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: Mahir Khan Wisc id: mhkhan9

## 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.

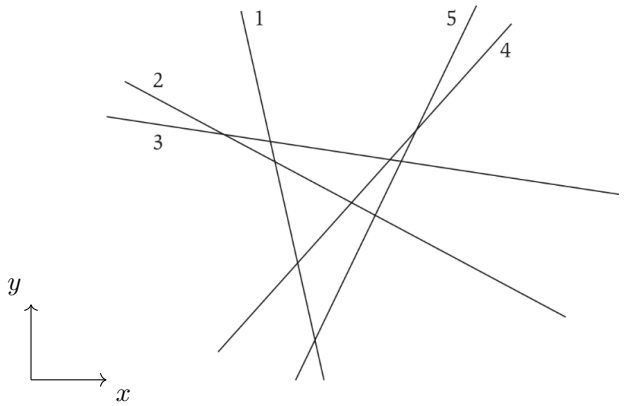



**Figure 5.10** An instance of hidden surface removal with five lines (labeled *1–5* in the figure). All the lines except for *2* are visible.

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

We know that a line is visible if it is uppermost for atleast one coordinate. Thus, for this algorithm, the Base Case is when there is only n=1 line given as input, and thus it is uppermost and visible.
For the rewsive case, split the list of lines into the front half and the back half, and rewse on them.
The rewsive calls will return a sorted list with the number of visible lines in the subarray, and then the 2 subarrays returned in the rewsive case will then be merged, and then each element in the subarrays being merged will be checked with one another to find the current merged subarrays' list of visible lines. Once a line has been checked for visibility once, it will not be rechecked again, thus the merge step is a linear time cost.

(b) Write the recurrence relation for your algorithm.

The algorithm is a modified version of a MergeSort, similar to the counting inversions algorithm.

↳ This is because there are 2 recursive calls on $\frac{n}{2}$ items, and a linear cost $(n)$ call onto the merge and count call.

↳ Thus $T(n) = 2T(n) + cn$, $T(1) = 1$

(c) Prove the correctness of your algorithm.

We will use strong induction to prove the correctness of our algorithm on a list of lines of size $n$.

Base Case: $n=1$, thus the algorithm returns the only line in the list, which is by default uppermost, thus the base case holds.

Inductive step: Assume that $n=k$, and that the recursive calls on the lists of size $\frac{k}{2}$ return the correct number of visible lines. Thus our algorithm will now merge these 2 sorted lists returned, and return the correct number of visible lines in the original list of size $n=k$

↳ Thus our algorithm holds as shown by strong 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:

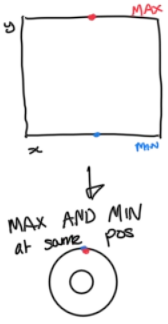
   (a) First consider the problem of searching for the closest pair of points in 3-dimensional space. Show how you could extend the single plane closest pairs algorithm to find closest pairs in 3D space. Your solution should still achieve $O(n \log n)$ run time.


1) In the 2D problem, we have a line $\ell$ that seperates Q and R from each other, which is how each rewsive call is divided. For the 3D problem, instead of a line, we can have a 2D plane P that acts as the "dividing line". Thus Q and R would be points on opposite sides of a point in the plane, i.e: the "highest" point in Q, or "lowest" point in R.
2) Since there are 3 planes, $Q_x, Q_y, Q_z$ (and R counterparts) are required, so that needs to be accounted for but should not incur a significant cost as pointers can be used.
3) Need to consider pairs of points within $\delta$ of the 2D dividing plane P, and put into a set S to see if $s_x, s_y, s_z \in S < \delta$, where $\delta = \min(d(q_x, q_y, q_z), d(r_x, r_y, r_z))$

   (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.

A sphere is a 3D object, meaning the idea of splitting 2 halves of the problem area with a 2D plane still holds. The only difference is that each split causes 2 hemispheres. The plane would also be circular, meaning that distances checked would be angular distance. The steps for dividing and conquer remain the same as the 3D problem above.

   (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.




Due to the wraparound behavior, having one dividing line seems inefficient and pointless. This is because it only acts a stopping border that is already present in a non-wraparound plane. Thus, instead of one dividing line, it seems like 4 dividing lines are required, 2 horizontal and 2 vertical. Thus, the partitioning is done by having one partition on the "outside" of 2 dividing lines, and the other partition on the "inside" of the dividing lines.
↳ Then, in each partition, the same algorithmic steps to find the closest pair will be done, with each rewrsion adding 4 more dividing lines. Points near wrapping edges will be placed in the appropriate halves due to wraparound behavior.

