

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: Ethan YanWisc id: 9084649137

Divide and Conquer

1. Kleinberg, Jon. *Algorithm Design* (p. 248, q. 5) Hidden surface removal is a problem in computer graphics where you identify objects that are completely hidden behind other objects, so that your renderer can skip over them. This is a common graphical optimization.

In a clean geometric version of the problem, you are given n non-vertical, infinitely long lines in a plane labeled $L_1 \dots L_n$. You may assume that no three lines ever meet at the same point. (See the figure for an example.) We call L_i "uppermost" at a given x coordinate x_0 if its y coordinate at x_0 is greater than that of all other lines. We call L_i "visible" if it is uppermost for at least one x coordinate.

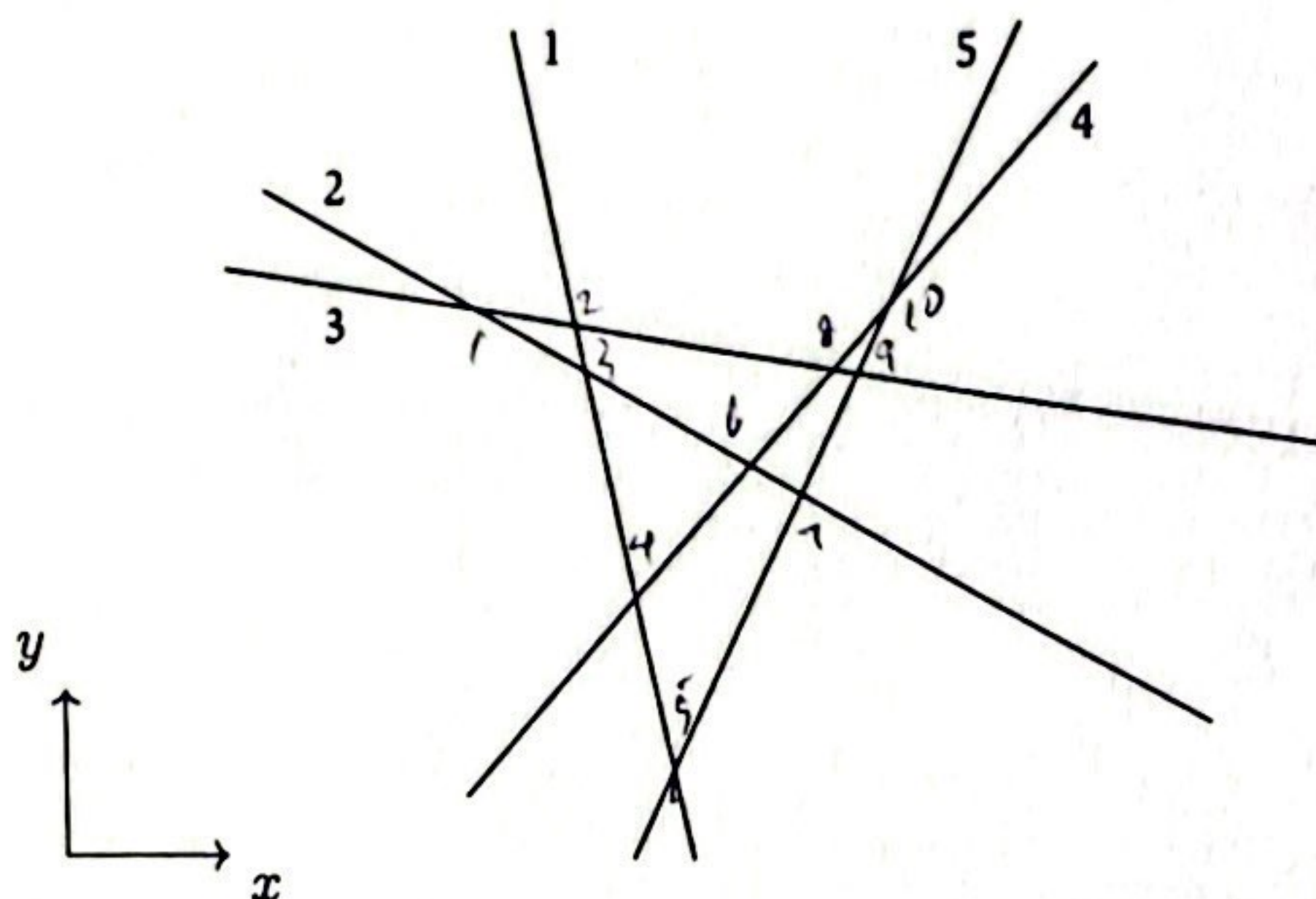


Figure 5.10 An instance of hidden surface removal with five lines (labeled 1-5 in the figure). All the lines except for 2 are visible.

A: 1, 3, 2 B: 4, 5
 1: $-\infty$
 2:
 3:
 4: 4
 5: 5

- (a) Give an algorithm that takes n lines as input and in $O(n \log n)$ time returns all the ones that are visible.

For any 2 lines, if non-parallel, they intersect at a point, aside from that point, only 1 line will be visible (whichever has greater y -coordinate).

For L_1, L_2, \dots, L_n , at any one point, only 1 line is visible, except intersections.

Perform ~~merge~~ merge sort by each line's x -coordinate of first visibility (ie x -coordinate when line first becomes visible).

- Initialize all to have $-\infty$ at start. Split half of ~~the~~ set of lines down recursively.
- To merge line sets $A \hat{=} B$, compare $A[i] \hat{=} B[i]$, find x -coordinate of intersection.
- Line that is higher on the left still has $-\infty$ label.
- When noting a line A_i over B_i , then we know next visible line is either A_{i+1} with its existing x -coordinate label or B_{i+1} at where B_{i+1} intersects A_i . This can be done in constant time, so merge sort done in $n \log n$.

Sort Input: A, output: sorted A
 | $|A| = 1$ then return A
 | $A = \text{sort}(\text{first half } A)$
 | $B = \text{sort}(\text{second half } A)$
 | return compare(A, B).

compare input: A, B | output: C
 | iteratively compare first elt of A & B
 | if $A_i > B_i$, then find next
