MATH 416, PROBLEM SET 3

Comments about homework.

- Solutions to homework should be written clearly, with justification, in complete sentences. Your solution should resemble something you'd write to teach another student in the class how to solve the problem.
- You are encouraged to work with other 416 students on the homework, but solutions must be written independently. Include a list of your collaborators at the top of your homework.
- You should submit your homework on Gradescope, indicating to Gradescope where the various pieces of your solutions are. The easiest (and recommended) way to do this is to start a new page for each problem.
- Attempting and struggling with problems is **critical** to learning mathematics. Do not search for published solutions to problems. I don't have to tell you that doing so constitutes academic dishonesty; it's also a terrible way to get better at math.

  If you get stuck, ask someone else for a hint. Better yet, go for a walk.

Warning. These are not necessarily *model solutions*; they are meant to help you understand the problems you didn't totally solve and maybe to give you alternative solutions. Sometimes I will give less or more detail here than I would expect from you.

Problem 1. By pre-sorting the input array at the beginning, show that the Closest-Pair algorithm we described in class can be improved to have worst-case running time $O(n \log n)$ (where n is the number of points), as advertised.

Solution. The point is that, at the beginning, we can create two (for convenience) auxiliary lists, one in which the input is sorted by x-coordinate and one in which the input is sorted by y-coordinate. (Link the lists so that from the position of an element in one list one can in constant time determine its position in the other.) It is easy to see that the algorithm does not require these lists to be reordered at any point, so there is no need for further sorting. This improves our "merge" step to linear cost (iteratively check the distance between each point in the narrow central strip and its 7ish successors in the sorted-by-y list). So there is some $O(n \log n)$ initial cost, and the post-sorting worst-case running time $T(n)$ follows the familiar recurrence $T(n) = 2T(n/2) + O(n)$, which has solution $T(n) \in O(n \log n)$ by the Master Theorem. The total cost is $O(n \log n) + O(n \log n)$, i.e., $O(n \log n)$. □

Problem 2.

(a) For a given constant $\delta > 0$, find an example (with proof) of an increasing function $f\colon \mathbb{N} \to \mathbb{N}$ such that f is $O(n^\delta)$, but f is not $O(n^\gamma)$ for any $\gamma < \delta$, and f is not $\Theta(n^\delta)$. Conclude by commenting on the claim, "any f that is $O(n^{\log_b a})$ is covered by one of the first two cases in the Master Theorem."

Recall the third case of the Master Theorem (assuming $a \geq 1$ and $b > 1$ are constants, $f\colon \mathbb{N} \to \mathbb{N}$ is increasing, ...):

> If both
> (i) f is $\Omega(n^\gamma)$ for some constant $\gamma > \log_b a$, and
> (ii) there is a constant $c < 1$ such that $af(n/b) \leq cf(n)$ for all n sufficiently large,
> then $T(n)$ is $\Theta(f(n))$.

(b) Show that if $f(n) = n^{\log_b a}$, then (ii) is false.

(c) Find an example of a function f for which (i) holds but (ii) fails. (You may choose your favorite values of a and b, e.g., $a = b = 2$.) (*Hint:* One approach is to build a function f for which the inequality $af(n/b) \geq f(n)$ holds for infinitely many n. You can choose which n these are in advance, and then you have a lot of freedom to decide the remaining values of the function in order for (i) to hold.)

(d) Show that in fact (ii) *implies* (i). (For simplicity you may consider only n that are exact powers of b.)

Solution. (a) Let $f(n) = n^\delta/\lceil \log n \rceil$ (with finitely many values suitably modified so the function is nonnegative). Certainly f is $O(n^\delta)$. Suppose toward a contradiction that f is $O(n^\gamma)$ for some $\gamma < \delta$. Let C and N_0 be constants witnessing this: for all $n > N_0$ the inequality $f(n) \leq Cn^\gamma$ holds. Rearrange:

$$n^{\delta-\gamma} \leq C\lceil \log n \rceil$$

for all n sufficiently large. But this contradicts the fact that $\log n$ is not dominated by any positive power of n. (Notice that $\delta - \gamma$ is positive.) So $f(n)$ cannot be $O(n^\gamma)$.

