CS 577 **Assignment 3 – Divide and Conquer** Summer 2023

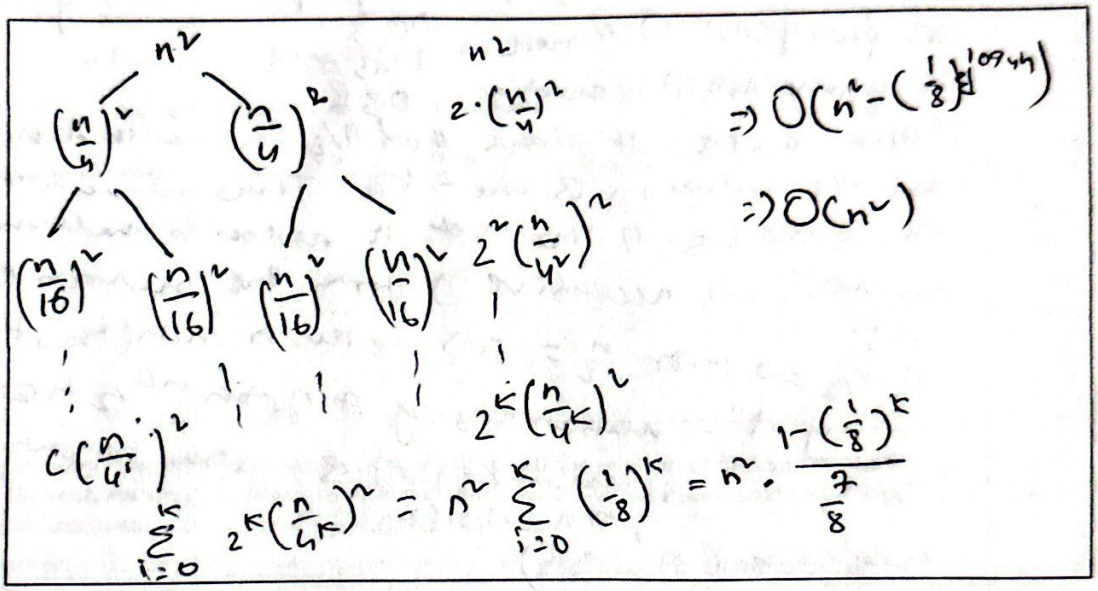
Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: GuruSharan Wisc id: Kunusoth

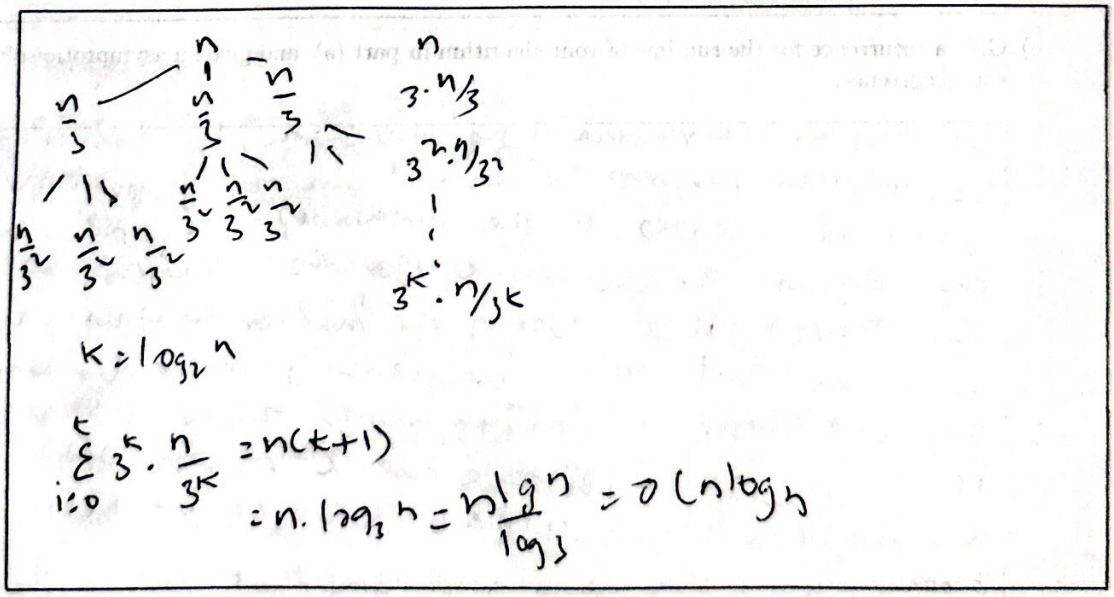
Divide and Conquer

1. *Erickson, Jeff. Algorithms (p.49, q. 6).* Use recursion trees to solve each of the following recurrences.

