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: Yongsang Park Wisc id: 908 022 3069

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. 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.

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

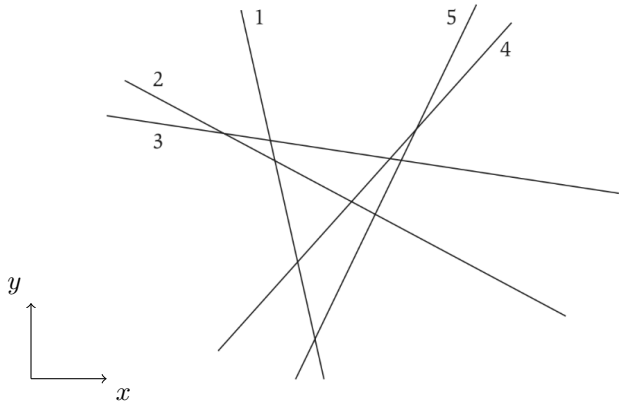


Figure 5.10 An instance of hidden surface removal with five lines (labeled *1–5* in the figure). All the lines except for *2* are visible.

Solution:

Input: n lines array L
Output: visible lines
VL
Sort L by slope
if slope < 0
  Let S = {L1, L2}
  A = point(x1, y1) (intersection of L1 & L2)
  while slope of Li < 0
    if Li is above A
      Add Li into S
      update A(xi-1, yi-1) (Li-1 & Li)
    end
    i++
  end
end
Add Li+1 into S
A = point(xi, yi) (cross of Li & Li+1)
while L is not empty
  if Li is above A
    Add Li into S
    update A(xi-1, yi-1)
  end
  i++
end
return S

Sorting costs O(nlogn)
each while loop costs O(n)
O(nlogn) + 2O(n)
= O(nlogn)

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.

   Solution:
   In 3D space, find the mid point of z. Create a plane which is parallel to xy plane.
   Find the shortest distance on each side, d, and find the points which are within d from the plane.
   Return min (d, distance near plane)

   (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.

   Solution: We can use the same algorithm.
   We can set points on the surface of a sphere into a 3D space.

   (c) Finally, one way to approximate the surface of a sphere is to take a plane and "wrap" at the edges, so a point at x coordinate 0 and y coordinate MAX is the same as x coordinate 0 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.

   Solution:
   Sort the point by x first, and divide the points by $y \geq$ plane.
   Do the divide & Conquer as same as closest pair in planes, and find the minimum length