Suppose toward a contradiction that f is $\Omega(n^\delta)$, witnessed by $D > 0$ and N_1: for $n > N_1$ the inequality $f(n) \geq Dn^\delta$ holds. Rearrange to get $\lceil \log n \rceil \leq 1/D$ for all sufficiently large n, an evident contradiction since $\lim_{n\to\infty} \log n = \infty$. So f is not $\Theta(n^\delta)$.

(b)

$$a\frac{f(n)}{f(n/b)} = a\frac{n^{\log_b a}}{(n/b)^{\log_b a}} = a\frac{a^{\log_b n}}{a^{\log_b n - 1}} = 1.$$

(c) The negation of (ii) says that for every $c < 1$ there are *infinitely many* n for which $af(n/b) \leq cf(n)$ holds.

To ensure that (ii) fails, it would be enough to arrange that

$$af(n/b) \geq f(n) \tag{0.1}$$

holds for infinitely many n.

Assume $a = 1$. In this case, we can conclude from (**??**) that f is constant on the interval $[n/b, n]$. It would be enough to arrange for f to be constant, with value v_k say, on the interval $[b^{2k}, b^{2k+2})$. But we still need (i) to hold. We are free to choose the values v_k however we like; we could even arrange that $f(n) = v_k > n$ for every n by setting $v_k = b^{2k+2}$, for instance. Now, consider $n \in \mathbb{N}$ and let k be the unique integer for which $n \in [b^{2k}, b^{2k+2})$. (That is, $k = \lfloor \frac{1}{2} \log_b n \rfloor$.) We have

$$f(n) = b^{2k+2} > n,$$

which certainly implies that f is $\Omega(n)$. And (ii) fails, since for the infinitely many n of the form b^{2k+1} we have

$$f(n) = f(b^{2k+1}) = v_k = f(b^{2k}) = f(n/b).$$

To construct an example with $a > 1$, arrange for $f(b^{2k+1}) = af(b^{2k})$, choose $f|_{(b^{2k+1}, b^{2k+2})}$ arbitrarily; as in the case $a = 1$ you have freedom to choose the values $v_k = f(b^{2k})$ to be as large as you like. In particular, you can arrange for f to be $\Omega(n^\gamma)$ for any prescribed γ.

(d) Assume that (ii) holds: for all $n \geq N_0$ we have $af(n/b) \leq cf(n)$. By increasing N_0 if necessary, we may assume that N_0 is a power of b.

Claim. For all $n \geq N_0$ that are powers of b and all $i \leq \log_b(n/N_0)+1$, we have $a^i f(n/b^i) \leq c^i f(n)$.

Proof of Claim. We prove the Claim by induction on i (for all n simultaneously). The base case is $i = 1$, for which the Claim reduces to our assumption.

Fix i and suppose inductively that for all n for which $i \leq \log_b(n/N_0)+1$ the inequality $a^i f(n/b^i) \leq c^i f(n)$ holds. Let n be such that $i + 1 \leq \log_b(n/N_0) + 1$, and notice that this allows us to apply the inductive hypothesis. Applying the assumption (ii) with n/b^i in place of n gives $af(n/b^{i+1}) \leq cf(n/b^i)$; now multiply each side of the inequality by a^i and combine that with our inductive hypothesis:

$$a^{i+1} f(n/b^{i+1}) \leq a^i(cf(n/b^i)) = c \cdot a^i f(n/b^i) \leq c^{i+1} f(n).$$

This finishes the induction and the proof of the Claim. □

The Claim for $i = \log_b(n/N_0)$ gives

$$a^{\log_b n - \log_b N_0} f(N_0) \leq c^{\log_b n - \log_b N_0} f(n).$$

which transforms by our usual log-swapping trick into the inequality

$$n^{\log_b a} N_0^{\log_b c} f(N_0) \leq n^{\log_b c} N_0^{\log_b a} f(n)$$

Rearrange to get

$$f(n) \geq f(N_0) N_0^{\log_b c - \log_b a} \cdot n^{\log_b a - \log_b c},$$

which holds for all $n \geq N_0$. (Of course, $f(N_0) N_0^{\log_b c - \log_b a}$ is constant.) Finally, notice that $\gamma := \log_b a - \log_b c > \log_b a$ since $c < 1$. $\square$

Problem 3. A *partial matching* of $[n] = \{1, \ldots, n\}$ is simply a set of pairwise-disjoint 2-element subsets of $[n]$. That is, it's a way of pairing up some of the elements of $[n]$, possibly leaving some unpaired. For example, $\{\{2,5\},\{3,6\},\{1,7\}\}$ is a partial matching of [8] (in which 4 and 8 are left unpaired). (The pairs are unordered.)

