1. [20 points] Solve the following recurrences by giving tight Θ-notation bounds in terms of n for sufficiently large n. Assume that T(n) represents the running time of an algorithm, i.e., T(n) is a positive and non-decreasing function of n. For each part below, briefly describe the steps along with the final answer:
   a. $T(n) = 9T(n/3) + n^2 logn$
   b. $T(n) = (4.01)T(n/2) + n^2 logn$
   c. $T(n) = \sqrt{6000}T(n/2) + n^{\sqrt{6000}}$
   d. $T(n) = 10T(n/2) + 2^n$
   e. $T(n) = 2T(\sqrt{n}) + log_2 n$

According to Master Theorem $T(n)=aT(\frac{n}{b})+f(n)$

a. $T(n)=9T(n/3)+n^2\log n$

$a=9$, $b=3$. $\Rightarrow n^{\log_b a}=n^{\log_3 9}=n^2$.

$f(n)=n^2\log n$

If $f(n)=\Theta(n^{\log_b a}\log^k n)$ for some $k\geq 0$, then

$T(n)=\Theta(n^{\log_b a}\log^{k+1} n)$

We know $f(n)=n^2\log n=\Theta(n^2\log n)$ $k=1$

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

b. $T(n)=(4.01)T(n/2)+n^2\log n$

$a=4.01$, $b=2$ $\Rightarrow n^{\log_b a}=n^{\log_2 4.01}>n^2$.

$f(n)=n^2\log n$, grows slower than $n^{\log_b a}$

$T(n)=\Theta(n^{\log_b a})=\Theta(n^{\log_2 4.01})$

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

$a=\sqrt{6000}$, $b=2$. $\Rightarrow n^{\log_b a}=n^{\log_2\sqrt{6000}}=n^{0.5\log_2 6000}=\Theta(n^{\frac{13}{2}})$

$f(n)=n^{\sqrt{6000}}$

∴ i. If $f(n)=\Omega(n^{\log_b a+\varepsilon})$ for some $\varepsilon>0$, and if $af(n/b)\leq cf(n)$ for some constant $c<1$ and all sufficiently large n, then $T(n)=\Theta(f(n))$

ii. We know $f(n)=\Omega(n^{77})=\Omega(n^{\frac{13}{2}+\varepsilon})$ for $0<\varepsilon<70.5$.

Also. $af(n/b) = \sqrt{6000}\left(\frac{n}{2}\right)^{\sqrt{6000}} \le c n^{\sqrt{6000}}$

holds for large enough $n$ and

$\because \frac{\sqrt{6000}}{2^{\sqrt{6000}}} n^{\sqrt{6000}} \le c n^{\sqrt{6000}}$, $\frac{\sqrt{6000}}{2^{\sqrt{6000}}} < 1$

$\therefore$ some $c < 1$ exist.

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

d. $T(n) = 10\,T(\frac{n}{2}) + 2^n$

$a = 10$, $b = 2$ $\Rightarrow n^{\log_b a} = n^{\log_2 10}$

$f(n) = 2^n$

$\because f(n) = \Omega(n^{\log_2 10 + \varepsilon})$ for any $\varepsilon > 0$.

Also, $af(n/b) = 10\,2^{\frac{n}{2}} = 10\sqrt{2^n}$

$\because 10\sqrt{2^n} \le c \cdot 2^n$ is true for larger $n$ and some $c < 1$.

$\therefore af(n/b) \le cf(n)$ for large $n$ and some $c < 1$

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

e Assume, $n = 2^m$ $T(n) = 2T(\sqrt{n}) + \log_2 n$.

$T(2^m) = 2T(2^{\frac{m}{2}}) + m$

Suppose $Q(m) = T(2^m)$

$Q(m) = 2Q(m/2) + m$ $\Rightarrow$ Master Theorm

$a = 2$, $b = 2$ $\Rightarrow m^{\log_b a} = m$

$f(m) = m = \Theta(m \log_b^a \log^k m) = \Theta(m)$ $k = 0$.

