CS 577 # Assignment 6 – Divide and Conquer 2 Spring 2023

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: ______________________ Wisc id: ______________________

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.

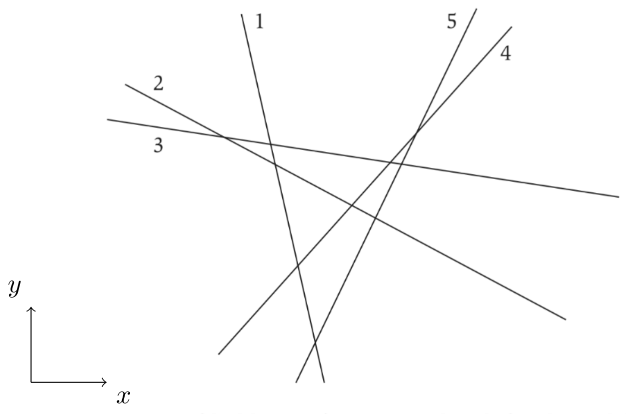

Figure 5.10 An instance of hidden surface removal with five lines (labeled *1–5* in the figure). All the lines except for *2* are visible.

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

Solution: Given two non-parallel, non-vertical, infinite lines they cross at a single point, and each is hidden from that point out to infinity in one direction or the other. As a result, a visible line is visible over a single (possibly infinite) interval. Additionally, given a set of lines $1 \dots k$, exactly one of those lines is visible over the others at any x coordinate except the coordinates where two intersect.

We conduct a modified mergesort on the n lines.

Label each line with the start of its visible interval, initially $-\infty$. To merge two sets of lines A and B, assume that they have been pre-sorted so that the first elements are the lines with the left-most visible intervals.

Starting with the first line in A and B, in constant time, find the coordinate x_0 where these two lines cross. The one that is higher on the left is the new first element of the merged set with label $-\infty$. If we most recently noted visible line A_i over B_i with crossing point x_i, then the next visible line is either A_{i+1} at its current label, or B_j for some $j \geq i$ at where A_i and B_j cross. Note that we can ensure that each line in both sets is "checked" at most once by doing a sweep over the set of lines. That is, if B_i is considered not visible, then we do not recheck it in the future again. So the entire merge process can be done linearly.

(b) Write the recurrence relation for your algorithm.

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

(c) Prove the correctness of your algorithm.

Solution: First, the merge process is correct as it considers all three possibility of the next visible line.

Claim: Given a set of lines as input, the algorithm returns a sorted list of visible lines.
Proof: By strong induction on k, the size of the set of lines.

- Base case: $k = 1$, correctly returns the line as it is the only visible line. $k = 2$, correctly returns both lines in order by definition of the merge process.
- Inductive step: By induction hypothesis, the two sets of lines A and B are sorted. By definition of the merge process, the correct sorted list of visible lines will be returned.

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: Our algorithm proceeds just as in the 2D case. We get a sorted list of the points by x, y, and now also z coordinate, and split the space in half using the x coordinate list, recursing on each half.

   Considering crossing pairs, note that by the same logic as in the 2D case there are only a constant number of points that may be "close enough" that we have to check if they are the closest pair, which is all we need to show that we can still do this in $O(n \log n)$ time.

   (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: The two closest points as measured along the surface are also the two closest points as measured straight through the interior of the sphere, so we can apply the algorithm from part a directly to our set of points, and it will return the correct answer.

   (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.

   Solution: The difference between the "wrapped" plane and the standard closest pairs problem is that we don't immediately have the ability to sort points, and we must worry about crossing pairs on all sides.

   We'll mostly use the exact same 2D algorithm, but at the top level we need to do something to address the wrapping. At this level, either the closest pair is interior to the left half, interior to the right half, crosses the middle, crosses the top-bottom edge, crosses the left-right edge, or may cross the top-bottom edge and one of the two other boundaries. We can handle each crossing pairs problem in the same way we do for the standard 2D case without increasing the asymptotic time required.

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: Assume for induction that BINARYGCD works for all $x \leq x_0$, $y \leq y_0$ (except x_0, y_0). We'll show that this implies BINARYGCD(x_0, y_0) as well. For a base case, note that BINARYGCD(1,1)=1 as it should.

We proceed for each case in the conditional and consider the mathematics. If $x = y$, then x is indeed the GCD. If both are even, then we can divide x, y by two and get the GCD divided by 2 as well, exactly as the algorithm does. If only x or y is even, then the GCD cannot itself be a factor of 2 and so we can divide out the 2 without changing the GCD–exactly as the algorithm does. Otherwise, since the GCD divides both x and y, it must divide $x - y$ as well. Note that both x and y must be odd, so $x - y$ must be even, so the algorithm can correctly divide by 2 as well. We've covered all possible combinations of x, y being even or odd, and in all cases correctly reduced BINARYGCD(x, y) to a recursive call on a strictly smaller case.

Assume that the running time of division and subtraction is $O(1)$. The worst-case running time of BINARYGCD is $O(log(x) + log(y)) = O(log(xy))$. In the worst-case, BINARYGCD will divide either x or y by 2 at each step. Alternatively, the solution can be written as $O(log(max(x, y))$

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:
Recursion tree is a path descending from the root. The work in each layer is 1. The number of layers is $\log_6(n)$. Total work is then

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

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

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

Solution:
Recursion tree is a path descending from the root. The work in each layer is one sixth that of the previous layer, where the root has work n. The number of layers is $\log_6(n)$. Total work is then

$$B(n) = \sum_{k=0}^{\Theta(\log_6(n))} \frac{n}{6^k} = n\frac{1 - \frac{1}{6^{\log_6(n)}}\frac{1}{6}}{1 - 1/6} = n\frac{1 - \frac{1}{6n}}{1 - 1/6} = n\frac{6n-1}{6n}\frac{6}{5} = \frac{6n-1}{5} \in \Theta(n)$$

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

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

Solution: The total work in each layer of the tree is $\frac{1}{6} + \frac{3}{5} = \frac{5}{30} + \frac{18}{30} = \frac{23}{30}$ times the previous layer, with n at the root. The value of $C(n)$ is then

$$C(n) = \sum_{k=0}^{\infty} n(\tfrac{23}{30})^k = \frac{n}{1 - \frac{23}{30}} = \frac{n}{7/30} = \frac{30n}{7} \in \Theta(n)$$

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

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

Solution: The total work in each layer of the recursion tree is $(d-1)/d$ times the previous layer, with x at the root. The value of $D(x)$ is then

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

5. Implement a solution in either C, C++, C#, Java, Python, or Rust 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.

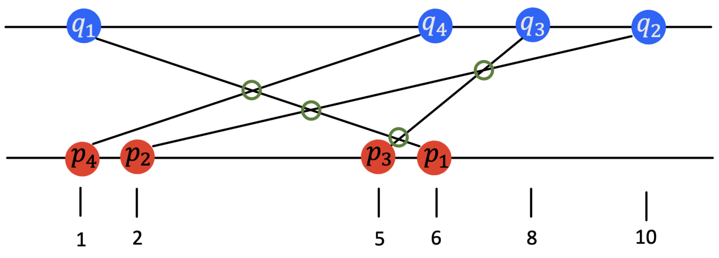

Figure 1: An example for the line intersection problem where the answer is 4

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
```