   (a) $C(n) = 2C(n/4) + n^2$; $C(1) = 1$.



   (b) $E(n) = 3E(n/3) + n$; $E(1) = 1$.



2. *Kleinberg, Jon. Algorithm Design (p. 246, q. 1).* You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values—so there are $2n$ values total—and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the nth smallest value.

However, the only way you can access these values is through queries to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the kth smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

(a) Give an algorithm that finds the median value using at most $O(\log n)$ queries.

median (array A, array B) // array representations of both DB's
k_A = query(A, n/2) // median of DB A
k_B = query(A, n/2) // median of DB B
// Suppose $k_B < k_A$. It shows that n/2 values in A are $< k_A$ and n/2 values in B are $> k_A$. Thus median m is $k_B \le m \le k_A$. // The set to research median is reduced, we recursively form the median this way.
if n/2 == 1 or n/2 == 0 return min(k_A, k_B)
array A' = subarray of A from n/2 to end
array B' = subarray of B from beginning to n/2.
median (A', B').

(b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence.

Let T(n) be the runtime of the algorithm. In each step, the algorithm performs a constant amount of work and makes two queries to the databases. Therefore, we can express the recurrence for the runtime as: $T(n) = T(n/2) + O(1)$. Using the master theorem, we can see that this recurrence falls under case 1, where $a=1$, $b=2$ and $f(n) = O(1)$. The master theorem states that if $f(n) = O(n^c)$. In this case $c=0$, so $O(1)$ is constant). and $\log_b(a) = \log_2(1) = 0$. Since $c < \log_b(a)$. We apply master theorems and find that $T(1) = \Theta(1^0) = \Theta(1)$.

(c) Prove correctness of your algorithm in part (a).

The algorithm correctly identifies the median by performing a modified binary search. It queries both databases, compares the values obtained, and updates the ranges accordingly. By systematically narrowing down the range, it eventually finds the median value. Therefore, the algorithm is correct in determining the median among 2n values.

3. *Kleinberg, Jon. Algorithm Design (p. 246, q. 2).* Recall the problem of finding the number of inversions. As in the text, we are given a sequence of n numbers $a_1, \ldots, a_n$, which we assume are all distinct, and we define an inversion to be a pair $i < j$ such that $a_i > a_j$.

We motivated the problem of counting inversions as a good measure of how different two orderings are. However, this measure is very sensitive. Let's call a pair a *significant inversion* if $i < j$ and $a_i > 2a_j$.

(a) Give an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings.

merge count:
Input = 2 ordered list A and B
output : merged list, count of significat inversions
Initialize S, S' to empty list.
C = 0, A' = A, B' = B
While either A or B' or B' is not empty, do
pop and append min{front of A, front of B'} to S
pop and append min{2×front of B', front of B'} to S'.
if appended item from B'
to S',
C = C + (A')
end

(b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence.

Count Sort.
Input = List A.
Output : Sorted array and number of inversions.
if (A) = 1 return (A, 0)
(A1, C1) = Count sort (first half of A)
(A2, C2) = Count sort (Back. half of A)
(A, C) = merge sort (A1, A2)
return (A, C1+C2+C).

2 3 7 8 | 1 4 5 6.
2 1 3 7 8

(c) Prove correctness of your algorithm in part (a).

The algorithm uses the divide & conquer approach of merge sort to split the array into smaller subarrays until the base case is reached. (ie, when the subarray has length 1 or 0). At each level of recursion, the algorithm counts the number of significat inversions in the left and right halves and merges the two sorted halves while counting the number of significat inversions.
During the merge step, the algorithm compares elents from the left and right halves, it adds the count of significat inversions equal to the number of elements to the right of the current elent. are smaller, and if current elent is greater than twice an elent in the right half, it is greater than all the remaining elents in the right half. By recersively applying this merge and count process, the algorithm correctly counts the number of significat inversions between two orderings.