$Q(m) = \Theta(m^{\log_2^2} \log^{k+1} m)$, $k = 0$

$= \Theta(m \log m)$

$\therefore T(n) = T(2^m) = Q(m) = \Theta(m \log m)$, $m = \log_2 n$

$= \Theta(\log_2 n \log(\log_2 n))$

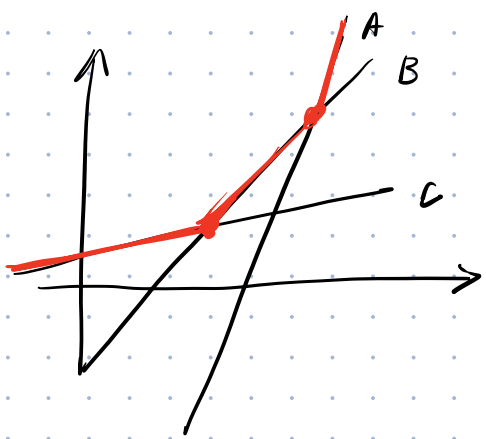
2. [25 points] Solve Kleinberg and Tardos, Chapter 5, Exercise 5.

5. *Hidden surface removal* is a problem in computer graphics that scarcely needs an introduction: when Woody is standing in front of Buzz, you should be able to see Woody but not Buzz; when Buzz is standing in front of Woody, … well, you get the idea.
The magic of hidden surface removal is that you can often compute things faster than your intuition suggests. Here's a clean geometric example to illustrate a basic speed-up that can be achieved. You are given $n$ nonvertical lines in the plane, labeled $L_1, \ldots, L_n$, with the $i^{th}$ line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines all meet at a single point. We say line $L_i$ is *uppermost* at a given $x$-coordinate $x_0$ if its $y$-coordinate at $x_0$ is greater than the $y$-coordinates of all the other lines at $x_0$: $a_i x_0 + b_i > a_j x_0 + b_j$ for all $j \neq i$. We say line $L_i$ is *visible* if there is some $x$-coordinate at which it is uppermost-intuitively, some portion of it can be seen if you look down from "$y = \infty$."
Give an algorithm that takes $n$ lines as input and in $O(n \log n)$ time returns all of the ones that are visible. Figure 5.10 gives an example.

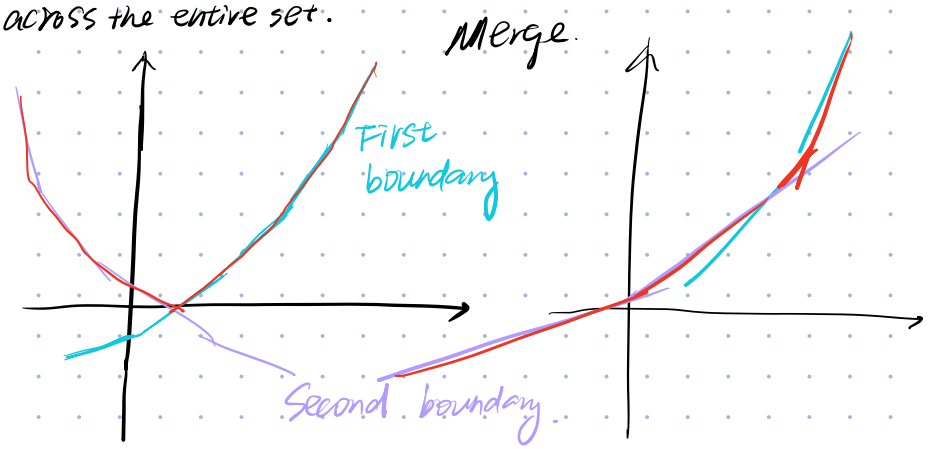
To simplify the problem, we assume there is a set of lines, each with different slopes. At the beginning of algorithm, these lines are sorted by slope value in ascending order. Then, we apply divide and conquer strategy.

First, split the set in half and tackle each subset recursively until a subset contains only one line, which is then deemed visible.

For each half, we identify visible lines and their intersection points, creating two sequences of lines and points sorted by their x-coordinates. This sorting ensures that for any two intersecting lines, the one with the lesser slope is visible to the left of their intersection.



