CSCI 570 - Fall 2021 - HW 5 Solution

Due date : 30th September 2021

For all divide-and-conquer algorithms follow these steps:
1. Describe the steps of your algorithm in plain English.
2. Write a recurrence equation for the runtime complexity.
3. Solve the equation by the master theorem

1. Solve the following recurrences by giving tight Θ-notation bounds in terms of n for sufficiently large n. Assume that T(·) 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^2logn
   b) T(n) = 8T(n/6) + nlogn
   c) T(n) = $\sqrt{6000}$ T(n/2) + $n^{\sqrt{6000}}$
   d) T(n) = 10T(n/2) + 2^n
   e) 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) = Θ($n^{\log_b a}$ $\log^k n$) for some k ≥ 0, then T(n) = Θ($n^{\log_b a}$ $\log^{k+1} n$).

a) Observe that f(n) = n^2log n and $n^{\log_b a} = n^{\log_2 4} = n^2$, so applying the generalized Master's theorem, T (n) = Θ(n^2 $\log^2 n$).
b) Observe that $n^{\log_b a} = n^{\log_6 8}$ and f(n) = nlogn = O($n^{\log_6 8 - \varepsilon}$) for any 0 < ε < $\log_6 8 - 1$. Thus, invoking Master's Theorem gives T(n) = Θ($n^{\log_b a}$) = Θ($n^{\log_6 8}$).

c) We have $n^{\log_b a} = n^{\log_2 \sqrt{6000}} = n^{0.5 \log_2 6006} = O(n^{0.5 \log_2 8192}) = n^{13/2}$ and $f(n) = n^{\sqrt{6006}} = \Omega(n^{70}) = \Omega(n^{\frac{13}{2} + \varepsilon})$ for any $0 < \varepsilon < 63.5$. Thus, from Master's Theorem $T(n) = \Theta(f(n)) = \Theta(n^{\sqrt{6006}})$.

d) We have $n^{\log_b a} = n^{\log_2 10}$ and $f(n) = 2^n = \Omega(n^{\log_2 10 + \varepsilon})$ for any $\varepsilon > 0$. Therefore, Master's Theorem implies $T(n) = \Theta(f(n)) = \Theta(2^n)$.

e) 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 $x \longmapsto 2^x$ and the positivity of $T(\cdot)$. All conditions for applicability of Master's 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 $T(n) = T(2^m) = S(m) = \Theta(m \log m) = \Theta(\log_2 n \log \log_2 n)$, for large enough n so that the growth expression above is positive.

2. 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. Please also provide proof of correctness for your algorithm.

   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 $A'[k-1] = A[k] \geq A[k+1] = A'[k]$)

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 recursively 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, $T(n) \leq T(\lceil n/2 \rceil) + \Theta(1)$. Assuming n to be a power of 2, this recurrence simplifies to $T(n) \leq T(n/2) + \Theta(1)$ and invoking Master's 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.

3. The recurrence T(n) = 7T (n/2) + n^2 describes the running time of an algorithm ALG. A competing algorithm ALG′ has a running time of $T'(n)$ = a$T'\left(\frac{n}{4}\right)$ + $n^2 \log n$. What is the largest value of a such that ALG′ is asymptotically faster than ALG?

Solution:

We shall use Master Theorem to evaluate the running time of the algorithms.

(a) For T(n), setting $0 < \varepsilon < \log_2 7 - 2 \approx 0.81$ we have $n^2 = O(n^{\log_2 7 - \varepsilon})$. Hence $T(n) = \Theta(n^{\log_2 7})$.

(b) For $T'(n)$, if $\log_4 a > 2$ then setting $0 < \delta < \log_4 a - 2$ implies $n^2 \log n = O(n^{\log_4 a - \delta})$ and hence $T'(n) = \Theta(n^{\log_4 a})$.

To have $T'(n)$ asymptotically faster than T(n), we need $T'(n) = O(T(n))$, which implies $n^{\log_4 a} = O(n^{\log_2 7})$ and therefore $\log_4 a \leq \log_2 7 \Longrightarrow a \leq 49$. Since other expressions for run time of ALG′ require $\log_4 a \leq 2 \Longrightarrow a \leq 16 < 49$, the largest possible value of a such that ALG′ is asymptotically faster than ALG is a = 49.

