CSCI 570 - Fall 2016 - HW 5

September 23, 2016

1. Solve the following recurrences by giving tight Θ-notation bounds in terms of n for sufficiently large n. Assume that $T(\cdot)$ represents the running time of an algorithm, i.e. $T(n)$ is positive and non-decreasing function of n and for small constants c independent of n, $T(c)$ is also a constant independent of n. Note that some of these recurrences might be a little challenging to think about at first.

   (a) $T(n) = 4T(n/2) + n^2 \log n$

   (b) $T(n) = 8T(n/6) + n \log n$

   (c) $T(n) = 2T(\sqrt{n}) + \log_2 n$

   Solution: In some cases, we shall need to invoke the Master Theorem with one generalization as described next. If the recurrence $T(n) = aT(n/b) + f(n)$ is satisfied with $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

   (a) Observe that $f(n) = n^2 \log n$ and $n^{\log_b a} = n^{\log_2 4} = n^2$, so applying the generalized Masters theorem, $T(n) = \Theta(n^2 \log^2 n)$.

   (b) Observe that $n^{\log_b a} = n^{\log_6 8}$ and $f(n) = n \log n = O(n^{\log_6 8 - \epsilon})$ for any $0 < \epsilon < \log_6 8 - 1$. Thus, invoking Master's Theorem gives $T(n) = \Theta(n^{\log_b a}) = n^{\log_6 8}$

   (c) Use the change of variables $n = 2^m$ to get $T(2^m) = 2T(2^{m/2}) + m$. Next, denoting $S(m) = T(2^m)$ implies that we have the recurrence $S(m) = 2S(m/2) + m$. Note that $S(\cdot)$ is a positive function due to the monotonicity of the increasing map $x \to 2^x$ and the positivity of $T()$. All conditions for applicability of Masters Theorem are satisfied and using the generalized version gives $S(m) = \Theta(m \log m)$ on observing that $f(m) = m$ and $m^{\log_b a} = m$. We express the solution in terms of $T(n)$ by

   $$T(n) = T(2^m) = S(m) = \Theta(m \log m) = \Theta(\log n \log \log n)$$

   , for large enough n so that the growth expression above is positive.

2. Solve Kleinberg and Tardos, Chapter 5, Exercise 3.

   In a set of cards, if more than half the cards belong to a single user, we call the user a majority user.

   Divide the set of cards into two roughly two equal halves, (that is one half is of size $\lfloor \frac{n}{2} \rfloor$ and the other, of size $\lceil \frac{n}{2} \rceil$). For each half, recursively solve the following problem, "decide if there exists a majority user and if he/she exists find a card corresponding to him/her (as a representative)".

   Once we have solved the problem for the two halves, we can combine them to solve the problem for the whole set as follows.

   If neither half has a majority user, then the whole set clearly does not have a majority user.

   If each of the halves have a majority user, then the whole set has a majority user if and only if both the halves have the same majority user. If both the halves have the same majority user, then we can pick either one of the output cards output by the halves as a representative for the whole set.

   If one half has a majority (say user U) and the other does not, then by comparing the representative card of user U with every other card in the whole set, count the number of cards that belong to user U in the whole set. If this count is greater than n/2, output that there is a majority and also the representative card.

   If $T(n)$ denotes the number of comparisons (invocations to the equivalence tester) of the resulting divide and conquer algorithm, then

   $$T(n) \leq 2T(\lceil \frac{n}{2} \rceil) + n - 1 \Rightarrow T(n) = \mathcal{O}(n \log n)$$

   Remark : There are ways to solve this problem with at most $\mathcal{O}(n)$ comparisons, can you think of one of such algorithm ?

3. Solve Kleinberg and Tardos, Chapter 5, Exercise 5.

   Let $L = \{L_1, L_2, \ldots, L_n\}$ be the sequence of lines sorted in increasing order of slope. From now on, when we say sort a set of lines, it is in increasing order of slope. We divide the set of lines in half and solve recursively. When we are down to a set with only one line, we return the line as visible.