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 )
```

Solution:
When $x=y$, it returns x which is GCD.
When both are even, 2 is an element of GCD, so $2 \cdot$ BINARYGCD($x/2$, $y/2$) makes sense.
When one of them is even, 2 is not an element of GCD, so BINARYGCD($x/2$, y) or BINARYGCD(x, $y/2$) makes sense.
When both of them are odd, subtracting one from the other is the same as Euclidean algorithm

The worst running case is when x, y are odd numbers, always $x \neq y$ till the end, and $(x-y)/2$ is always odd.
It is $O(n^2)$

4. Here we explore the structure of some different recursion trees than the previous homework.
   (a) Asymptotically solve the following recurrence for $A(n)$ for $n \geq 1$.

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

Solution:
$A(n) = A(n/6) + 1$
$= (A(n/6^2) + 1) + 1$
$= \cdots$
$= A(n/6^k) + k$

Basecase: $A(1) = 1$
$A(n/6^k) = 1$, then
$n/6^k = 1$
$\Rightarrow k = \log_6 n$

$A(n) = A(1) + \log_6 n$

So, $O(\log n)$

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

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

Solution:

$B(n) = B(n/6) + n$

$= (B(n/6^2) + \frac{n}{6}) + n = B(n/6^2)$

$= \cdots$

$= B(n/6^k) + \sum_{i=0}^{k-1} \frac{n}{6^i} = B(n/6^k) + n \cdot \frac{1-(\frac{1}{6})^{k+1}}{\frac{5}{6}}$

Base case $B(1) = 1 = B(n/6^k)$

$n/6^k = 1 \Rightarrow k = \log_6 n$

$B(n) = B(1) + \frac{6}{5}n \cdot (1 - 6 \cdot 6^{-\log_6 n})$

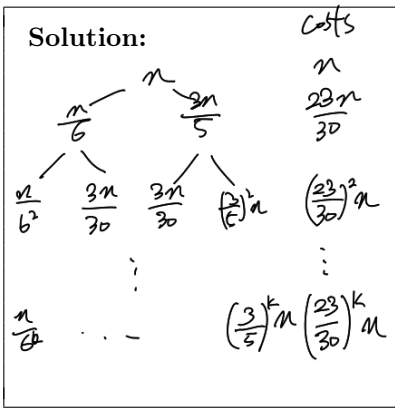
$= 1 + \frac{6}{5}n + \frac{6}{5}n \cdot (-\frac{1}{n}) = \frac{6}{5}n - 5$

$\Rightarrow O(n)$

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

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

Solution:



$C(n) = \sum_{i=0}^{k} (\frac{23}{30})^i n = n \cdot \frac{1-(\frac{23}{30})^k}{1-\frac{23}{30}}$

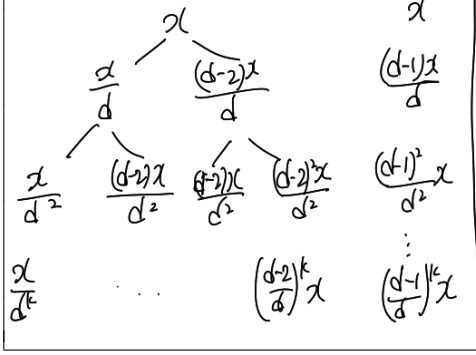
Asymptotically, when $k \to \infty$, $n \frac{1-(\frac{23}{30})^k}{1-\frac{23}{30}} \to \frac{30}{7}n$

So, $C(n) = O(n)$

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

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

Solution:



$D(x) = \sum_{i=0}^{k} (\frac{d-1}{d})^i x = x \cdot \frac{1-(\frac{d-1}{d})^k}{1-\frac{d-1}{d}} = dx(1-(\frac{d-1}{d})^k)$

$\lim_{k \to \infty} dx(1-(\frac{d-1}{d})^k) = dx$

So, $D(x) = dx \in O(x)$

5. Implement a solution in either C, C++, C#, Java, or Python to the following problem.

   Suppose you are given two sets of n points, one set $\{p_1, p_2, \ldots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \ldots, q_n\}$ on the line $y = 1$. Create a set of n line segments by connecting each point p_i to the corresponding point q_i. Your goal is to develop an algorithm to determine how many pairs of these line segments intersect. Your algorithm should take the $2n$ points as input, and return the number of intersections. Using divide-and-conquer, you should be able to develop an algorithm that runs in $O(n \log n)$ time.
   Hint: What does this problem have in common with the problem of counting inversions in a list?

   Input should be read in from stdin. The first line will be the number of instances. For each instance, the first line will contain the number of pairs of points (n). The next n lines each contain the location x of a point q_i on the top line. Followed by the final n lines of the instance each containing the location x of the corresponding point p_i on the bottom line. For the example shown in Fig 1, the input is properly formatted in the first test case below.

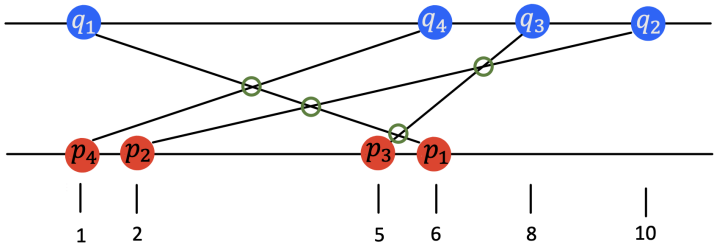


Figure 1: An example for the line intersection problem where the answer is 4

Constraints:

- $1 \leq n \leq 10^6$
- For each point, its location x is a positive integer such that $1 \leq x \leq 10^6$
- No two points are placed at the same location on the top line, and no two points are placed at the same location on the bottom line.
- Note that in C\C++, the results of some of the test cases may not fit in a 32-bit integer. If you are using C\C++, make sure you use a 'long long' to store your final answer.

Sample Test Cases:

input:
2
4
1
10
8
6
6
2
5
1
5
9
21
1
5
18
2
4
6
10
1

expected output:
4
7