

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: Matej Popovski _____

Wisc id: popovski / 9083541632 _____

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.

- (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: The following is an algorithm that takes n lines as input and in $O(n \log n)$ time returns all the ones that are visible:

1. If $n=1$, return the line.
2. Divide the n lines into two sets: the first set contains the first $n/2$ lines and the second set contains the remaining $n/2$ lines.
3. Recursively call the algorithm on the first set to obtain the visible lines in the first set.
4. Recursively call the algorithm on the second set to obtain the visible lines in the second set.
5. Merge the visible lines obtained from steps 3 and 4 by selecting the uppermost line at each x coordinate.
6. Return the merged set of visible lines.

- (b) Write the recurrence relation for your algorithm.

Solution: The recurrence relation for the algorithm is $T(n) = 2T(n/2) + O(n \log n)$. This is because we divide the problem into two subproblems of size $n/2$ each and then merge the results in $O(n \log n)$ time.

- (c) Prove the correctness of your algorithm.

Solution: To prove the correctness of the algorithm, we need to show that it correctly identifies the visible lines. We will use induction on the number of lines n . Base case: When $n = 1$, the algorithm returns the line which is trivially the only visible line. Induction hypothesis: Assume that the algorithm correctly identifies the visible lines for $n/2$ lines. Induction step: Consider n lines. By dividing the lines into two sets and recursively applying the algorithm to each set, we obtain two sets of visible lines. Let $L1$ be the uppermost line in the left set and $L2$ be the uppermost line in the right set. If $L1$ and $L2$ intersect, then the algorithm correctly identifies the visible lines by selecting the uppermost line at each x coordinate. If $L1$ and $L2$ do not intersect, then there must be a line $L3$ that is between $L1$ and $L2$. Since $L3$ is not in either set of lines, it must be completely hidden behind either $L1$ or $L2$. Without loss of generality, assume that $L3$ is hidden behind $L1$. Then $L2$ must also be hidden behind $L1$, since $L1$ is uppermost at all x coordinates to the left of the intersection point of $L1$ and $L2$. Therefore, the algorithm correctly identifies the visible lines by selecting $L1$ and the visible lines in the left set. By induction, we have shown that the algorithm correctly identifies the visible lines for all n lines.

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: As in 2D we perform following steps: divide: split point set (in half), 2conquer: find closest pair in each partition. Combine: Merge the solutions. Difference will be that in 3D we have to sort according the 3 axes: let P_x be points sorted by x-coordinate, P_y points sorted by y-coordinate and P_z points sorted by z-coordinate (complexity $O(n \log n)$). Determine plane in point x_0 median point of P_x coordinateness (parallel to yz plain) that divide space in 2 sets Q and R (left and right according x_0). Recursively find the closest pair of points in the left and right sets. We can find the set of distances using a modified search that considers the points sorted by y and z -coordinates (complexity $O(n)$). Let the minimal distance from the points in sets Q and R be $\min(d(q_0^*, q_1^*), d(r_0^*, r_1^*))$, where d is calculated by Euclidian distance formula for 3D. To compare the pairs of points across the separating plane we have to consider the two parallel planes on the distance of plane in x_0 . Similar to the 2D case now at most 63 points in the cube $4 \times 4 \times 4$ have to be checked for at most n points in this corridor if their distance is less than (complexity $O(n)$). So overall complexity is $O(n \log n)$.

- (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.

Solution: To adapt the divide and conquer algorithm for finding the closest pair of points on a sphere, we can follow a similar approach as the one used for finding the closest pair of points in the 2D. We can split the points to upper and lower hemisphere. However, instead of calculating the Euclidean distance between points, we need to use the distance across the surface of the sphere. As an example here is Haversine distance formula

$$d = 2r * \arcsin(\sqrt{\sin^2((\text{latB} - \text{latA})/2) + \cos(\text{latA}) * \cos(\text{latB}) * \sin^2((\text{lonB} - \text{lonA})/2)})$$
 where r is the radius of the sphere. If us smallest distance in either one, we have to check distance between the points from the two hemisphere. The 4×4 patch similar to one in 2D solution contain 15 possible points for each n point in the range distances of Equator to be checked for only 15 points, giving complexity $O(n)$.

- (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: To find the closest pair of points on the surface of a torus, we can use a similar approach to the 3D case, but we need to take into account the toroidal geometry. Specifically, we'll need to use a periodic distance function to compute the distance between two points on the torus. To partition the torus into two regions, we'll use a plane that's perpendicular to one of the coordinate axes, as in the 3D case. However, we need to be careful when checking if there's a closer pair of points that spans both regions. We'll need to consider pairs of points that are on either side of the dividing plane and that may be "wrapped around" the torus. This can be done by checking pairs of points that are within a certain distance of the dividing plane, where the distance is less than the minimum distance between any two points in either region. The overall runtime is still $O(n \log n)$.