(a) List or draw all partial matchings of [4].
(b) Let m_n be the number of partial matchings of $[n]$. Prove that this sequence satisfies the recurrence $m_{n+1} = m_n + nm_{n-1}$ for $n \geq 1$ and $m_0 = m_1 = 1$.
(c) The *exponential generating function* of a sequence[1] (a_n) is the (formal) power series

$$\sum_{n=0}^{\infty} \frac{a_n}{n!} z^n.$$

Let $M(z)$ be the exponential generating function of (m_n). Verify that $M(z)$ and $e^{z+\frac{1}{2}z^2}$ each satisfy the initial value problem $\frac{d}{dz}M(z) = (1+z)M(z)$, $M(0) = 1$ (and hence are equal).

Solution. (a) There are 10 partial matchings of [4], which I will indicate here in a way that is hopefully clear:

$$\begin{array}{lllll} (1)(2)(3)(4) & (1,2)(3)(4) & (1,3)(2)(4) & (1,4)(2)(3) & (2,3)(1)(4) \\ (2,4)(1)(3) & (3,4)(1)(2) & (1,2)(3,4) & (1,3)(2,4) & (1,4)(2,3) \end{array}$$

(b) Divide the matchings of $[n+1]$ into two classes: those in which $n+1$ is unmatched and those in which $n+1$ is matched. The first class is in bijection with the matchings of $[n]$ by removing $n+1$. The second class is in bijection with n copies of the set of partial matchings of $n-1$. To see this, send a matching of $n+1$ to the pair (k, M), where k is the number $n+1$ is matched with and M is a matching of the remaining $n-1$ elements. We conclude that $m_{n+1} = m_n + nm_{n-1}$.

[1]The exponential generating function is often useful for counting objects that involve some choice of ordering.

(c) Use the chain rule to see that $e^{z+\frac{1}{2}z^2}$ satisfies the differential equation. As for $M(z)$, differentiate term-by-term and use the recurrence:

$$\begin{aligned}
M'(z) &= \sum_{n=1}^{\infty} \frac{m_n}{n!} n z^{n-1} \\
&= \sum_{n=1}^{\infty} \frac{m_n}{(n-1)!} z^{n-1} \\
&= \sum_{n=0}^{\infty} \frac{m_{n+1}}{n!} z^n \\
&= 1 + \sum_{n=1}^{\infty} \frac{m_{n+1}}{n!} z^n \\
&= 1 + \sum_{n=1}^{\infty} \frac{m_n + n m_{n-1}}{n!} z^n \\
&= 1 + \sum_{n=1}^{\infty} \frac{m_n}{n!} z^n + \sum_{n=1}^{\infty} \frac{m_{n-1}}{(n-1)!} z^n \\
&= \sum_{n=0}^{\infty} \frac{m_n}{n!} z^n + \sum_{n=0}^{\infty} z^{n+1} \\
&= M(z) + zM(z).
\end{aligned}$$

(Of course, the way $e^{z+\frac{1}{2}z^2}$ was obtained in the first place was to run this argument in reverse: deduce the differential equation from the recurrence and then solve it.)

It is a standard fact (that you weren't expected to prove) that a formal initial value problem has at most one solution. □

Problem 4. Suppose that you are given n nonvertical lines in the plane, labeled $L_1, \ldots, L_n$, with the i^{th} line specified by the equation $y = a_i x + b_i$. Assume also that no three lines intersect in a single point. Say that the line L_i is *uppermost* at an x-coordinate x_0 if $a_i x_0 + b_i > a_j x_0 + b_j$ for all $j \neq i$. Say that the line L_i is *visible* if there is some x-coordinate at which it is uppermost. (Intuitively, this means that some portion of the line can be seen "looking down from $y = \infty$.") Give (with proof) an algorithm that takes n lines as input and (with proof) in $O(n \log n)$ time returns exactly the visible lines.

(*Hint:* First, sort the list of lines by slope. Then recursively apply the algorithm to the first $n/2$ lines and to the second $n/2$ lines. But it won't be enough to know which of the first $n/2$ lines are visible and which of the second $n/2$ lines are visible; your algorithm should report a bit more than that. (Consider the case $n = 4$.))