   Recursively compute $L_{Bslash} = \{L_{i_1}, L_{i_2}, \ldots, L_{i_m}\}$, the sorted sequence of visible lines of the set $\{L_1, L_2, \ldots, L_{\lfloor \frac{n}{2} \rfloor}\}$. In addition compute the set of points $A = \{a_1, a_2, \ldots, a_{m-1}\}$ where a_j is the intersection of L_{i_j} and $L_{i_{j+1}}$.

Likewise compute $L_{slash} = \{L_{k_1}, L_{k_2}, \ldots, L_{k_r}\}$, the sorted sequence of visible lines of the set $\{L_{\lfloor \frac{n}{2} \rfloor+1}, \ldots, L_n\}$. In addition compute the set of points $B = \{b_1, b_2, \ldots, b_{r-1}\}$ where b_j is the intersection of L_{k_j} and $L_{k_{j+1}}$.

Observe that by construction $\{a_1, a_2, \ldots, a_m\}$ and $\{b_1, b_2, \ldots, b_r\}$ are in increasing order of x-coordinate since if two visible lines intersect, the visible part of the line with smaller slope is to the left.

We now describe how the solutions for the two halves are combined. Merge the two sorted lists A and B, to form $C = \{c_1, c_2, \ldots, c_{m+r}\}$ ($\mathcal{O}(n)$ time).

Let $L_{up}(j)$ be the uppermost line in $L_{Bslash}(j)$ at the x-coordinate of c_j and $\bar{L}_{up}$ the uppermost line in L_{slash} at the x-coordinate of c_j. Let ℓ be the smallest index at which $\bar{L}_{up}$ is above L_{up} at the x-coordinate of c_j.

Let s and t be the indices such that $L_{up}(\ell) = L_{i_s}$ and $\bar{L}_{up}(\ell) = L_{j_t}$.

Let (a, b) be the intersection of $L_{up}(\ell)$ and $\bar{L}_{up}(\ell)$. Then a lies between the x-coordinates of $c_{\ell-1}$ and c_ℓ. This implies that $L_{up}(\ell)$ is visible immediately to the left of a and $\bar{L}_{up}(\ell)$ to the right. Hence the sorted set of visible lines of L is $L_{i_1}, L_{i_2}, \ldots, L_{i_{s-1}}, L_{i_s}, L_{j_t}, Lj_{t+1}, \ldots, L_r$.

The combination step takes $\mathcal{O}(n)$ time. If $T(n)$ denotes the running time of the algorithm, then

$$T(n) \leq 2T(\lfloor \frac{n}{2} \rfloor) + \mathcal{O}(n) \Rightarrow T(n) = \mathcal{O}(n \log(n))$$

4. Assume that you have a black-box that can multiply two integers. Describe an algorithm that when given an n-bit positive integer a and an integer x, computes x^a with at most $\mathcal{O}(n)$ calls to the black-box.

   If a is odd,
   $$x^a = x^{\lfloor \frac{a}{2} \rfloor} \times x^{\lfloor \frac{a}{2} \rfloor} \times x$$
   If a is even,
   $$x^a = x^{\lfloor \frac{a}{2} \rfloor} \times x^{\lfloor \frac{a}{2} \rfloor}$$
   In either case, given $x^{\lfloor \frac{a}{2} \rfloor}$ it takes at most three calls to the black-box to compute x^a. We have thus reduced the problem of computing x^a to computing $x^{\lfloor \frac{a}{2} \rfloor}$ (which is an identical problem with input of size one bit smaller). Let $T(n)$ denote the running time of the corresponding divide-conquer algorithm. Thus

   $$T(n) \leq T(n-1) + 3 \Rightarrow T(n) = \mathcal{O}(n)$$

   Remark: Such computations arise frequently in fields like cryptography. For instance, during the encryption of messages in the RSA cryptosystem, one has to make such a computation (m^e modulo a fixed number) where the exponent e is around 1024 bits. With the above algorithm the number of multiplication calls to the blackbox is around 1024. If a naive method is used, it could take 2^{1024} multiplications which would be highly undesirable as this number far exceeds the number of atoms in the known universe !