4. *Kleinberg, Jon. Algorithm Design (p. 246, q. 3).* You're consulting for a bank that's concerned about fraud detection. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud.

   It's difficult to read the account number off a bank card directly, but the bank has an "equivalence tester" that takes two bank cards and determines whether they correspond to the same account.

   Their question is the following: among the collection of n cards, is there a set of more than $\frac{n}{2}$ of them that all correspond to the same account? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester.

   (a) Give an algorithm to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

merge sort:
input = 2 list A B with
majority appearance.
n_A, n_B (type of n_B search
longer than size(A)/2)
out put merged list C, majority
of this list M_C.
1. merge A, B into C.
2. if $M_B \neq$ null
compare M_B with every cards in C
count appearance L_B
else L_A 20.
3. Same for M_B, get L_B

   (b) Give a recurrence for the runtime of your algorithm in part (a), and give an asymptotic solution to this recurrence.

If $L_A > \frac{1}{2}$ size (C) returns (C, M_A)
else if $L_B > \frac{1}{2}$ size (C) return (C, M_B)
else return (C, null).

$\Theta(n(\log n))$

(c) Prove correctness of your algorithm in part (a).

count majority:
Input: list A.
Output: majority of B.
If |A| = 1, return (A, a)
where a is only item in A
(A1, a) = count majority (first half of B)
(A2, b) = count majority (Back half of B)
(A, B) = merge and count (A, A1, a, b)
return A.

When algorithm divides the collection into two halves, each containing n cards, it recursively applies the algorithm to each half. By the induction hypothesis, this step correctly determines whether there is a set of size n/2 corresponding to the same answer in each half. If both halves indicate that there is such a set, the algorithm correctly returns "yes" as the answer. If both halves indicate that there is no such set, the algo correctly returns "No" as the answer.
If one half indicates that there is a set and the other half indicates that there is no set, the algo proceeds to compare one card from each half. If these two cards correspond to the same account, the algorithm correctly returns "yes" as the answer. Otherwise, it correctly returns "No" as the answer.

5. *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.

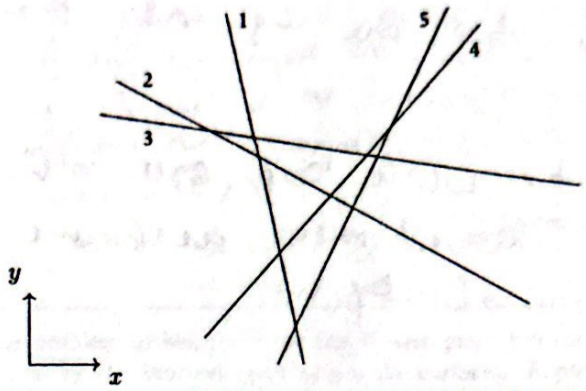


Figure 5.10 An instance of hidden surface removal with five lines (labeled *1-5* in the figure). All the lines except for *2* are visible.

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

Input: n lines
Output: solution
Sort n line based on slope and add them one by one.
We have $L_1, L_2 \dots L_n$. Let $S = \{L_1, L_2\}$.
Let T_1, T_2 denote the most recent added lines in S when adding a new L he

for $i = 3 \dots n$
  compute intersect of L_1, L_2. $A = (x_A, y_B)$
  consider: $L_1 = ax + by + c = 0$.
  while $ax_A + by_3 + c < 0$ (y_A smaller than the y point on L_i with same x_a value)
  remove T_1 from S. get new T_1, T_2. get new $A(x_A, y_B)$
  end
  add L_i to S.
end

(b) Write the recurrence relation for your algorithm.

Sorting requires $O(n \log n)$.
If we don't enter while loop, which means we don not remove lines, we just add lines one by one, then this is $O(n)$.
If we enter while loop. Still each line can only be removed once, we have $O(n)$.
Total is $O(n \log n)$

(c) Prove the correctness of your algorithm.

since there are no 3 lines ever meet at the same point, we sort lines by slope and add them one by ~~one~~ one.
Sorting is ~~$O(n \log n)$~~ $O(n \log n)$.