 | visible line from A_{i+1}, B_i, B_{i+1} (constant time)
 | append to C.

(b) Write the recurrence relation for your algorithm.

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn ; T(1) = c$$

identical to recurrence relation for mergesort, so $O(n \log n)$.

$2T\left(\frac{n}{2}\right)$: we make 2 recursive calls on each half of input array.

cn : merge step worst case compares every element in array (between A & B), requires n steps.

(c) Prove the correctness of your algorithm.

We will do a proof by strong induction on length k of input sequence.

IH: On input of array of length k , algo correctly returns sorted array of visible lines. Assume holds for $1 \leq k \leq n$.

Base case: For $k=1$, only 1 line, so line is visible throughout and returned as only visible line.

Divide step recurses on 1st & 2nd halves of array, of which both are an array of length $< n$. So by IH, the recursive call returns a sorted array of both 1st & 2nd half. Now need show compare and merge step is correct.

we iterative compare leftmost coordinates for A & B, append leftmost coordinate associated line if ~~the~~ it is next visible. Thus, this algorithm would return sorted output.

Therefore, strong induction holds.

2. In class, we considered a divide and conquer algorithm for finding the closest pair of points in a plane. Recall that this algorithm runs in $O(n \log n)$ time. Let's consider two variations on this problem:

- (a) First consider the problem of searching for the closest pair of points in 3-dimensional space. Show how you could extend the single plane closest pairs algorithm to find closest pairs in 3D space. Your solution should still achieve $O(n \log n)$ run time.


Sort by x, y and z coordinates.
 Split space along x -axis by half using sorted x -coordinates. Recurse.
 Consider points that may cross partition. As is in 2D case, there are constant number of points that could be ~~#~~ close enough to ~~cause~~ be closest pair. Thus, algorithm still runs in $O(n \log n)$.

- (b) Now consider the problem of searching for the closest pair of points on the surface of a sphere (distances measured by the shortest path across the surface). Explain how your algorithm from part a can be used to find the closest pair of points on the sphere as well.

