which is equivalent to $l > 1$. Therefore, a sufficient condition for this method to work is $l > 1$.

When $l = 1$, we are considering groups of size 3 and the recurrence reduces to

$$T(n) \le \begin{cases} O(1) & \text{if } n < n_0, \\ T\left(\lceil \frac{n}{3} \rceil\right) + T\left(\frac{2}{3}n + 4\right) + O(n) & \text{if } n \ge n_0. \end{cases}$$

Using the recursion tree method, we can show $T(n) = O(n \log n)$. We do an $O(n)$ amount of work per level and there are $O(\log n)$ levels in the recursion tree.

(d) Assuming we divide the n elements into groups of 6, repeat Parts (a) - (c). The median of 6 elements is the 3rd smallest number.

Solution: (5 points)

We follow the same line of reasoning as in Parts (a) - (c). The difference is that the median of a even-sized group is not exactly in the middle. We use the convention that if there are an even number of elements, the median is the lower median. The median of 6 elements is the 3rd smallest element. The number of elements smaller than the median of medians is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{6} \right\rceil \right\rceil - 2\right) \ge \frac{n}{4} - 6 \ ,$$

and hence, the number of elements greater than the median of medians is at most

$$n - \left(\frac{n}{4} - 6\right) \le \frac{3n}{4} + 6 \ .$$

Similarly, the number of elements greater than the median of medians is at least

$$4\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{6} \right\rceil \right\rceil - 2\right) \ge \frac{n}{3} - 8 \ ,$$

and the number of elements smaller than the median of medians is at most

$$n - \left(\frac{n}{3} - 8\right) \leq \frac{2n}{3} + 8 \ .$$

Since the part we are recursing over has size at most $\frac{3n}{4} + 6$ for large enough n in the worst case, we obtain the following recurrence:

$$T(n) \leq \begin{cases} O(1) & \text{if } n < n_0, \\ T\left(\lceil \frac{n}{6} \rceil\right) + T\left(\frac{3}{4}n + 6\right) + O(n) & \text{if } n \geq n_0. \end{cases}$$

As before, we can use the substitution method to show that $T(n) = O(n)$.

3 Even Sub-arrays (15 points)

Suppose you are given an array of positive integers A of size n. For $i, j \in \mathbf{N}$ and $1 \leq i \leq j \leq n$, a sub-array of A is an array of the form $[A[i], A[i+1], \ldots, A[j]]$ where $i, j \in \mathbf{N}$, $1 \leq i \leq j \leq n$ and $A[i]$ refers to the ith element of A. A prefix of A is a sub-array of the form $[A[1], \ldots, A[i]]$ for $i \in \mathbf{N}$ and $1 \leq i \leq n$. A suffix of A is a sub-array of the form $[A[i], \ldots, A[n]]$ for $i \in \mathbf{N}$ and $1 \leq i \leq n$. The sum of a sub-array refers to the sum of all its elements. In this problem we will design a divide and conquer algorithm that counts the number of sub-arrays with *even* sum.

(a) A naive approach to this problem would consist of computing the sum of all possible sub-arrays and then returning the number of these of even sum. What would be the resultant running time of this algorithm?

Solution: (2 points)

The number of sub-arrays of an n-element array A is $n + \binom{n}{2} = n(n+1)/2 = O(n^2)$ (n for 1-element sub-arrays and $\binom{n}{2}$ for choosing $1 \leq i < j \leq n$). For each sub-array, computing the sum can be done in linear time. Overall, this algorithm runs in time $O(n^3)$.

(b) Now show how to compute the sums of all prefixes of A in $O(n)$ time. Given these values, show that the sum of any sub-array can be computed in $O(1)$ time. How long does it take now to go through all possible sub-arrays and return the ones of even sum?

Solution: (3 points)

Let $S[i]$ denote the sum of a prefix $A[1:i]$ of A (i.e., the first i entries). Clearly, $S[1] = A[1]$. For $i > 1$, $S[i] = S[i-1] + A[i]$. By simply iterating over A from $A[1]$ to $A[n]$, we can compute $S[1]$ through $S[n]$ in time $O(n)$.

To compute the sum of a sub-array $A[i:j]$, we simply compute $S[j] - S[i-1]$ (let us define $S[0] = 0$). Using this optimization (by pre-computing the sums of prefixes), the algorithm from Part (a) now runs in time $O(n^2)$ as we check the sum of each sub-array in time $O(1)$ and there are $O(n^2)$ sub-arrays.

(c) You are not satisfied with the running time of the last algorithm and realize that you can solve this problem faster using a divide-and-conquer approach. Now suppose that you divide A into two disjoint sub-arrays L and R of length $n/2$ [1] and determine the number of sub-arrays of even sum for both of these two sub-arrays recursively. Why is this information insufficient to compute the number of even sum sub-arrays of A? Be concise.