6. 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 got P_x, P_y, P_z as sorted x, y, z values. O(log n) find a hyperplane $x > k$ divides space into Q (left half of P_x) & (right half of P_x) then seperate corresponding y, z values to arranged points in Q, R (O(n)) and do this recursively. we have $\min\{d(q_1^*, q_2^*), d(r_1^*, r_2^*)\}$
then the total complexity is ~~O(n log n)~~ $\cdot O(n \log n)$

   (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.

we use the same algorithm, we consider points on subsphere as a subset of larger 3D space. we modify our distance metric $d(p_1, p_2)$ to distance on surface. the we use the same algorithm.

   (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.

we treat surface on sphere as a plane. Then we implement regular closest-pair algorithm on this single plane. Then, for checking the points in S (subset of points within δ) we also add points within δ of edge of plane into set S. We "padding" the ~~#~~ edge use ~~#~~ the opposite of points. We copy the points at lower S-edge of plane and add to upper of the plane. Then we not only check pairs cross divider but check pairs across edge. Then the following algo is the same

7. *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 )
```

• Then we show the correctness, since the algo return the correct result, and terminate in $\lceil \log x \rceil + 1 + \log(|x-y|+1)$. The algorithm is correct.

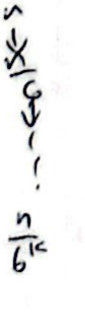
• worst-running time.
The recursion will stop when $x = y$, in worst-case, it stop at $x = y = 1$.
The running time is $O(\log x + \log y)$

1) first we show invariat. Consider k is the greatest common divisor of x and y. ① $x > y$ then $k \geq x > y$. 2) x, y both even then k is even and $k/2$ is GCD of $x/2, y/2$. {A is set of CD of (x, y). B is set of CD of $(\frac{x}{2}, \frac{y}{2})$ then $A = 2B$}. 3) x is even, y is odd. Then k is odd then any d in x, y is divisible by d. $x/2, y$ is divisible by d. vice versa, then $CD(x,y) = CD(\frac{x}{2}, y)$. we have GCD $(x \cdot y) = GCD(\frac{x}{2}, y)$. 4) x is odd, y is even, we have same metric - 3) x is odd. y is odd. $x > y$ we let $x = c_1 d$, $y = c_2 d$. we have $(x-y) = (c_1 \cdot c_2) d$. Since d is odd we have $\frac{x-y}{2}$ is divisible by d. vice versa then $CD(x,y) = (\frac{x-y}{2}) \cdot y$, then they have same GCD. 6) x, y are odd $x < y$. have same result.

