CS 341, Spring 2020
Semih Salihoglu

Assignment 2 (due Sunday, June 21, midnight EST)

Instructions:

- Hand in your assignment using Crowdmark. Detailed instructions are on the course website.
- Give complete legible solutions to all questions.
- Your answers will be marked for clarity as well as correctness.
- For any algorithm you present, you should justify its correctness (if it is not obvious) and analyze the complexity.

1. [*12 marks*] Consider the following recurrence:

$$T(n) = \begin{cases} T(\lfloor n/4 \rfloor) + T(\lfloor n/9 \rfloor) + T(\lfloor n/36 \rfloor) + 2016 & \text{if } n \geq 36 \\ 341 & \text{if } 1 \leq n \leq 35. \end{cases}$$

Prove inductively (i.e., with the guess-and-check or substitution method) the upper bound $T(n) = O(\sqrt{n})$. [Hint: use $T(n) \leq c\sqrt{n} - c'$ as the "guess", where c and c' are suitable constants.]

We calim that $T(n) \leq 1349\sqrt{n} - 1008$.
Base Case: $1 \leq n \leq 35$.

$$\begin{aligned} T(n) &= 341 \\ &\leq 1349 - 1008 \\ &\leq 1349\sqrt{n} - 1008 \end{aligned}$$

inductive hypothesis: Assume $T(k) \leq 1349\sqrt{k} - 1008$ for all $35 \leq k < n$. Need to show that $T(n) \leq 1349\sqrt{n} - 1008$
induction step:

$$\begin{aligned} T(n) &= T(\lfloor n/4 \rfloor) + T(\lfloor n/9 \rfloor) + T(\lfloor n/36 \rfloor) + 2016 \\ &= 1349(\sqrt{\lfloor n/4 \rfloor} + \sqrt{\lfloor n/9 \rfloor} + \sqrt{\lfloor n/36 \rfloor}) - 1008 \cdot 3 + 2016 \quad \text{by IH} \\ &= 1349(\sqrt{n/4} + \sqrt{n/9} + \sqrt{n/36}) - 1008 \\ &= 1349(\sqrt{n}/2 + \sqrt{n}/3 + \sqrt{n}/6) - 1008 \\ &= 1349(\sqrt{n}) - 1008 \end{aligned}$$

Hence by induction, $T(n) = O(\sqrt{n})$

2. [*12 marks*] Given an array A of integers. An element in A is a peak it is not smaller than its neighbours. For elements at the two ends of the array, we consider them having only one neighbour. For example, in an array of $\{5, 3, 6, 2, 1\}$, 5 and 6 are peak elements. Give an $O(\log n)$ time algorithm to find a peak element in A. Show the correctness of the algorithm and analyze its time complexity.

   For every iteration, we find the midpoint in the interval (i, j), if it is a peak we just return it. Otherwise, we update one of i and j according to the neighbours of *mid*.

Algorithm 1: FindPeak($A[0, \ldots, n-1]$)

```
1  i = 0
2  j = n − 1
3  while True do
4  |  mid = ⌊(i + j)/2⌋
5  |  if A[mid − 1]exists and A[mid + 1]exists and
6  |  A[mid − 1] ≤ A[mid] and A[mid + 1] ≥ A[mid] then
7  |  |  return A[mid]
8  |  else if i = mid then
9  |  |  return Max(A[mid − 1], A[mid])
10 |  if A[mid] ≤ A[mid + 1] then
11 |  |  i = mid + 1
12 |  else
13 |  |  j = mid − 1
```

proof of correctness: If the midpoint of interval (i, j) is the peak, then we just return the element in midpoint. Otherwise, we claim that interval (i, j) in each iteration contains at leat one peak. Assume (i, j) does not contain a peak. Then $A[i]$ to $A[j]$ is non-decreasing or decreased first then increased.

Assume (i, j) is non-decreasing. If $j = n-1$, $A[n-1]$ is a peak, a contradiction. If $j \neq n-1$, then $A[j+1] \leq A[j]$. So $A[j]$ is a peak, a contradiction.

