Homework 5

All ungraded

1. For each part below, solve the following recurrences by giving tight Θ-notation bounds in terms of n for sufficiently large n, and briefly describe the steps. Assume that T(n) is a positive and non-decreasing function of n and represents the running time of an algorithm. In some cases, we shall need to invoke the Master Theorem with a generalization of case 2:

   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≥0, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

   a. $T(n) = 49T(n/7) + n^2 \log n$

      Observe that $f(n) = n^2 \log n$ and $n^{\log_b a} = n^{\log_7 49} = n^2$, so applying the case 2 of Master's theorem, $T(n) = \Theta(n^2 \log^2 n)$.

   b. $T(n) = 4.001\,T(n/2) + n^2 \log^4 n$

      Observe that $n^{\log_b a} = n^{\log_2 4.001}$ and $f(n) = n^2 \log^4 n = O(n^{\log_2 4.001 - \epsilon})$ for any $0 < \epsilon < \log_2 4.001 - 2$. Thus, invoking the case 1 of Master's theorem gives T(n) = $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 4.001})$.

   c. $T(n) = 100T(n/2) + n^{50}$

      We have $n^{\log_b a} = n^{\log_2 100} = O(n^7) = O(n^{50})$.
      Also, we should check the regularity condition $a.f(n/b) \le c.f(n)$. Replacing the values, we have $100.(n/2)^{50}/n^{50} \le c$. So, any c such that $100/(2^{50}) < c < 1$ satisfies the regularity condition. Therefore, from case 3 of Master's theorem,
      $T(n) = \Theta(f(n)) = n^{50}$

   d. $T(n) = 10T(n/2) + 2^n$

      We have $n^{\log_b a} = n^{\log_2 10}$ and $f(n) = 2^n = \Omega(n^{\log_2 10+\epsilon})$ for any $\epsilon > 0$. Also, we should check the regularity condition $a.f(n/b) \le c.f(n)$. Replacing the values, we have $10.2^{n/2}/2^n \le c$. As n gets large, $10.2^{n/2}/2^n$ decrease, so we can say that $a.f(n/b) \le c.f(n)$ for any suitable $c < 1$ for all large n. Therefore, using case 3 of Master's Theorem implies that $T(n) = \Theta(f(n)) = \Theta(2^n)$.

   e. $T(n) = 2T(\sqrt{n}) + \log n$

      Master Theorem does not directly apply since the first term is not in the $aT(n/b)$ form. We analyze the recurrence tree as follows: At problem size n, the amount of work we need to put is $\log n$. On the 2 subsequent subproblems in the next level, the problem size becomes $\sqrt{n}$. The amount of work we need to put in for each is $\log \sqrt{n} = 1/2 \log n$.

Therefore, at the next level, the total work we need to do equals $1/2\ log\ n + 1/2 log\ n = log\ n$. We can see that the total work of all subproblems at each level stays the same = $log\ n$.

Now, we need to figure out how many levels there are. Suppose the initial size is $n = 2^k$ (where $k = \log n$). Since the size of the subproblem goes from 2^k to $2^{k/2}$ in the next level, the exponent becomes half each time. So it reaches from k to a constant in $\log k$ levels. Since $k = \log n$, we have $loglog\ n$ levels. Since at each level, the workload is $log\ n$, the complexity becomes - $\Theta(log\ n\ loglog\ n)$

Also, the number of subproblems doubles each time, so the no. of leaves (i.e. at level $\log k$) is $2^{\log k} = k = \log n$. So the leaves contribute to an additional $\Theta(log\ n)$, thus, the overall complexity remains $\Theta(log\ n\ loglog\ n)$.

