CS 577 # Assignment 6 – Divide and Conquer Spring 2024

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: Ki Min Kang Wisc id: 908 404 5062

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.

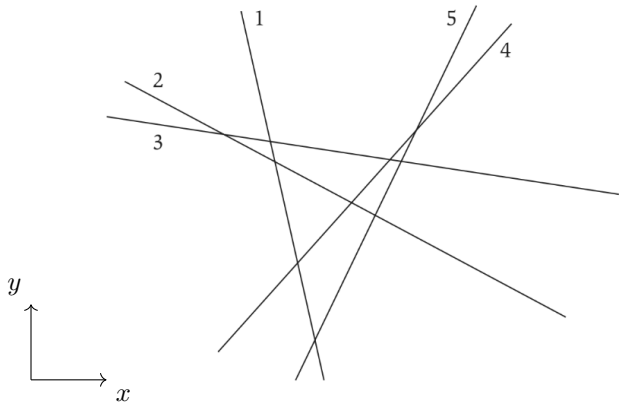

Figure 5.10 An instance of hidden surface removal with five lines (labeled *1–5* in the figure). All the lines except for *2* are visible.

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

Each line is labeled with the starting point of its visible interval, initially set to $-\infty$.
This label represents the x-coordinate from which the line is visible.

Before merging, assume that we have two sets of lines, A and B.

Merging Process:
Start by comparing the first lines of both sets A and B.
Find the x-coordinate x_0, where these two lines intersect.
Then, determine which line is higher at x_0. The higher line will be
the one that remains visible at and after this intersection point.

If the last noted visible line was from set A over a line from set B at their intersection x_i,
then the next visible line will be the next line in set A or the next line in set B.

The comparison to determine which line is higher at their intersection points is done in constant time.
Continue this process linearly through the sets of lines.

The algorithm proceeds through all the lines, updating the visible intervals
based on the intersections, and concludes with a set of visible lines.

(b) Write the recurrence relation for your algorithm.

The recurrence relation of the algorithm is:

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

This reflects the divide-and-conquer strategy, where the problem is divided into two smaller problems of half size ($2T(\frac{n}{2})$) and the merging process, which is linear ($O(n)$)

(c) Prove the correctness of your algorithm.

Proof by induction.

Base Case:
For a single line ($K=1$), it is obviously visible since there is no other line to obscure it.

Inductive Step:
Assume that the algorithm works for K lines.
When the K+1-th line is added, it is compared with the existing K lines during the merge process. If the new line is higher at any intersection point, it becomes the next visible line.
Otherwise, the previous logic applies, maintaing the visibility order.

Therefore, the algorithm correcely identifies all visible lines by maintaing a sorted list of lines by induction.

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.

We begin by sorting all points in the 3D space according to their X-coordinate.
This is similar to the first step in the 2D case.
Next, split the set of points into two halves.
Apply the same algorithm recursively to find the closest pair of points in each of the two halves.
After the recursive step, check for pairs of points where one point is in one half and the other is in the opposite half.
Assume that there is only a constant number of points that need to be checked around the dividing plane.
Finally, compare the closest pairs found in the recursive steps.

The algorithm runs in O(nlogn) time, similar to 2D case.

   (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.

To find the closest pair of points on a sphere's surface, we can leverage the fact that the shortest path on the surface of a sphere between two points is directly related to the straight-line distance through the sphere's interior.

Therefore, we can use the same algorithm from part (a), which finds the closest pair of points in 3D space.

   (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.

To solve this, we can use the same approach as we do for finding the closest points on a flat plane (2D), but we have to add an extra step to consider the wrapping

Basically, we still sort the points and split them up to find pairs, but now we also have to check across the wrapped edges - the top wraps to the bottom and the left to the right.

Therefore, we can apply the same logical steps, and the process maintainus our overall efficiency.

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 )
```

The algorithm calculates the greatest common divisor (GCD) of two numbers x and y using a divide & conquer approach that reduces the problem size

1) If x equals y, then the GCD is x (or y), since a number is always divisible by itself.

2) If both x and y are even, then their GCD also must be even. The algorithm divides both by 2.

3) If one number is even and the other is odd, the algorithm divides only the even number by 2. This is because the GCD of an even and an odd cannot be an even.

4) If both numbers are odd and x (y) is greater than y (x), the algorithm subtracts y from x, divides the result by 2, since the result must be even.

The algorithm guarantees that in each recursive call, at least one of the numbers is reduced by half. The maximum number of halving operations needed is proportional to the logarithm of the larger number, the worst-case running time is $O(\log(\max(x, y)))$

4. Use recursion trees or unrolling to solve each of the following recurrences. Make sure to show your work, and do NOT use the master theorem.

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

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

Since we are dividing by 6 each time, the depth of A(n) is $\log_6(n)$
At each level of the tree, we have a constant constant amount of work, 1.
The total amount of work done is the sum of the work at each level.
Since each level contributes a constant amount of work and there are $\log_6(n)$ levels,
the total work is:

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

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

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

The number of layers in the recursion tree is $\log_6(n)$ with the root layer having work n.
The total work of layers is the sum of each layer with $\frac{1}{6}$ times the previous layer.

The total work for the recurrence $B(n) = B(n/6) + n$ can be expressed as:
$\sum_{k=0}^{\log_6(n)} \frac{n}{6^k} = n \times \frac{1-(\frac{1}{6})^{\log_6(n)+1}}{1-\frac{1}{6}} = \frac{6}{5}n - \frac{1}{5}$

The result indicates that the total work $B(n)$ is proportional to n, which is reflected by $\Theta(n)$

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

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

The total work of layers is the sum of each layer with $\frac{1}{6} + \frac{3}{5} = \frac{23}{30}$ times the previous layer with the root layer having work n.

The total work for the recurrence $C(n) = C(n/6) + C(3n/5) + n$ can be expressed as

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

Since $\frac{30}{7}$ is a constant, the complexity of $C(n)$ is linearly proportional to n.
Therefore, the complexity of $C(n)$ is $\Theta(n)$.

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

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

The total work of layers is the sum of each layer with $\frac{d-1}{d}$ times the previous layer with the root layer having work x.

The total work for the recurrence $D(x)$ is:

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

Therefore, the complexity of $D(x)$ is linearly proportional to x, denoted as $\Theta(x)$.

Coding Questions

5. **Line Intersections:**
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, your code needs to run in $O(n \log n)$ time.
Hint: How does this problem relate to 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.

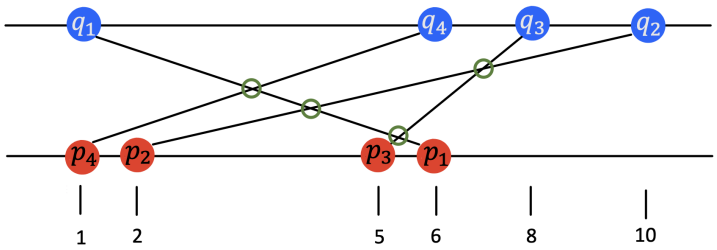

Figure 1: An example for the line intersection problem where the answer is 4

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 the results of some of the test cases may not fit in a 32-bit integer.

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