Solution: (2 points)

Although the two recursive calls compute the number of sub-arrays of even sum in L and R, respectively, we also need to compute the number of even sub-arrays which span across the middle point. More

[1]We will assume that n is a power of 2.

specifically, if there is an even sub-array which contains a (proper) suffix of L and a (proper) prefix of R, then this would not be accounted for by the two recursive calls. This is the extra computation we need to perform, and simply returning the number of sub-arrays of even sum does not help us compute this quantity.

(d) Suppose now that your recursive algorithm computes and returns extra information beyond the number of even sum sub-arrays: what *constant* size extra information can be returned so that from the recursive solutions for L and R one can recover in $O(1)$ time (1) the number of even sum sub-arrays of A and (2) also the extra information to return when you pop out of the recursion for A? (Hint: Think about the number of prefixes and suffixes with even and odd sum in R and L.)

Solution: (5 points)

From Part (c) we need to know the number of prefixes with even/odd sum in R and suffixes with even/odd sum in L. Thus we can modify the algorithm to return four extra numbers (in addition to the number of sub-arrays of even sum), which is the number of suffixes/prefixes with even/odd sum.

For any array X, let us define the following five quantities:

- $a(X)$: The number of sub-arrays of X with even sum,
- $p_e(X)$: The number of prefixes of X with even sum,
- $p_o(X)$: The number of prefixes of X with odd sum,
- $s_e(X)$: The number of suffixes of X with even sum, and
- $s_o(X)$: The number of suffixes of X with odd sum.

Suppose we computed these quantities when $X = L$ and $X = R$ (by two recursive calls). We need to compute each of them when $X = A$ in $O(1)$ time.

$$a(A) = a(L) + a(R) + s_e(L) \cdot p_e(R) + s_o(L) \cdot p_o(R)$$

.

To see why, any suffix of L with even sum and any prefix of R with even sum constitute a sub-array of A with even sum which is not accounted for within L or within R. Similarly, the last term counts the number of such cases for odd-sum suffix plus odd-sum prefix.

Suppose that the sum of all entries in L is even (which we can check in $O(1)$ from previous part using the array $S[]$). Then any prefix of L with even sum qualifies as a prefix of A with even sum. A prefix of A can span L and at least one element of R, which can be counted by counting the number of prefixes of R with even sum. If the sum of L is odd, we need to count the number of prefixes of R with odd sum instead. $p_e(A) = p_e(L) + p_e(R)$ (if sum of L is even) or $p_e(A) = p_e(L) + p_o(R)$ (if sum of L is odd).

Analogously, $p_o(A), s_e(A)$, and $s_o(A)$ can be computed in $O(1)$ time.

We can choose the base case to be of size 1 (one-element array), for which computing these quantities is trivial.

(e) Write a recurrence for your divide-and-conquer algorithm in part (d). Solve the recurrence using your favorite method. What is the running time of the algorithm?

Solution: (3 points)

Let $T(n)$ be the worst-case running time of our algorithm when it is called on an array of size n.

$$T(n) \leq 2T(n/2) + O(1)$$

.

As we saw in lecture, this recurrence resolves to $T(n) \leq O(n)$. The overall algorithm runs in $O(n)$ time as the pre-computing the sum of prefixes (from previous part) can be done in time $O(n)$.

4 Recurrences Galore (15 points)

Please solve the recurrences below giving tight upper-bounds of the form $T(n) \leq O(f(n))$ for an appropriate function f. You can use any method from class. Show your work. Note: Unless otherwise stated, log refers to base 2 log, and ln refers to natural log.

(a) $T(n) = 14 \cdot T(\frac{n}{3}) + n^2 \ln n$.

Solution: (2 points)

Using the Master Theorem, we get $a = 14$, $b = 3$, and $n^{\log_b a} = n^{\log_3 14}$. $f(n) = n^2 \ln n = O(n^{\log_3 14 - \epsilon})$, where the quotient ϵ:

$\frac{n^2 \ln n}{n^{\log_3(14)}} = \frac{\ln n}{n^{log_3 \frac{14}{9}}} = O(n^{-\epsilon})$

For any $0 < \epsilon < \log_3 \frac{14}{9}$. Therefore, by case 1 of the Master Theorem: $T(n) = \Theta(n^{\log_3 14})$.

(b) $T(n) = 4 \cdot T(\frac{n}{4}) + n \cdot (\log n)^2$.