We then merge these sequences from both halves. The key step is to find the intersection point where the visibility boundary of the first half meets that of the second. This point helps us determine the continuos visibility boundary across the entire set.




The merging process, designed to be efficient, combines these sequences while maintaining their order by x-coordinate. The final list represents a boundary of visibility when viewed from above.

The recurrence relation can be expressed as.

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

$2T(\frac{n}{2})$: recursive division of the problem into two halves.

$O(n)$: linear time $O(n)$ spent on merging the solutions of these halfs.

By master theorem, we know $T(n) = O(n \log n)$

3. [20 points] 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.

Idea: The algorithm reduces the power operation into smaller, manageable sub-problems. It does so by recursively solving for $x^{\lfloor a/2 \rfloor}$, halving the problem size each time.

Recursive Method

If a is an odd number, $x^a$ can be decomposed into $x^{\lfloor a/2 \rfloor} * x^{\lfloor a/2 \rfloor} * x$.

If a is an even number, $x^a$ can be decomposed into $x^{\lfloor a/2 \rfloor} * x^{\lfloor a/2 \rfloor}$

Simplify the problem $x^a$ to calculating $x^{\lfloor a/2 \rfloor}$ each time, reducing the size of the problem until a is reduced to 1 or 0.

Each recursive call handles a problem of one bit less, and each step requires at most three calls to the multiplication black box.

Recursive relation: $T(n) \leq T(n-1) + 3$.

Time complexity. $\rightarrow T(n) = \Theta(n)$.

Iteration Method.

Step1. Initialization: result `res` is 1.

Step2: For a given n-bit integer a, traverse its binary representation from the lowest bit to the highest bit.

Step3: For every one:

If the current bit is 1, then multiply `res` by the current X value (using a black box to perform the multiplication.) i.e. `res = res * x`.

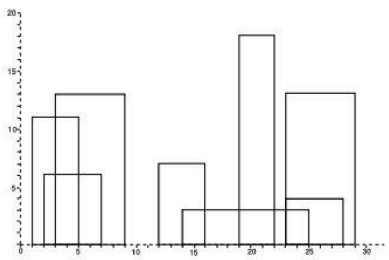
No matter what the current bit is, after each bit is traversed, x is squared, (`x = x * x`) in preparation for the calculation of the next bit.

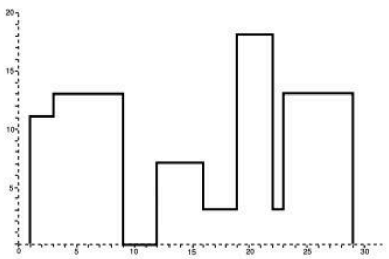
Step4. When all bits have been traversed, the value of res is $X^a$.

For an n-bit integer, at most each bit of it needs to be traversed for calculation, which involves at most one multiplication and one x calculation of the square of.

Thus, the total number of black box is linearly dependent on n, at most 2n times, which is O(n) complexity.

4. [25 points] 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)


buildings


skyline

Step 1: Divide

If the set of buildings consists of more than one building, divide it into two halves. If there is only one building $B_i = (L_i, H_i, R_i)$. Its skyline is $[(L_i, H_i), (R_i, 0)]$.
$\Rightarrow O(1)$

Step 2: Conquer

Recursively compute the skyline for each half. The recursion breaks down the problem into smaller subsets of buildings until it reaches the base case of a single building.
$\Rightarrow$ Recursion depth $O(\log n)$.

Step 3: Merge

Merge the two skyline obtained from the left and right halves of the building set:

1. Keeping track of the current height in both skylines as iterate.
2. At each step, comparing the x-coordinates from current points in both skylines and choosing the

point with the lower x-coordinate to proceed.

3. Updating the height at each step to reflect the current highest point between the two skylines being merged. If the next point changes the current height, that point is added to the final skyline

=> A linear complexity O(n) for merge. Since the number of points can grow in relation to the number of buildings, the worst case scenario for merging operation through out the algorithm scale as O(n log n)

Overall,

$T(n) = 2T(n/2) + O(n)$

According to master theorem.

$T(n) = \Theta(n \log n)$