↳ The rest of the steps with checking points between partitions should be the same as the 2D algorithm using $\delta = \min(d(q_x, q_y), d(r_x, r_y))$

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):
Base Case: if x = y:
               return x
Case 1     else if x and y are both even:
               return 2*BINARYGCD(x/2,y/2)
Case 2     else if x is even:
               return BINARYGCD(x/2,y)
Case 3     else if y is even:
               return BINARYGCD(x,y/2)
Case 4     else if x > y:
               return BINARYGCD( (x-y)/2,y )
Case 5     else
               return BINARYGCD( x, (y-x)/2 )
```

**Correctness** Need to show that BinaryGCD returns correct output (the gcd) and it also terminates.

The invariant we need to show is that the GCD of a call $(x^*, y^*)$ is the same as the original call $(x, y)$.

**Base Case:** If $x$ and $y$ are equal, then returning $x$ is valid as the GCD, since $x^* = x$, and $x = y$.

**Recursive Calls:** Assume that base case not satisfied thus $x \neq y$.

**Case 1:** In this case, both $x$ and $y$ are even (i.e $= 2k$), thus halving them (i.e: $k$) will not alter the GCD's result, it only goes towards the base case, thus valid. Furthermore, the multiplication of 2 at the end end of the unrolling is because the algorithm will return an odd number, thus the $2x$ fixes that.

**Case 2/3:** If $x$ is even but $y$ isn't, it is halved to reduce problem size. vice versa for $y$ = even, $x \neq$ even

**Case 4/5:** subtracting lesser value + halving to reduce problem size to base case, and return valid answer.

↳ Thus, correctness + termination shown!