5. Consider the following algorithm *StrangeSort* which sorts n distinct items in a list A.

   (a) If $n \leq 1$, return A unchanged.

   (b) For each item $x \in A$, scan A and count how many other items in A are less than x.

   (c) Put the items with count less than $n/2$ in a list B.

   (d) Put the other items in a list C.

   (e) Recursively sort lists B and C using *StrangeSort*.

   (f) Append the sorted list C to the sorted list B and return the result.

   Formulate a recurrence relation for the running time $T(n)$ of *StrangeSort* on a input list of size n. Solve this recurrence to get the best possible $O(\cdot)$ bound on $T(n)$.

   Solution: Suppose that A has n elements. Then steps (c), (d) and (f) can take up to $O(n)$ time, step (a) takes constant time and step (e) takes $2T(n/2)$ time. Step (b), when implemented in the way described above, takes $\Theta(n^2)$ time since each scan of A takes $\Theta(n)$ time and the scan is repeated for each of the n elements of A. Counting times taken for all the steps, we have

   $$T(n) = 2T(n/2) + O(n) + \Theta(n^2) + O(1) = 2T(n/2) + \Theta(n^2)$$

   Comparing the above recurrence with $T(n) = aT(n/b) + f(n)$ we have $f(n) = \Theta(n^2)$ and $n^{\log_b a} = n$ so that Masters Theorem gives $T(n) = \Theta(n^2)$.

   Note: Although steps (b) and (c) together can be implemented more efficiently by sorting A (or finding the median of A) first, the question specifically asks for the running time of the given implementation without any further optimizations.

6. Consider an array A of n numbers with the assurance that $n > 2, A[1] \geq A[2]$ and $A[n] \geq A[n-1]$. An index i is said to be a local minimum of the array A if it satisfies $1 < i < n, A[i-1] \geq A[i]$ and $A[i+1] \geq A[i]$.

   (a) Prove that there always exists a local minimum for A.

   (b) Design an algorithm to compute a local minimum of A. Your algorithm is allowed to make at most $O(\log n)$ pairwise comparisons between elements of A.

Solution: We prove the existence of a local minimum by induction. For $n = 3$, we have $A[1] \geq A[2]$ and $A[3] \geq A[2]$ by the premise of the question and therefore $A[2]$ is a local minimum. Let $A[1 : n]$ admit a local minimum for $n = k$. We shall show that $A[1 : k+1]$ also admits a local minimum. If we assume that $A[2] \leq A[3]$ then $A[2]$ is a local minimum for $A[1 : k+1]$ since $A[1] \geq A[2]$ by premise of the question. So let us assume $A[2] > A[3]$. In this case, the k length array $A[2 : k+1] = A'[1 : k]$ satisfies the induction hypothesis ($A'[1] = A[2] \geq A[3] = A'[2]$ and $A'[k-1] = A[k] \leq A[k+1] = A'[k]$) and hence admits a local minimum. This is also a local minimum for $A[1 : k+1]$. Hence, proof by induction is complete.

Consider the following algorithm with array A of length n as the input, and return value as a local minimum element.

```
(a) If $n = 3$, return $A[2]$.
(b) If $n > 3$,
    i. $k \leftarrow \lfloor n/2 \rfloor$
    ii. If $A[k] \leq A[k-1]$ and $A[k] \leq A[k+1]$ then return $A[k]$
    iii. If $A[k] > A[k-1]$ then call the algorithm recursively on $A[1 : k]$,
         else call the algorithm on $A[k : n]$.
```

Complexity: If the running time of the algorithm on an input of size n is $T(n)$, then it involves a constant number of comparisons and assignments and a recursive function call on either $A[1 : \lfloor n/2 \rfloor]$ (size $= \lfloor n/2 \rfloor$) or $A[\lfloor n/2 \rfloor : n]$ (size $= n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$). Therefore, $T(n) \leq T(\lceil n/2 \rceil) + \Theta(1)$. Invoking Masters Theorem gives $T(n) = O(\log n)$.