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

Solution:

Let us call a card to be a *majority card* if it belongs to a set of equivalent cards encompassing at least half of the total number of cards to be tested, *i.e.* for a total of n cards, if there is a set of more than n/2 cards that are all equivalent to each other, then any one of these equivalent cards constitutes a majority card. To keep the conquer step as simple as possible, we shall require the algorithm to return a majority card if one exists and return nothing if no majority card exists (this can be implemented by indexing the cards from 1 to n and a return value of 0 could correspond to returning nothing). Then the algorithm consists of the following steps, with S denoting the set of cards being tested by the algorithm and |S| denoting its cardinality.

a) If $|S| = 1$ then return the card.
b) If $|S| = 2$ then test whether the cards are equivalent. If they are equivalent then return either card, else return nothing.
c) If $|S| > 2$ then arbitrarily divide the set S into two disjoint subsets S1 and S2 such that $|S1| = \lfloor |S|/2 \rfloor$ and $|S2| = \lceil |S|/2 \rceil$.
d) Call the algorithm recursively on the set of cards S1.
e) If a majority card is returned (call it card m1), then test it for equivalence against all cards in $S \setminus \{m1\}$. If m1 is equivalent to at least $\lfloor |S|/2 \rfloor$ other cards in S, then return card m1.
f) If m1 is not equivalent to at least $\lfloor |S|/2 \rfloor$ other cards in S, OR if no majority card was returned by the recursive call on S1, then call the algorithm recursively on the set of cards S2.
g) If a majority card is returned (call it card m2), then test it for equivalence against all cards in $S \setminus \{m2\}$. If m2 is equivalent to at least $\lfloor |S|/2 \rfloor$ other cards in S, then return card m2.
h) If m2 is not equivalent to at least $\lfloor |S|/2 \rfloor$ other cards in S, OR if no majority card was returned by the recursive call on S2, then return nothing.

Complexity: Let T(n) be the number of equivalence tests performed by the algorithm on a set of n cards. We have T(2) = 1 from step (b). Steps (d) and (f) can incur a total of $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$ equivalence tests whereas steps (e) and (g) could require up to $2(n - 1)$ equivalence tests. Therefore, we have the recurrence $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2n - 2$, which can be solved from first principles like in (6). Alternatively, the recurrence simplifies to $T(n) \leq 2T(n/2) + 2n - 2$ on assuming n to be a power of 2. The simplified recurrence can be solved using Master's Theorem to yield $T(n) = O(n \log n)$.

Proof of Correctness: We need to show that the algorithm returns the correct answer in both cases, *viz.* majority card present and majority card absent. If $|S| = 1$ then this single card forms a majority by our definition and is correctly returned by step (a). If $|S| = 2$ then either card is a majority card if both cards are equivalent and neither card is a majority card if they are not equivalent. Clearly, step (b) implements exactly this criterion. For steps (d)-(h) we consider what happens when a majority card is present or absent.

(a) *Majority card present:* If a majority card is present (say m), then at least one of the subsets S1 or S2 must have m as a majority card. It is simple to see this by contradiction since if neither S1 nor S2 have a majority card (or if m is not a mojority card in either subset) then

necessarily the number of equivalent cards (or cards equivalent to m, including m) is upper bounded by $0.5\lfloor|S|/2\rfloor + 0.5\lceil|S|/2\rceil = 0.5|S|$ which immediately implies the absence of any majority card in S. Thus, at least one of m1 (in step (e)) or m2 (in step (g)) will be returned and at least one of these will be equivalent to m. Since steps (e) and (g) respectively test m1 and m2 against all other cards, the algorithm will necessarily find one of them to be equivalent to m and hence also as a majority card (since the set of majority cards is necessarily unique) and would return the same.

(b) *Majority card absent:* There are multiple possibilities in this scenario. If neither S1 nor S2 returns a majority card then clearly there is no majority card, and the algorithm terminates at step (h) returning nothing. If m1 or m2 or both are returned as valid majority cards then they would be respectively discarded as candidates in steps (e) and (g) when checked against all other cards in S as they are not truly in majority on the set S. Thus the algorithm correctly terminates in step (h) returning nothing.