All cases preserve the GCD and make the problem size smaller, working towards the base case!

**Worst Case running time:** → only constant operations

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

$T(1) = 1$ → $T(\frac{n}{2^k}) + kn \rightarrow T(1) + \log n$

$= 1 + \log n = O(\log n)$

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 $n \geq 1$.

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

$A(n) = A(n/6) + 1$

$= A(A(n/36) + 1) + 1$

$= A(A(A(n/216) + 1) + 1) + 1$

$\vdots$

$= A(\frac{n}{6^k}) + k$

$1 = \frac{n}{6^k}$

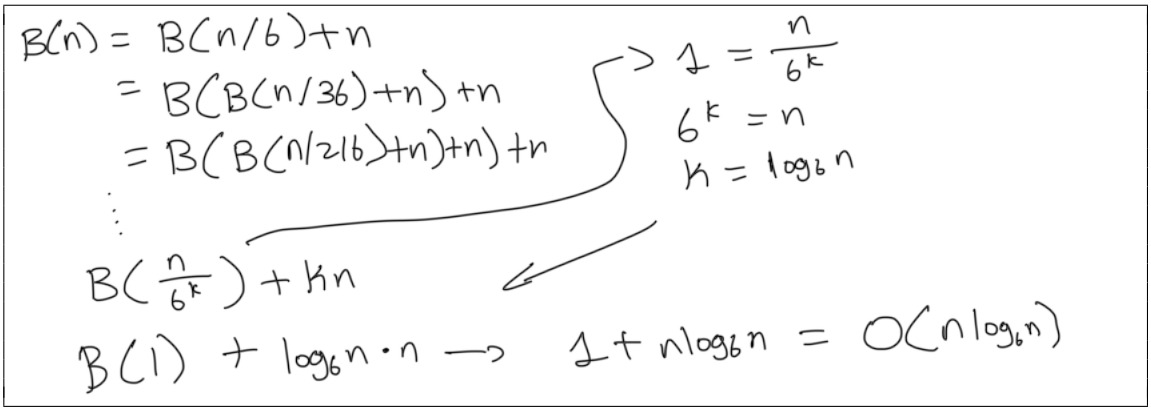
$6^k = n$

$k = \log_6(n)$

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

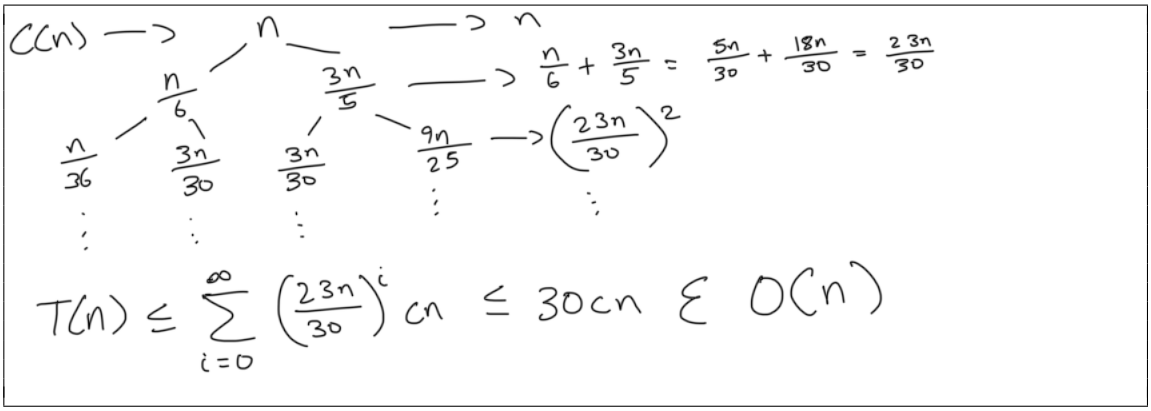
(b) Asymptotically solve the following recurrence for $B(n)$ for $n \geq 1$.

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




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

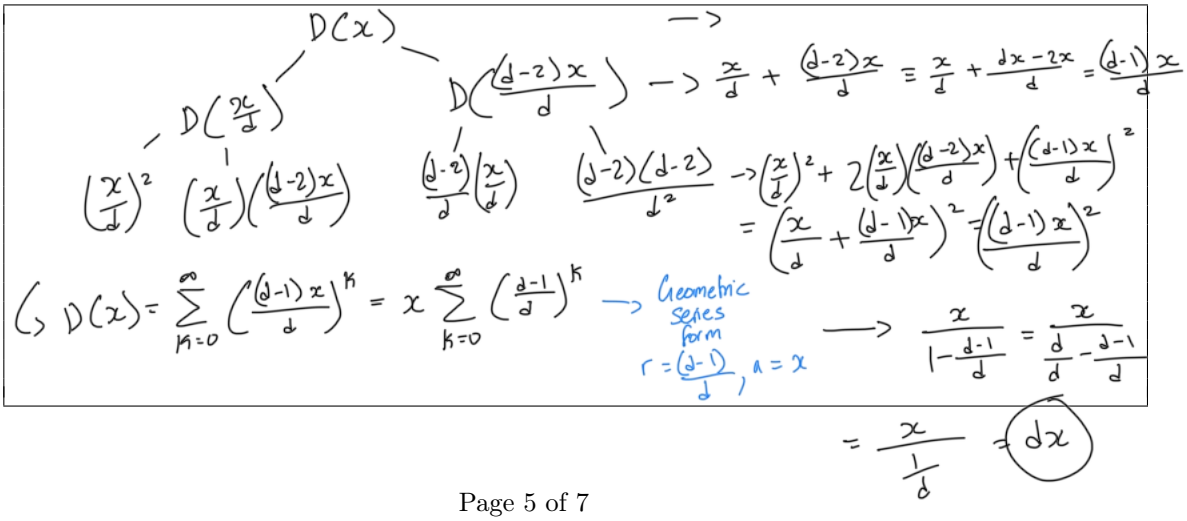
$$C(n) = C(n/6) + C(3n/5) + n \qquad \text{with base case} \qquad C(0) = 0$$




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

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

Assume $d = 3$



# 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.

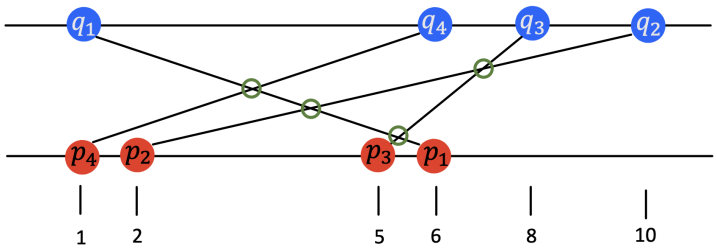



Figure 1: An example for the line intersection problem where the answer is 4

**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