Assume (i, j) decreased first then increased. If $i = 0$, then $A[i]$ is a peak, a contradiction. If $i \neq 0$, then $A[i-1] \leq A[i]$ by the algorithm above. Hence $A[i]$ is a peak, a contradiction.

Therefore, interval (i, j) in each iteration contains at leat one peak. Hence the algorithm will always terminate and return the correct output.
runtime analysis: Each time we reduce the size of the interval by a half. So $T(n) = \log_2 n \cdot O(1) = O(\log n)$.

3. [*21 marks*] Consider the following problem: given a sequence of n 2×2 matrices $A_1 \dots A_n$ where each entry in A_i is either 0 or 1, we want to compute the product $A_1 A_2 \dots A_n$. Note that the result is a 2×2 matrix, where the four entries may be very large integers.

   (a) [*3 marks*] Explain why each entry of the final product requires at most $O(n)$ bits.

   $[A_1 \dots A_n]_{i,j} \leq 2^n$ for all $i, j \in \{1, 2\}$. Clearly, 2^n can be represented by $n + 1$ bits. Hence the final product requires at most $O(n)$ bits.

   (b) [*6 marks*] Describe and analyze a simple algorithm to solve this problem, by iteratively multiplying the current product with the next matrix A_i for i from 1 to n. Show that this algorithm runs in $O(n^2)$ time.

   We just multiply matrices one by one iteratively and return the final answer.

Algorithm 2: MProd$(A_1, \dots, A_n)$

```
1 prod = A_1
2 for i = 2...n do
3 |   temp = prod
4 |   prod[1][1] = temp[1][1] × A_i[1][1] + temp[1][2] × A_i[2][1]
5 |   prod[1][2] = temp[1][1] × A_i[1][2] + temp[1][2] × A_i[2][2]
6 |   prod[2][1] = temp[2][1] × A_i[1][1] + temp[2][2] × A_i[2][1]
7 |   prod[2][2] = temp[2][1] × A_i[1][2] + temp[2][2] × A_i[2][2]
8 return prod
```

proof of correctness: This algorithm follows the standard process of matrices multiplication. Hence it is correct.

Runtime analysis: Each multiplication takes $O(1)$ time. Since we are always multiplying a number with 1. Since each entry of the final product is at most $O(n)$ bits, so we are adding 2 $O(n)$ bits number. Hence each addition takes $O(n)$ time. So the total runtime of one iteration is $4O(n) + 8O(1) = O(n)$. Hence the total runtime is $O(n^2)$.

(c) [*12 marks*] Next describe and analyze a recursive algorithm to solve this problem. Show that this algorithm runs in $O(n^{1.59})$ time. [Hint: you may use Karatsuba and Ofman's integer multiplication algorithm as a subroutine.]. You may assume n is a power of 2.

We denote Karatsuba and Ofman's integer multiplication algorithm as $ko\_mult(a, b)$ where a, b are two numbers.

We use a devide & conquer algorithm to calculate the product of n matrices.

Algorithm 3: DC-MProd($A_1, \ldots, A_n$)

MatricesMult(A, B):
  Input: A, B are 2×2 matrices.
  Let $prod = A \times B$
  $prod[1][1] = ko\_mult(A[1][1], B[1][1]) + ko\_mult(A[1][2], B[2][1])$
  $prod[1][2] = ko\_mult(A[1][1], B[1][2]) + ko\_mult(A[1][2], B[2][2])$
  $prod[2][1] = ko\_mult(A[2][1], B[1][1]) + ko\_mult(A[2][2], B[2][1])$
  $prod[2][2] = ko\_mult(B[2][1], B[1][2]) + ko\_mult(A[2][2], B[2][2])$
  return *prod*
DC-MProd($A[A_1, \ldots, A_n]$):
  if $|A| = 1$ **then**
    return $A[0]$
  $temp1 = DC\text{-}Mprod([A_1, \ldots, A_{n/2}])$
  $temp2 = DC\text{-}Mprod([A_{n/2+1}, \ldots, A_n])$
  return $MatricesMult(temp1, temp2)$

proof of correctness: Since n is a power of 2, we can always calculate the product of first $\frac{n}{2}$ matrices and last $\frac{n}{2}$ matrices and return the multiplication of them.

