Homework 4
Sara Pesavento
spesaven@usc.edu
[redacted]

Sara Pesavento
Homework 5
7618843154

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**

   f(n) = n^2logn =~ n^log2(4). Special case 2. = $\Theta(n^{\log_b a} log^{k+1} n)$

   T(n) = $n^2 log^2 n$

   b) **T(n) = 8T(n/6) + nlogn**

   f(n) = nlogn < n^log6(8). Case 1. = T(n) = $\Theta(n^{\log_b a})$

   T(n) = $\Theta(n^{\log_6 8})$

   c) **T(n) = √6000 T(n/2) + $n^{\sqrt{6000}}$**

   f(n) = $n^{\sqrt{6000}}$ > n^log2(√6000). Case 3?. if af(n/b) <= c*f(n)... √6000 * $(n/2)^{\sqrt{6000}}$ <= c*$n^{\sqrt{6000}}$

   T(n) = $\Theta(n^{\sqrt{6000}})$

   d) **T(n) = 10T(n/2) + 2^n**

   f(n) = 2^n > n^log2(10). Case 3?. if 10*2^(n/2) = c * 2^n

   T(n) = $\Theta(2^n)$

   e) **T(n) = 2T(√n) + $log_2 n$**

   f(n) = log2(n) n^log_sqrt(2)_ (2)

   use n = 2^m to get T(2^m) = 2*T($2^{m/2}$) + m

   f(m) = m, m^logb(a) = m.

   T(n) = T(2^m) = S(m) = Θ(mlogm) = $\Theta(log_2 n \, loglog_2 n)$

2. Consider an array A of n numbers with the assurance that n > 2, A[1] >= A[2] and A[n] >= 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] >= A[i] and A[i+1] >= A[i].
***Your algorithm is allowed to make at most O(log n) pairwise comparisons between elements of A.**

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

   Let us assume there is no local minimum. We already know A[1] is not a local minimum. Since A[2] is not a minimum, that means that A[3] <= A[2], otherwise A[2] would be a minimum. This progression continues until this last element. Without any local minimum, A[i] < A[i - 1]. However, if the last element, A[n -1] is less than it's previous A[n - 2], then A[n - 1] would be a local minimum. This is a contradiction to our initial statement.

b) Design an algorithm to compute a local minimum of A

For a log(n) algorithm, we can utilize binary search.

```
if n == 3:
    return A[2]
if n > 3:
        k = [n//2]  set to middle element
        if we have found a local minimum (A[k] lower than it's neighbors A[k-1] and A[k+1]):
                return A[k]
        if A[k] is greater than it's previous element, recursively run on the first half (it's going down so
it must have a local minimum in that half)
        else:
            recursively run on the second half (it's going down so it must have a local minimum in that
half)
```

Time complexity: By the Master's Theorem we are cutting out half the problem each time, so a = 1 and b = 2, thus, T(n) = T(n/2) + O(1). Then case 2 gives us: O(logn)

Proof:
We can observe the base case is n = 3 since A[2] is a local minimum by problem definition. Assume that we find the local minimum correctly for all values up until the last input m. Our k will search for the middle with (m//2). This will return if it found a local minimum. Otherwise it looks for the minimum in the first or second half based on if it is decreasing in the first half, or in the second half. Both ways it will find a minimum based on part a that we proved a local minimum exists.

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 $T'(n) = aT'(n/4) + n^2 \log n$. What is the largest value of a such that ALG' is asymptotically faster than ALG?

Based on the Master Theorem, the run time for ALG is case 1, T(n) = $\Theta(n^{\log_2 7})$

The run time for ALG' is f(n) = $n^2 \log n$, a = a, b = 4. n^log4(a) < $n^2 \log n$.

To have T'(n) run faster, we need T'(n) = O(T(n)) which implies n^log4(a) = O(n^log2(7))
or log4(a) <= log2(7), a = 49

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

Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that two bank cards are equivalent if they correspond to the same account. It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computations, determines whether they are equivalent. Their question is the following: among the collection of n cards, is there a set of more than n/2 of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only O(n log n) invocations of the equivalence tester.

Algorithm:
If there is one card, return 1.
If there are two cards, call the function to test their equivalence. If equivalent, return one of the cards, otherwise return None.
Divide the cards in half into two sets
Recursively call algorithm for cards in set 1 to find a most commonly occurring card
If one exists, check if it occurs more than n/2 times in all both sets using “equivalence tester”, if so return that card
Otherwise recursively call algorithm for cards in set 2 to find a most commonly occurring card
If one exists, check if it occurs more than n/2 times in all both sets using “equivalence tester”, if so return that card
return nothing

Time complexity:
T(n) = 2T(n/2) + 2n -2, giving out case 2, T(n) = O(nlogn)

Proof:
Let’s look at all the cases. If there is a single bank card, then it is the majority and we return that card. If there are two cards, then we check if they are equivalent. If yes, return either one. If not, there is no majority between the two.
If a majority card is present, then since it must appear over n/2 times, then it must be the majority in S1 or S2. Then we search all the cards in S, so after a majority is found, the equivalence tester will find if there are > n/2 total. If the majority card is not in the set of bank cards, then if there is 1 or 2 cards only, these will both return nothing. Otherwise it may find a majority in a particular half, but when using the equivalence tester to compare it with all the cards in S, it will not return anything since no card appears > n/2 times.

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() ≥ p.value() ≥ l.value(). Hint: use divide and conquer

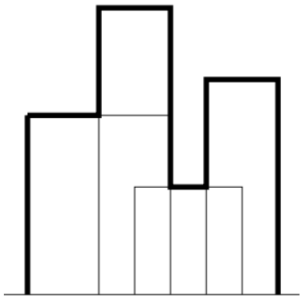
We can utilize the fact that this is a binary tree to find the ancestor of x and y
Algorithm:

```
#base case
if the x node value is less that the current node value and the y node value is more:
    return current node as the ancestor
else if the x value is less and the y value is less, recurse on left side
    return function(r.leftChild(), x, y)
else:
    return function(r.rightChild(), x, y)
```

Time Complexity:
Recurrence = T(n/2) + C = case 2 = O(logn)

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 Bi is represented by a triple (Li, Hi,

Ri), where Li and Ri denote the left and right x coordinates of the building, respectively, and Hi denotes the building's height. A skyline is a 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 sor
ted order.



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) steps.

Merge the two sorted lists. Start at the first index and choose the lowest x from the two lists. Remove that element (both it's x and height) and set the current left height to h. 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 (if none, output empty list). Otherwise, split the buildings into two groups, recursively compute skylines, and output the result of merging them using part (a).
The runtime is bounded by the recurrence function $T(n) \leq 2T(n/2) + O(n)$, which implies that $T(n) = O(n \log n)$.