Solution: (3 points)

From the recursion tree:

$$\begin{aligned}
T(n) &= 4T(\frac{n}{4}) + n\log^2 n \\
&= n\log^2 n + 4[4T(n/4^2) + n/4\log^2(n/4)] \\
&= n\log^2 n + n\log^2(n/4) + 16[4T(n/4^3) + n/4^2\log^2(n/4^2)] \\
\ldots \\
T(n) &= n\log^2 n + n\log^2(n/4) + n\log^2(n/4^2) + ... + nT(1)
\end{aligned}$$

Note that there are $\Theta(\log n)$ terms, and each term is no greater than $n\log^2 n$. Thus, $T(n) = O(n\log^3 n)$. For the lower bound, notice that $\left\lfloor \frac{\log_4 n}{2} \right\rfloor$ (roughly the first half) of the terms are at least as big as:

$n\log^2 \frac{n}{4^{\lfloor \frac{\log_4 n}{2} \rfloor - 1}} \geq n\log^2 \frac{n}{4^{\log_4 n * \frac{1}{2} - 1}} = n\log^2 \frac{4n}{n^{\frac{1}{2}}} = n\log^2 4n^{\frac{1}{2}} > n\log^2 n^{\frac{1}{2}} = \frac{1}{4}n\log^2 n$ Thus, $T(n) = \Omega(n\log^3 n)$. Together we have $T(n) = \Theta(n\log^3 n)$.

(c) $T(n) = 161^2 \cdot T(\sqrt[161]{n}) + 161 \cdot (\log n)^2$.

Solution: (2 points)

Define $2^k = n$ and let $G(k) = T(n)$. Then $G(k) = 161^2 G(\frac{k}{161}) + 161k^2$. Since $\log_{161} 161^2 = 2$, by Master's Theorem (case 2) the solution is $G(k) = \Theta(k^2 \log k)$. Hence $T(n) = G(k) = \Theta(k^2 \log k) = \Theta((\log n)^2 \log\log n)$.

(d) $T(n) = \left(T(\frac{n}{161})\right)^{161} \cdot n$.

(Clarification added on 4/12/2016: Please see Piazza post @123 for a clarification.)

Solution: (3 points)

Let $G(n) = \log T(n)$. Then $G(n) = 161G(\frac{n}{161}) + \log n$. Using Master's Theorem, we get $G(n) = \Theta(n)$, but we need to obtain a specific function $g(n) = \Theta(n)$ such that $G(n) \leq g(n)$ (including constants) so that we can recover $T(n) = 2^{G(n)} = O(2^{g(n)})$. (Otherwise, the answer will be in the form $2^{O(n)}$ instead of $O(f(n))$.)

Let $g(n) = cn - d\log n$ where $c, d > 0$ (Note that if one tries $g(n) = cn$, there is no constant c such that $G(n) \leq g(n) = cn$ for all large n, at whcih point one should infer that $g(n)$ should be smaller than some linear function cn, and $cn - d\log n$ is one such function. Also note that $cn - d\log n = \Theta(n)$.)

Using the substitution method, we get $G(n) = 161g(\frac{n}{161}) + \log n = 161\left(c\frac{n}{161} - d\log\frac{n}{161}\right) + \log n \leq cn - d\log n$; this inequality must hold for large n (we are to choose c, d). The terms cn on both sides cancel out (which means, c can be any positive number), and we are left with $-161d\log\frac{n}{161} + \log n \leq -d\log n$. Any $d > 1/160$ ensures that this inequality holds fod any large n. That is, we prove that $G(n) \leq cn - d\log n$ for any positive c and any $d > 1/160$.

To bound $T(n)$ we get $T(n) = 2^{G(n)} \leq 2^{cn-d\log n} = \frac{2^{cn}}{n^d}$. For instance, if we choose $c = 1$ and $d = 1$, we get $T(n) = O(\frac{2^n}{n})$. We could have chosen a smaller value for $c > 0$ (while keeping $d = 1$), and therefore we accept any $f(n)$ in the form of $\frac{a^n}{n}$ where $a > 1$.

(e) $T(n) = 3 \cdot T(n/4) + n\log n$.

Solution: (2 points) We can apply the Master Theorem (case 3). First, $n\log n = \Omega(n^{(\log_4 3)+\epsilon})$ (to see why let $n = 4^k$ and choose $\epsilon = 1/2$, for instance). Case 3 has another condition, which is to show that $3(n/4)\log(n/4) \leq cn\log n$ for some (positive) constant $c < 1$. The LHS is $(3/4)n\log n - (3/2)n$, so we can choose $c = 3/4$, for instance. This leads to $T(n) = \Theta(n\log n)$.

(f) $T(n) = T(\lfloor\sqrt[4]{n}\rfloor) + T(\lceil\sqrt{n}\rceil) + \log n$. When analyzing this one, you can ignore the floor and ceiling functions for simplicity.

Solution: (3 points)

Let us ignore the floor/ceiling functions (treat them as integers).

Then, with the substitution $m = \log n$, we get the recurrence $T(2^m) = T(2^{m/4}) + T(2^{m/2}) + m$. Using $S(m) = T(2^m)$, this becomes $S(m) = S(m/4) + S(m/2) + m$, for which the solution is $S(m) = \Theta(m)$ by using a recursion tree. Thus, $T(n) = T(2^m) = S(m) = \Theta(m) = \Theta(\log n)$.

(Note: Earlier we posted our solutions with $O(m)$, but it should be $\Theta(m)$ to show tightness.)