Solution. It will be convenient to write $x(p)$ for the x-coordinate of a point p and $y(p)$ for its y-coordinate.

First label the lines in order of increasing slope, and then use a divide-and-conquer approach. Note:

- If the line $y = a_i x + b_i$ is given to us as a pair (a_i, x_i), then sorting by the a_i takes $O(n \log n)$ time.
- If there are ties among the slopes, discard all but the line with largest y-intercept. (Only this one will be visible.) So we may assume in what follows that all the slopes are distinct.

If $n \geq 3$ (the "base case" of the divide-and-conquer), then we can easily find the visible lines in constant time: the first and third lines will always be visible; the second will be visible if and only if it meets the first line to the left of where the third line and the first line meet.

Let $m = \lceil n/2 \rceil$. We first recursively compute the sequence of $s \leq m$ visible lines among $L_1, \ldots, L_m$: call them $L_{i_1}, \ldots, L_{i_s}$ in increasing order of slope. We also recursively compute the sequence of points $p_1, \ldots, p_{s-1}$ where p_k is the intersection of line L_{i_k} with line $L_{i_{k+1}}$. Notice that $q_1, \ldots, q_{t-1}$ have increasing x-coordinates, since if two lines are each visible, then the region in which the line of smaller slope is uppermost lines to the left of the region in which the line of larger slope is uppermost. Similarly, we recursively compute the sequence $L_{j_1}, \ldots, L_{j_t}$ of t visible lines among $L_{m+1}, \ldots, L_n$ together with intersection points $q_1, \ldots, q_{t-1}$.

Define $\mathcal{S}_L = \{L_{i_1}, \ldots, L_{i_s}\}$ and $\mathcal{S}_R = \{L_{j_1}, \ldots, L_{j_t}\}$. To complete the algorithm, we show how to determine the visible lines in $\mathcal{S}_L \cup \mathcal{S}_R$, together with the corresponding intersection points, in linear time. We know that L_{i_1} and L_{j_t} will be visible, since they have, respectively, the minimal and maximal slopes among all lines in this list. (For x_L less than all x-coordinates of all intersection points of all the lines, the line of minimal slope is uppermost at x_L. Similarly, the line of maximal slope is uppermost at any sufficiently large x-value.)

Merge the sorted lists $p_1, \ldots, p_{s-1}$ and $q_1, \ldots, q_{t-1}$ into a single list

$$P_1, \ldots, P_{s+t-2}$$

of points ordered by x-coordinate. (This takes $O(s+t) = O(n)$ time.)

Notice that the recursion gives us:

(a) for any $k = 1, \ldots, s$, the line L_{i_k} is uppermost among all lines in $\mathcal{S}_L$ at any $x \in [x(p_{k-1}), x(p_k)]$ (taking $x(p_0) = -\infty$); and
(b) for any $k = 1, \ldots, t-1$, the line L_{j_k} is uppermost among all lines in $\mathcal{S}_R$ at any $x \in [x(q_k), x(q_{k+1})]$ (taking $x(q_t) = +\infty$).

Using these observations, search through the sorted list, comparing for each k the y-coordinates of the $\mathcal{S}_L$-uppermost line and the $\mathcal{S}_R$-uppermost line at $x(P_k)$.

(You know exactly which lines these are by the two observations above, so no additional search is required. The total cost of this additional search is still only $O(n)$.)

Let $l^* \leq s+t-2$ be the maximal index such that at P_{l^*} the $\mathcal{S}_L$-uppermost line, call it L_{i_e}, is above the $\mathcal{S}_R$-uppermost line; let $r^* \leq s+t-2$ be the minimal index such that at P_{r^*} the $\mathcal{S}_R$-uppermost line, call it L_{j_f}, is above the $\mathcal{S}_L$-uppermost line.

It is possible that l^* as defined does not exist, in which case we take $l^* = 0$ and $i_e = i_1$; it is possible that r^* as defined does not exist, in which case we take $r^* = s+t-1$ and $j_f = j_t$.

Compute the intersection of the $\mathcal{S}_L$-uppermost line at P_{l^*} with the $\mathcal{S}_R$-uppermost line at P_{r^*}: call this point of intersection P^*.

Using the fact that the slopes are listed in increasing order, one can verify that $l^* \leq r^*$.