3. *Erickson, Jeff. Algorithms (p. 58, q. 25 d and e)* Prove that the following algorithm computes $\text{gcd}(x, y)$ 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: Correctness. Consider k is greater common divisor of x and y

1. $x=y$ then $k=x=y$ is GDC.
2. If x, y both are even then k is even and $k/2$ is GDC of $x/2, y/2$. Let A is set of CD (x,y) , B is set of CD $(x/2,y/2)$. Then $A=2B$.
3. x is even, y is odd, then k is odd and it is valid that $\text{CD}(x,y)=\text{CD}(x/2,y)$, we have $\text{GCD}(x,y)=\text{GCD}(x/2,y)$. Vice versa if x is odd and y is even, then $\text{GCD}(x,y) = \text{GCD}(x,y/2)$.
4. If x is odd and y is odd. Let $x > y$, then for d divisor $x=c_1 d, y=c_2 d$. We have $(x-y)=(c_1-c_2)d$. Since d is odd we have $(x-y)/2$ is divisible by d . Then for k , $\text{CD}(x,y)=\text{CD}((x-y)/2,y)$ (the same GCD). Vice versa $x < y$ $\text{CD}(x,y)=\text{CD}(x,(y-x)/2)$.

In the worst case, the algorithm will continue to make recursive calls until the inputs are both odd and $x \leq y$. At this point, the larger of the two inputs will be at most $n/(2 \text{ to the power of } k)$, where k is the number of recursive calls made. Since we want to find the worst-case time complexity, we can assume that the algorithm will make the maximum number of recursive calls possible.

In conclusion the overall time complexity is $O(\log n)$

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 $n \geq 1$.

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

Solution: $T(n)=T(n/6)+1$ $A(1)=1$

$$T(n)=T(n/6)+1=T(n/6^2)+1+1=\dots=T(n/6^k)+k \Rightarrow T(n/6^k)=T(1) \Rightarrow n/6^k=1 \Rightarrow k=\log_6 n$$

$$T(n) \leq 1 + \log_6 n \Rightarrow T(n)=O(\log n)$$

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

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

Solution: $T(n) = T(n/6) + n$ $A(1) = 1$

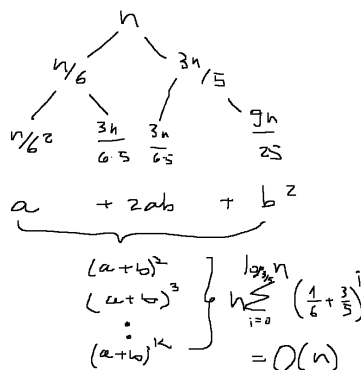
$$T(n) = T(n/6) + n = T(n/6^2) + n + n = \dots = T(n/6^k) + kn \Rightarrow T(n/6^k) = T(1) \Rightarrow n/6^k = 1$$

$$k = \log_6 n$$

$$T(n) \leq 1 + n \log_6 n \Rightarrow T(n) = O(n \log n)$$

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

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

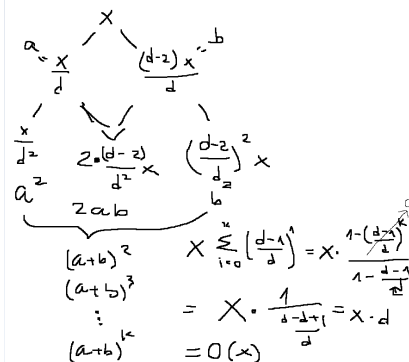


$$\left. \begin{array}{l} (a+b)^2 \\ (a+b)^3 \\ \vdots \\ (a+b)^k \end{array} \right\} \log_{3/5} n = O(n)$$

Solution:

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

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



$$\left. \begin{array}{l} (a+b)^2 \\ (a+b)^3 \\ \vdots \\ (a+b)^k \end{array} \right\} \log_{d/3} x = O(x)$$

Solution:

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 $\{p_1, p_2, \dots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \dots, q_n\}$ 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.

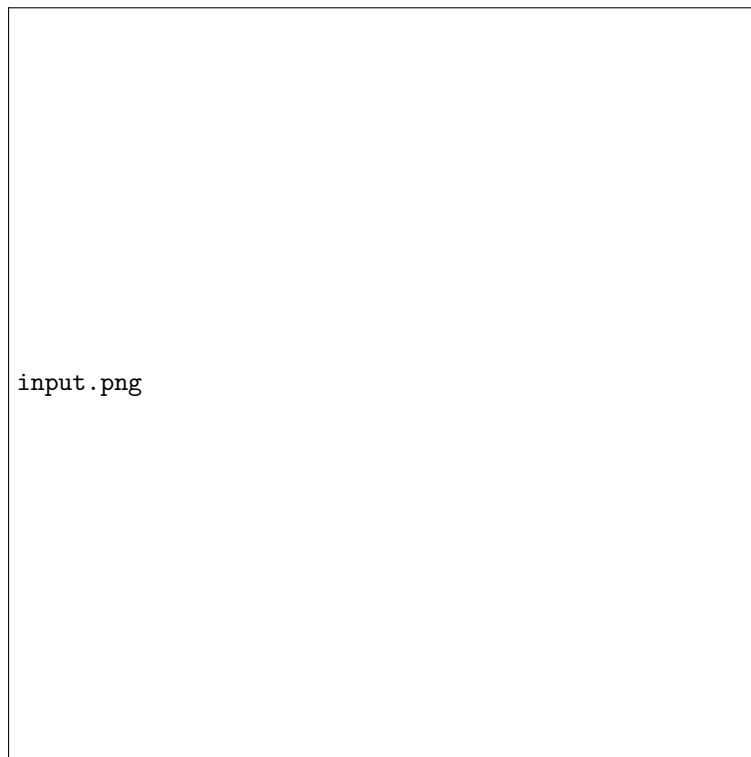


Figure 1: An example for the line intersection problem where the answer is 4

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