Apply algorithm in (a) directly. Closest points on surface are closest points in 3D space as well.

- (c) Finally, consider the problem of searching for the closest pair of points on the surface of a torus (the shape of a donut). A torus can be thought of taking a plane and "wrap" at the edges, so a point with y coordinate MAX is the same as the point with the same x coordinate and y coordinate MIN. Similarly, the left and right edges of the plane wrap around. Show how you could extend the single plane closest pairs algorithm to find closest pairs in this space.

Use same algorithm as 2D. But, ^{instead of} considering points that cross partition, consider potential closest pair being interior to left half, interior to right half, cross middle, which can still be done in constant time. Therefore, can use same algorithm as per 2D scenario, with time complexity of $O(n \log n)$.



3. Erickson, Jeff. *Algorithms* (p. 58, q. 25 d and e) Prove that the following algorithm computes $\gcd(x, y)$ the greatest common divisor of x and y , and show its worst-case running time.

```

BINARYGCD(x, y):
  if x = y:
    return x
  else if x and y are both even:
    return 2 * BINARYGCD(x/2, y/2)
  else if x is even:
    return BINARYGCD(x/2, y)
  else if y is even:
    return BINARYGCD(x, y/2)
  else if x > y:
    return BINARYGCD((x-y)/2, y)
  else
    return BINARYGCD(x, (y-x)/2)

```

Base case: $x=y=1$, algo returns 1 ✓

IH: Assume for $x \leq x_k \wedge y \leq y_k$ holds.

Consider each case: if $x=y$: x returned, correct ✓

if both even: then both $x \wedge y$ can be divided by 2, since GCD divided by 2 as well, multiplied by 2, correct ✓.

if either even: then cannot divide odd num by 2, GCD not factor of 2, so can divide even num by 2, does not change GCD ✓

if both odd: $x-y$ or $y-x$ is even, so algo can divide by 2 ✓

In all cases, algo recurses on smaller number, thus returns accurate GCD by IH.


~~Worst case running time~~

Worst case running time: $O(\log n)$.

4. Use recursion trees or unrolling to solve each of the following recurrences.

(a) Asymptotically solve the following recurrence for $A(n)$ for $n \geq 1$.

$$A(n) = A(n/6) + 1 \quad \text{with base case} \quad A(1) = 1$$


 $\log_6 n$ layers so $A(n) = \sum_{k=1}^{\log_6 n} 1 = \Theta(\log n)$

(b) Asymptotically solve the following recurrence for $B(n)$ for $n \geq 1$.

$$B(n) = B(n/6) + n \quad \text{with base case} \quad B(1) = 1$$

$$B(n) = \sum_{k=1}^{\log_6 n} \frac{n}{6^k} = n \sum_{k=1}^{\log_6 n} \left(\frac{1}{6}\right)^k = \frac{1 - \left(\frac{1}{6}\right)^{\log_6 n}}{1 - \frac{1}{6}} (n) = \frac{1 - \frac{1}{n}}{1 - \frac{1}{6}} (n)$$

$$= (n) \frac{n-1}{n} \left(\frac{6}{5}\right) \in \theta(n)$$

(c) Asymptotically solve the following recurrence for $C(n)$ for $n \geq 0$.

$$C(n) = C(n/6) + C(3n/5) + n \quad \text{with base case} \quad C(0) = 0$$

$$C(n) = \sum_{k=0}^{\infty} \left(\frac{23}{30}\right)^k n$$

$$= n \left(\frac{1}{1 - \frac{23}{30}} \right)$$

$$= \frac{30}{7} n \in \theta(n)$$

(d) Let $d > 3$ be some arbitrary constant. Then solve the following recurrence for $D(x)$ where $x \geq 0$.

$$D(x) = D\left(\frac{x}{d}\right) + D\left(\frac{(d-2)x}{d}\right) + x \quad \text{with base case} \quad D(0) = 0$$

$$D(x) = \sum_{k=0}^{\infty} \left(1 - \frac{1}{d}\right)^k x$$

$$= x \frac{1}{1 - \left(1 - \frac{1}{d}\right)} = dx \in \theta(x).$$