Claim. The lines visible among $\mathcal{S}_L \cup \mathcal{S}_R$ are $L_{i_1}, \ldots, L_{i_e}, L_{j_f}, \ldots, L_{j_t}$.

Proof of Claim. Our choice of i_e guarantees that the line L_{i_e} is visible in $\mathcal{S}_L \cup \mathcal{S}_R$ (indeed, on the interval $[x(p_{e-1}), x(P^*)]$), and the fact that the lines are ordered by slope guarantees that all preceding lines in $\mathcal{S}_L$ are visible. By similar reasoning, all of $L_{j_f}, \ldots, L_{j_t}$ are visible in $\mathcal{S}_L \cup \mathcal{S}_R$. We must show that none of the other lines are visible.

We have already observed that the line L_{i_1} of minimal slope will be visible in $\mathcal{S}_L \cup \mathcal{S}_R$. Another line $L_{i_{k+1}} \in \mathcal{S}_L$ for $k \geq 1$ will be invisible in $\mathcal{S}_L \cup \mathcal{S}_R$ if and only if there is a line $L_{j_l} \in \mathcal{S}_R$ obscuring it, meaning that at $x(p_k)$ the line L_{j_l} is above $L_{i_{k+1}}$. (Since L_{j_l} has greater slope, this condition implies that L_{j_l} is above $L_{i_{k+1}}$ on the whole interval $[x(p_k), x(p_{k+1})]$.) This is true iff the line in $\mathcal{S}_R$ uppermost at $x(p_k)$ is above $L_{i_{k+1}}$, which must occur at all p_k listed after P_{l^*}, by maximality of l^*. That is, this condition holds of any $k \in \{e, \ldots, s-1\}$, so the lines $L_{i_{e+1}}, \ldots, L_{i_s}$ are invisible in $\mathcal{S}_L \cup \mathcal{S}_R$.

Using similar reasoning but the minimality of r^* instead of l^*, one can show that each of the lines $L_{j_1}, \ldots, L_{j_{f-1}}$ is invisible in $\mathcal{S}_L \cup \mathcal{S}_R$. $\square$

In light of the claim, we complete the recursion by returning

$$L_{i_1}, \ldots, L_{i_e}, L_{j_f}, \ldots, L_{j_t}$$

and the sequence of intersection points $p_1, \ldots, p_{e-1}, P^*, q_f, \ldots, q_{t-1}$.

The recurrence for the worst-case running time looks like $T(n) = 2T(n/2) + O(n)$, which gives $T(n)$ is $\Theta(n \log n)$ by the Master Theorem. $\square$

Problem 5. Suppose you are given a $2^n \times 2^n$ checkerboard with one (arbitrarily chosen) square removed. Describe an algorithm in pseudocode that computes a tiling of the board by **L**-shaped tiles, each composed of exactly three squares. Your input is the integer n and two n-bit integers representing the row and column of the missing square. The output is a list of the positions and orientations of $(4^n - 1)/3$ tiles. Your algorithm should run in $O(4^n)$.

Solution. Input: n and two n-bit numbers. Output: Positions and orientations of $(4^n - 1)/3$ tiles.

Consider the 2×2 base case. The missing square is denoted by two 1-bit numbers. The four possibilities are $(0, 0)$, $(1, 0)$, $(0, 1)$, and $(1, 1)$. Clearly there is an **L**-shaped tile that makes up the other squares. In this way we can refer to the orientation of an **L**–shaped tile by referring to the missing location. E.g. (1,1), looks like an actual **L**.

Now consider the general case. Break the $2^n \times 2^n$ board into quadrants. The quadrant location of the missing tile is determined by the first bits of the two n-bit integers. For example, $(1011, 0111)$ is located in the bottom right quadrant because the first digits are $(1, 0)$. If the two n-bit integers are $(a_1 \dots a_n, b_1 \dots b_n)$ then we add an **L**-shaped tile with orientation (a_1, b_1) at the center of the board. Now the four quadrants each have exactly one tile removed, and we proceed recursively. (Note: one tile is added at each step of the recursion at precisely the center of the board.)

The algorithm runs in $O(4^n)$-steps as there are $O(4^n)$ tile locations to write. On the other hand, each assembly time is $O(1)$ in the input as the locations of the **L**–shapes can be read from the first digits.