Proof of Correctness: We employ induction. For n = 3 it is clear that step (a) returns a local minimum using the premise of the question that $A[1] \geq A[2]$ and $A[3] \geq A[2]$. Let us assume that the algorithm correctly finds a local minimum for all $n \leq m$ and consider an input of size $m + 1$. Then $k = \lfloor (m+1)/2 \rfloor$. If step (b)(ii) returns, then a local minimum is found by definition and the algorithm gives the correct output. Otherwise, step (b)(iii) is executed since one of $A[k] > A[k-1]$ or $A[k] > A[k+1]$ must be true for step (b)(ii) to not return. If $A[k] > A[k-1]$, then $A[1 : k]$ must admit a local minimum by the first part of the question and the given algorithm can find it correctly if $k \leq m$, using the induction hypothesis on the correctness of the algorithm for inputs of size up to m. This holds if $\lfloor (m+1)/2 \rfloor \leq m$ which is true for all $m \geq 1$. Similarly, if $A[k] > A[k+1]$, then $A[k : m+1]$ must admit a local minimum by the first part of the question and the given algorithm can find it correctly if $m - k + 2 \leq m$, using the induction hypothesis on the correctness of the algorithm for inputs of size up to m. This holds if $k \geq 2$ or equivalently $\lfloor (m+1)/2 \rfloor \geq 2$ which holds for all $m \geq 3$. Therefore, the algorithm gives the correct output for inputs of size $m + 1$ and the proof is complete by induction.

7. Given a sorted array of n integers that has been rotated an unknown number of times, give an $O(\log n)$ algorithm that finds an element in the array. An example of array rotation is as follows: original sorted array $A = [1, 3, 5, 7, 11]$, after first rotation $A' = [3, 5, 7, 11, 1]$, after second rotation $A'' = [5, 7, 11, 1, 3]$.

   Solution: There are many different ways to solve this problem. Here, we shall consider reusing well known algorithms and algorithms developed in other questions on this homework as building blocks. Consider the following approach.

   (a) Compute the rotation index p, i.e. find the minimum p such that $A[1 : p]$ and $A[p+1 : n]$ are both sorted arrays in ascending order (if there is no rotation, then return p = 0).

   (b) If $p \neq 0$, run binary search on $A[1 : p]$ to find the element. Return *success* if found.

   (c) if not found then run binary search on $A[p+1 : n]$ to find the element. Return *success* if found and *failure* if not found.

   Correctness of the algorithm is obvious. The two binary searches can be accomplished in $O(\log n)$ time each. Hence, if we can accomplish step (a) in $O(\log n)$ time then the problem is solved. If there is no rotation, it can be confirmed by testing whether $A[1] < A[n]$ is true in constant time. In the presence of rotations it is necessarily true that $A[1] > A[n]$. If rotations are present then to find the rotation index p, we recall the algorithm in question 6, to find the local minimum in an array, and assume that all elements in A are distinct. Using the definitions in question 6, it is straightforward to see that $A[p + 1]$ is the only local minimum if there exists a non-zero rotation p and thus finding the local minimum index is equivalent to finding the rotation index. In particular, we have $A[1] < A[2] < \cdots < A[p], A[p + 1] < A[p + 2] < \cdots < A[n]$ and $A[p] > A[p + 1]$. Thus, the rotation index p can be found in $O(\log n)$ time from the complexity analysis in question 6, and this completes our solution.