Note: There are a lot of repeated equivalence tests in the algorithm described above, primarily because we are solving the problem in a top-down manner. Can you think of a bottom-up approach to solving this problem with potentially fewer equivalence tests? Hint: It is possible to achieve O(n) complexity in terms of the number of equivalence tests required.

5. Given a binary search tree T, its root node *r*, and two random nodes *x* and *y* in the tree. Find the lowest common ancestry of the two nodes *x* and *y*. Note that a parent node *p* has pointers to its children *p.leftChild()* and *p.rightChild()*, a child node does **not** have a pointer to its parent node. The complexity must be $O(\log n)$ or better, where *n* is the number of nodes in the tree.
   Recall that in a binary search tree, for a given node *p*, its right child *r*, and its left child *l*, $r.value() \geq p.value() \geq l.value()$. Hint: use divide and conquer

   Solution: Use divide and conquer:

```
Node  LCA(r, x, y){
        if (x.value() < r.value && y.value() > r.value())
                Return r;
        else If(x.value() < r.value && y.value() < r.value())
                Return LCA(r.leftChild() x, y);
        else
```

Return LCA(r.rightChild() x, y);
}

Recurrence relation: T(n/2)+c
Complexity O(log n)

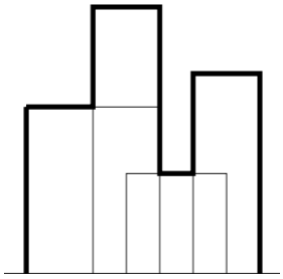
Note: Using the property of the binary search tree to decide if two nodes belong to the same branch or different branches of a parent node is crucial for maintaining a complexity of O(log n). Simply using the binary search to answer this question will make the overall complexity of the algorithm to be $O(\log^2 n)$.

Note that the property of a classic binary search tree is that no duplicates are allowed, so students need not care about the fact that the elements in the tree can be equal. Some few modern implementations allow duplicates in the binary search tree, but that is specifically implemented according to the requirements of each application and is not considered as a pure binary search tree.

6. Suppose that you are given the exact locations and shapes of several rectangular buildings in a city, and you wish to draw the skyline (in two dimensions) of these buildings, eliminating hidden lines. Assume that the bottoms of all the buildings lie on the x-axis. Building B_i is represented by atriple (L_i, H_i, R_i), where L_i and R_i denote the left and right x coordinates of the building, respectively, and H_i denotes the building's height. A skyline isa list of x coordinates and the heights connecting them arranged in order from left to right.
For example, the buildings in the figure below correspond to the following input

(1, 5, 5), (4, 3, 7), (3, 8, 5), (6, 6,8).

The skyline is represented as follows: (1, **5**, 3, **8**, 5, **3**, 6, **6**, 8). Notice that the x-coordinates in the skyline are in sorted order.



Solution:

a) Given a skyline of n buildings and another skyline of m buildings in the form (x1, **h1**, x2, **h2**, ….xn) and (x'1, **h'1**, x'2, **h'2**, ….x'm) , show how to compute the combined skyline for the m + n buildings in O(m + n) step.

Merge the "left" list and the "right" list iteratively by keeping track ofthe current left height (initially 0), the current right height (initially 0),finding the lowest next x-coordinate in either list; we assume it is the left list. We remove the first two elements, x and h, and set the currentleft height to h, and output x and the maximum of the current left and right heights.

b) Assuming that we have correctly built a solution to part a, give a divide and conquer algorithm to compute the skyline of a given set of n buildings. Your algorithm should run in O(n log n) steps.

If there is one building, output it.
Otherwise, split the buildings into two equal groups, recursively compute skylines, output the result of merging them using part (a).

Algorithm :

getSkyline for n buildings :

- If n == 0 : return an empty list.
- If n == 1 : return the skyline for one building (it's straightforward).
- leftSkyline = getSkyline for the first n/2 buildings.

- rightSkyline = getSkyline for the last n/2 buildings.
- Merge leftSkyline and rightSkyline.

Recurrence Relation: $T(n) \leq 2T(n/2) + O(n)$

Time Complexity :
The runtime is bounded by the recurrence$T(n)$
$\leq 2T(n/2) + O(n)$,
$a = 2, b = 2, k = 1$
which implies that $T(n) = O(n \log n)$ by masters theorem.