4 points each for giving the correct answer:

1. 2 points for correct bound
2. 2 points for correct explanation

1. Assume that you have a blackbox 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 $O(n)$ calls to the blackbox. (a 'blackbox' simply means a 'function' whose implementation specifics are *unknown*)

If a is odd,

$$x^a = (x^{\lfloor a/2 \rfloor} \times x^{\lfloor a/2 \rfloor}) \times x$$

If a is even,

$$x^a = x^{\lfloor a/2 \rfloor} \times x^{\lfloor a/2 \rfloor}$$

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

$$T(a) \leq T(a/2) + 2 \Rightarrow T(a) = \Theta(log\ a) = \Theta(n)$$

The recurrence is solved using Master Method (Case 2), and since a is n-bit, so, $n \sim log\ a$.

1. 15 points for the correct algorithm:
    a. 10 points for showing x^a can be calculated in constant time if we know $x^{a/2}$. (Need to show both even and odd conditions)
    b. 5 points for the correct recurrence.
2. 5 points for showing the correct time complexity with Master's Theorem.

3. Solve Kleinberg and Tardos, Chapter 5, Exercise 3.
   In a set of cards, if more than half of 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 $\lfloor n/2 \rfloor$ and the other, of size $\lceil n/2 \rceil$).
   For each half, recursively solve the following problem, "decide if there exists a majority user and if she exists, find a card corresponding to 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, i.e. finding the global majority user. We can do that as follows.

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

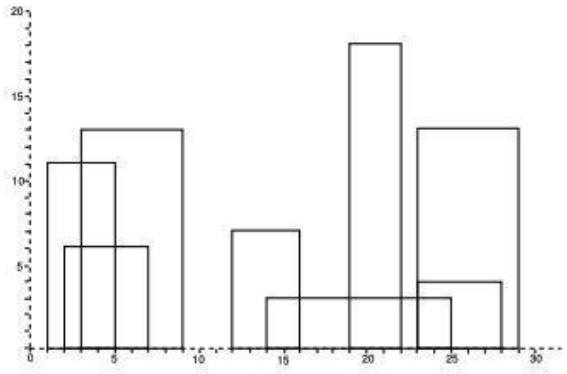
   If both the halves have the same majority user, then that user is the global majority user. We can pick either one of the output cards returned by the halves as a representative for the whole set.

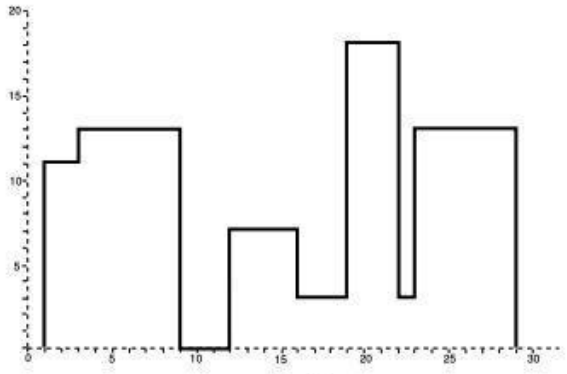
   If the majority users are different, or if only one of them has a majority user, we need to check if any of these users is a global majority user. We can do this in a linear manner by comparing the representative card of the majority user with every other card in the whole set, counting the number of cards that belong to the same majority user.

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

$$T(n) \leq 2T(\lceil n/2 \rceil) \; + \; O(n) \Rightarrow T(n) \; = \; O(n \log n)$$

4. A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. A building B_i is represented as a triplet (L_i, H_i, R_i) where L_i and R_i denote the left and right x coordinates of the building, and H_i denotes the height of the building. Describe an O(n log n) algorithm for finding the skyline of n buildings.
   For example, the skyline of the buildings {(3, 13, 9),(1, 11, 5),(12, 7, 16),(14, 3, 25),(19, 18, 22),(2, 6, 7),(23, 13, 29),(23, 4, 28)} is {(1, 11),(3, 13), (9, 0), (12, 7), (16, 3), (19, 18), (22, 3), (23, 13), (29, 0)}.
   (Note that the x coordinates in a skyline are sorted)





a)
Solution:
We will use divide and conquer to merge the buildings into a skyline.