8. Use recursion trees or unrolling to solve each of the following recurrences.

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

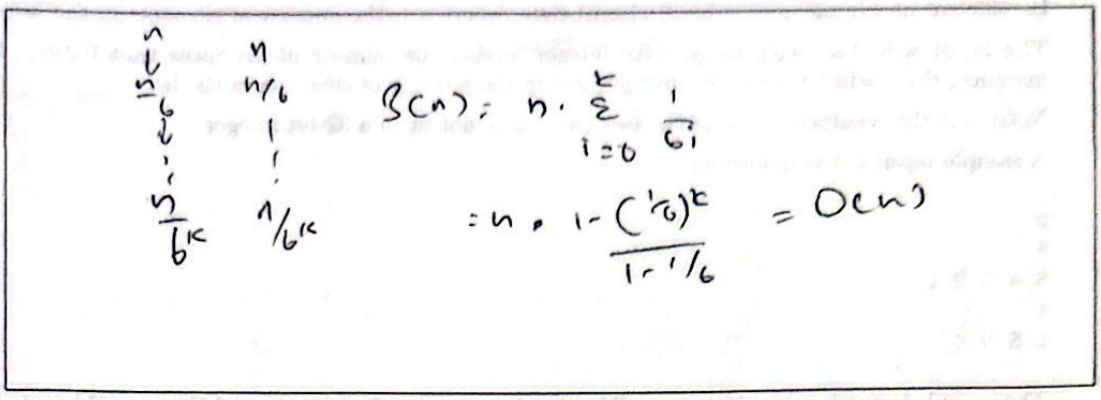
$$A(n) = A(n/6) + 1 \qquad \text{with base case} \qquad A(1) = 1$$



$$A(n) \leq \sum_{i=0}^{k} 1 = k = \log_6 n = O(\log_2 n)$$

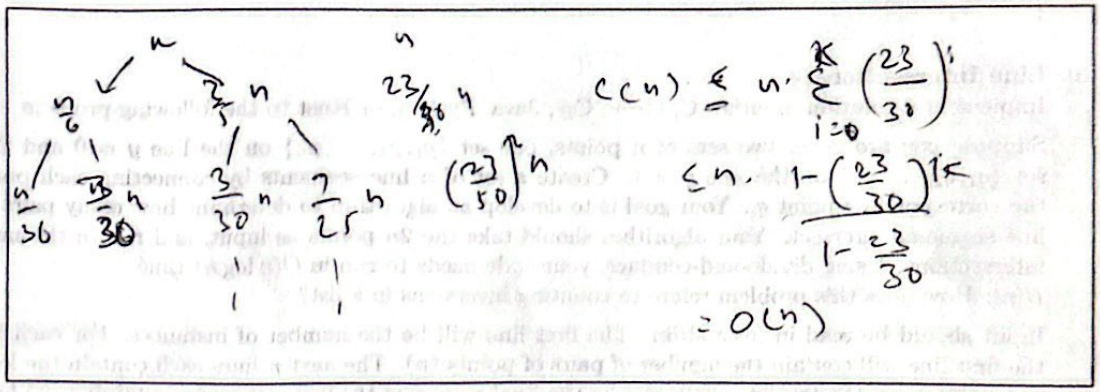
(b) Asymptotically solve the following recurrence for $B(n)$ for $n \geq 1$.

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



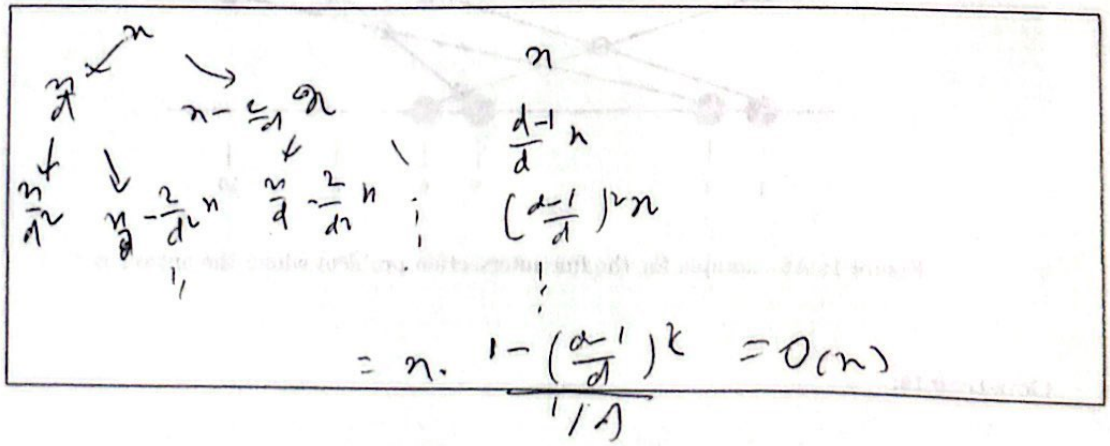
(c) Asymptotically solve the following recurrence for $C(n)$ for $n \geq 0$.

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



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

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



Coding Questions

9. **Inversion Counting:**
Implement the optimal algorithm for inversion counting in either C, C++, C#, Java, Python, or Rust. Be efficient and implement it in $O(n \log n)$ time, where n is the number of elements in the list.

The input will start with an positive integer, giving the number of instances that follow. For each instance, there will be a positive integer, giving the number of elements in the list.

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

A sample input is the following:

```
2
5
5 4 3 2 1
4
1 5 9 8
```

The sample input has two instances. The first instance has 5 elements and the second has 4. For each instance, your program should output the number of inversions on a separate line. Each output line should be terminated by a newline. The correct output to the sample input would be:

```
10
1
```

10. **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.

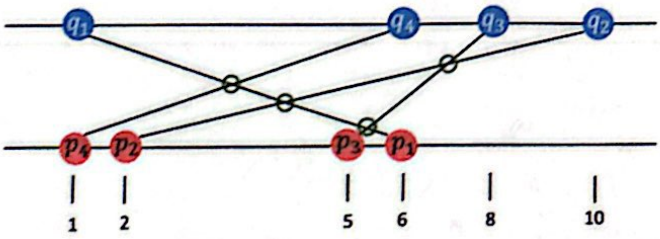


Figure 1: An example for the line intersection problem where the answer is 4

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