Runtime analysis: By part (a), the final product requires at most $O(n)$ bits. So the entries of product of any two matrices are at most $O(n)$ digits. The runtime of $ko\_mult(a, b)$ is $O(n^{1.59})$. Hence, the runtime of $MatricesMult(A, B)$ is $8 \cdot O(n^{1.59}) + 4 \cdot O(n) = O(n^{1.59})$. So, $T(n) = 2T(\frac{n}{2}) + O(n^{1.59}) = O(n^{1.59})$ by Master Method($a = 2, b = 2, d = 1.59, a < b^d$).

4. [*22 marks*] Consider the following problem: We are given a set R of n_1 rectangles in 2D, where the sides of the rectangles are parallel to the x- and y-axes (i.e., the rectangles are not rotated) and the rectangles do not overlap. We are also given a set P of n_2 points in 2D. Each point $p \in P$, p may or may not be contained inside a rectangle but if it is covered, it is covered with a unique rectangle because of the nonoverlapping assumption. Our goal is to find for every point $p \in P$, the rectangle $r_p \in R$ that contains p. Let $n = n_1 + n_2$. This is not necessary but for simplicity, you can assume that all coordinates of points and corners of rectangles are distinct.

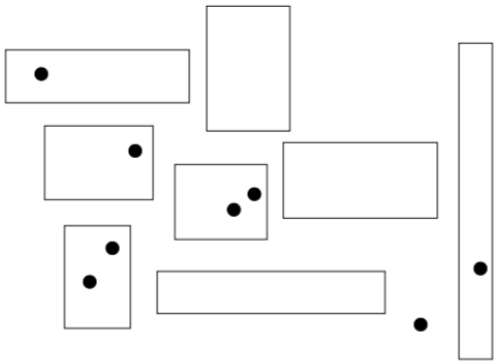

Figure 1: Example of the general case of the problem.

(a) [*10 marks*] First consider the special that when all rectangles of R are assumed to intersect a given horizontal line ℓ. Now design an $O(n \log(n))$-time algorithm for this special case.

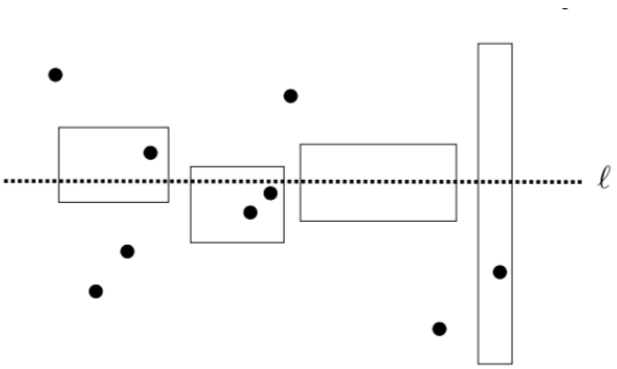

Figure 2: Example of the special case from Q3a.

Let $r \in R$, define $r.topLeft$ to be the coordinate of the top left point in r and $r.botRight$ to be the coordinate of the bottom right point in r.

We first sort R according to the x-coordinate of tge top left point of each rectangle. Then find the rectangle that contain $p \in P$ iteratively. For every $p \in P$, we use an algorithm that is similar to binary search to find the rectangle that contains p or conclude that p is not in any rectangle.

Algorithm 4: Find-Rec-l(P, R)

```
Rec-Cand(R, p):
    Iuput: An array of rectangles sorted by the x-coordinate of the top left point of each
     rectangle.
    Output: The index of the rectangle that may contain p.
    lo = 1
    hi = |R|
    while lo ≤ hi do
        mid = ⌊(lo + hi)/2⌋
        if R[mid].topLeft.x = P[i].x then
            return mid Break
        else if R[mid].topLeft.x < P[i].x then
            lo = mid + 1
        else
            hi = mid − 1
    return hi
Find-Rec-l(P, R):
    Ouput: An array A where A[i] is a rectangle that contains P[i] or − if P[i] is not in
     any rectangle.
    sort(R) such that r_i.topLeft.x ≤ r_j.topLeft.x for all i < j, r_i, r_j ∈ R
    for i = 1 ··· n_2 do
        j = Rec-Cand(R, P[i])
        if j = 0 or
        not (R[j].topLeft.x ≤ P[i].x ≤ R[j].botRight.x and
         R[j].topLeft.y ≤ P[i].y ≤ R[j].botRight.y) then
            A[i] = −
        else
            A[i] = R[j]
    return A
```