Skyline(n buildings):
if n == 1 return the building
A =Skyline(the first n/2 buildings)
B =Skyline(the last n/2 buildings)
merge skylines A and B
Merge (A , B):
We can merge the two skylines in linear time. Let x be the next x-coordinate in order. It can be from A or B or both.
We keep track of the current skyline height using crtH. Initialize it to 0
When we select x, there are two cases:
Case A) If height(x) > crtH then the skyline jumps up. We output (x, height(x)) to the skyline, and update crtH = height(x).
Case B) if height(x) < crtH then the skyline drops. It can drop to height(x) or the last height in the other skyline, whichever is larger.
Each x is handed in O(1) hence runtime of merge is O(n)
This problem has the same recurrence relation as mergesort and hence has a runtime complexity of O(n logn).

Rubrics:

1. 15 pts for correctly merging skylines in linear time
    a. 6 points - Correctly identifies when the skyline jumps
    b. 6 points - Correctly identifies when the skyline drops
    c. 3 points - correctly ensures x is in sorted order
2. 5 points - Base case & functions calls for the divide & conquer approach
3. 5 points for correct time complexity

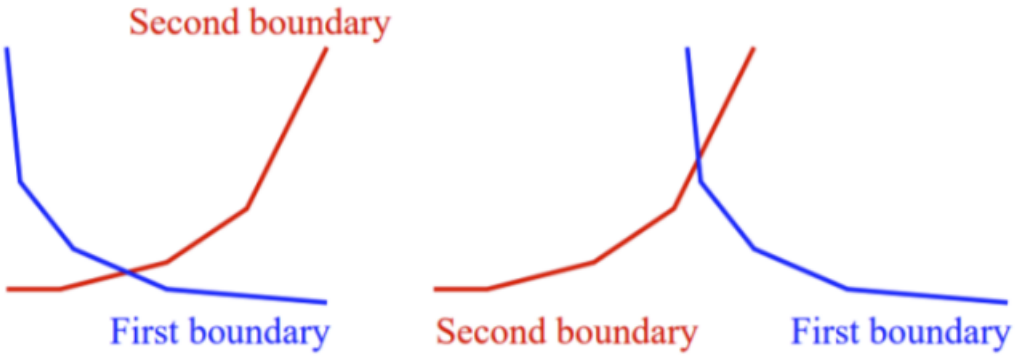
5. Solve Kleinberg and Tardos, Chapter 5, Exercise 5.

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 $\{L_1, L_2, \ldots, L_{\lfloor n/2 \rfloor}\}$.
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 $\{L_{\lceil n/2 \rceil}, \ldots, L_n\}$. 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. For this, we need to merge the two recursively computed sorted lists to get the list for the combined set of lines. The set of visible lines essentially forms a boundary, when seen from above. The intuition here is to find the point where the boundaries for the two halves intersect. This can then directly be used to find the boundary for the whole set, i.e. finding the set of visible lines for the whole set. To locate the intersection point, we parse the two recursively-computed sorted lists to locate the first instance where a line from the first half is below a line from the second half. The intersection of these lines gives us the desired point. The figure below illustrates two examples of such intersections1. This merging step can be done in O(n) time.



More specifically, we need to merge the two sorted lists A and B. Let $L_{up}(j)$ be the uppermost line in L_{Bslash} (j) and $\overline{L}_{up}$ the uppermost line in L_{slash}. Let l be the smallest index at which $\overline{L}_{up}$ is above L_{up}.

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

Let (a, b) be the intersection of $L_{up}(l)$ and $\overline{L}_{up}(l)$. This implies that $L_{up}(l)$ is visible immediately to the left of a and $\overline{L}_{up}(l)$ to the right. Hence the sorted set of visible lines of L is

$L_{i_1}, L_{i_2}, \ldots, L_{i_{s-1}}, L_{i_s}, L_{j_t}, L_{j_{t+1}}, \ldots, L_r$

1. 10 points for correctly describing the need for sorting by slope and also calculating intersection points.
2. 10 points for the correct merge step between 2 sorted sets of lines by identifying the intersection.
3. 5 points for correct time complexity.