proof of correctness: We prove that Rec-Cand(R, p) always terminates and returns the index of a rectangle that we only need to compare.
The while loop in Rec-Cand(R, p) always by the properties of binary search.
Since all rectangles intersect with a horizontal line ℓ. Then there can't exist a rectangle r such that $r'.topLeft.x \leq r.topLeft.x \leq r'.botRight.x$, $r'.topLeft.y \leq r.botRight.y$ or $r'.botRight.y \geq r.topLeft.y$ for some $r' \in R$. So if $R[i].topLeft.x \leq p.x \leq R[i+1].topLeft.x$ for some point p, p is either in $R[i]$ which has the largest top left x-coordinate

that is smaller than $P.[i].x$ or not in any rectangle.
If $P[i].x = R[j].topLeft.x$ for some $1 \leq j \leq n$. By the proof above, we only have to check whether $P[i]$ is in $R[j]$.
If $P[i].x \neq R[j].topLeft.x$ for all $1 \leq j \leq n$. Since we take the midpoint every iteration, $1 \leq hi \leq n$. We claim $R[hi]$ has the largest top left x-coordinate that is smaller than $P[i]$. If $P[i].x$ is greater than all rectangle's top left x-coordinate, lo would be $n + 1$ evetually and hi is always n. Hence the statement holds. So we can assume $P[i].x$ is smaller than some x-coordinate in R. Then at some point, hi is updated to $mid - 1$ where the element in the new value of hi has a smaller x-coordinate since $p[i].x$ is not in R but the element in mid has a greater x-coordinate. Only lo is updated after that points since $mid \leq hi$ and $R[hi].topLeft.x < P[i].x$. Hence the claim holds. Bt the previous proof we only need to check $R[hi]$.
Runtime analysis: Rec-Cand(R, p) takes $O(\log n_1)$ time. Hence Find-Rec-l(P, R) has runtime $O(n_1 \log n_1) + n_2 O(\log n_1) = O(n \log n)$.

(b) [*12 marks*] Now give an $O(n \log^2(n))$-time algorithm (or better) for the general case of the problem. (Hint: use divide-and-conquer and use part (a) as a subroutine).
We start with a set A that stores the answers and $A[i] = -$ for all i at the beginning. We put the x-coordinates of points and $\frac{r.topLeft.y+r.botRight.y}{2}$ into an array and sort them. Let ℓ be the median value of B. We determine whether a point is in the rectangles that intersect with ℓ by using the algorithm in part (a). Then divide the problem into two subproblems with size $\leq \frac{n}{2}$. At each recursion, we update the set that contains the answer.

Algorithm 5: Find-Rec(P, R)

```
1  Find-Rec-B(B):
2      Input: B = P ∪ R
3      Let A be a set that stores the answers.
4      if |B| = 0 or (|B| = 1 and B[1] is a point) then
5          return
6      ℓ = (r.topLeft.y + r.botRight.y)/2 if B[n/2] is a rectangle or B[n/2].y if B[n/2] is a
        point.
7      for i = 1, ··· , n do
8          if B[i] is a rectangle and B[i].topLeft.y ≤ ℓ ≤ B[i].botRight.y then
9              R_1 = R_1 ∪ B[i]
10             B = B \ B[i]
11     A = Find-Rec-l(P, R_1)
12     Remove the points in B that are in a rectangle in set R_1 and the points on ℓ
13     n' = |B|
14     update A based on Find-Rec(B[1, ··· , ⌊n'/2⌋])
15     update A based on Find-Rec(B[⌊n'/2⌋ + 1, ··· , n'])
16     return A
17 Find-Rec(P, R):
18     B = P ∪ R
19     Sort B according to p.y or (r.topLeft.y + r.botRight.y)/2 in ascending order.
20     Find-Rec-B(B)
```

Proof correctness: We compare all the points with every rectangle intersects with ℓ. If a point is in one of the rectangle intersects with ℓ, we can dorp this point since rectangles do not overlap. In addition, we can also drop the rectangles on ℓ since we have already compared them with all points. If a point p's y-coordinate is greater than ℓ and $(r.topLeft.y + r.botRight.y)/2$ is smaller than ℓ for a rectangle, we have already compared them by calling Find-Rec-l from part (a). So the division does not lose any solution. Eventually, we will remove all the rectangle or leave with only one point in B. The algorithm always terminates. Hence it is correct.
Runtime analysis: $T(n) = O(n \log n) + T'(n)$ where $T'(n)$ is the runtime of Find-Rec-B.

Assume $n > 1$, we have at most n points and rectangles if we don't remove any rectangles or points. Then B is divided into two parts where each part has length at most $\frac{n}{2}$. $T'(n) \leq 2T(n/2) + O(n) + O(n \log n) = 2T'(n/2) + O(n \log n)$
So,

$$\begin{aligned}
T'(n) &= 2T'(n/2) + O(n \log n) \\
&\leq \sum_{i=1}^{\log n} c_i \cdot i \frac{n}{i} \log(\frac{n}{i}) \quad \text{by the definition of big-O} \\
&\leq cn \sum_{i=1}^{\log n} \log(\frac{n}{i}) \quad c = max\{c_1, \cdots, c_{\log n}\} \\
&= cn \sum_{i=1}^{\log n} \log n - \log i \\
&\leq cn \sum_{i=1}^{\log n} \log n \\
&= cn \log^2 n \\
&= O(n \log^2 n)
\end{aligned}$$

Therefore, $T(n) = O(n \log n) + O(n \log^2 n) = O(n \log^2 n)$

5. [*15 marks*] Your friend has invited you to play a game which goes as follows. There are two sets of sticks: (i) R that consists of n red sticks; and B that consists of n blue sticks. Each stick has a height (you can assume they are distinct for simplicity). Each player pairs red and blue sticks (so creates n pairs of stick, where each pair contains one red and one blue stick) and the winner is the one that minimizes the "sum, across all pairs, the differences in heights. So suppose you have paired the sticks and in your i'th pair the height of red stick is r_i and the height of the blue stick be b_i, then you incur $|r_i - b_i|$ penalty. Design an algorithm that runs in $O(n \log n)$ time (or better) that pairs the sticks with the minimum possible penalty. In other words, your algorithm minimizes the following sum across all possible pairings:

$$\Sigma_{i=1\ldots n}|r_i - b_i|$$

We sort two arrays and pair $R[i], B[i]$

Algorithm 6: PairSticks(R, B)

```
Output: A
Sort R by heights in ascending order.
Sort B by heights in ascending order.
for i = 1, ··· , n do
    A[i] = (R[i], B[i])
return A
```

Proof of correctness: We claim that the algorithm above is optimal.
Let $A' = (r_1, b_1), (r_2, b_2), \cdots, (r_n, b_n)$ be the set that a random algorithm returns. There exists two pairs (r_i, b_i) and (r_j, b_j) such that $r_i \leq r_j$ and $b_j \leq b_i$. Let a, c be the one with smaller height in the two pairs and b, d be the one with greater height in the two pairs. We have $a \leq c \leq d \leq b$. If a and c do not have same color, we can swap a and c to get a even smaller penalty. If a and c have same color, then the penalty of the pairs (a, d) and (c, b) is $d - a + b - c = d - c + c - a + b - d + d - c = (b - d + d - c + c - a) + (d - c) = b - a + d - c$. $b - a + d - c$ is the penalty of (a, b) and (c, d).
Hence the algorithm above is optimal. **Runtime analysis:** Sorting R and B take $2O(n \log n) = O(n \log n)$ time. Pairing each element takes $O(n)$ time. hence the overall runtime is $O(n \log n)$.