M1961 Lighthouse

Background

Problem idea by Alex Tung

Prepared by Percy Wong

O(N^2) Solution

O(N^2) Solution

It is known that the lighthouse is the only point which `detect` is always `false`
Otherwise, among the 8 detections, at least one should be `true`

Check all (2N+1) * (2N+1) points!
Worst case cost = 8 * (2N+1) * (2N+1) = 323208 (for subtask 1)

Score ≃ 12/30 + 0/70

O(N^2) Solution

Some possible (small, or not huge at least) improvements:

- random (to avoid worst case, as the grader is not adaptive)
- given that a point lights up at time (8k + t), it's so likely that the points next to it will also light up at time (8k + t) // k means any non-negative integers
  // t means integers between [0, 8)

  therefore we can query some other points which are likely to light up at time (8k + t + 1) right after detected light up at time (8k + t)

O(N) Solution

O(N) Solution

For all points lying on the x-axis, count how many times it will light up in 8 consecutive seconds

If there exists a point that lights up 0 times, that's the lighthouse
Otherwise:

- it implies that the lighthouse is not on the x-axis
- there must be two or three points lying on the x-axis which each lights up 2 times
- by solving some equations, we can deduce to two possibilities of the lighthouse's position // e.g. (-1, 0) and (3, 0) -> possibilities: (1, 2), (-1, 2)
- for each possibility, just detect 8 times to see if it lights up or not

O(N) Solution

Worst case cost = 8 * (2N + 1) + 8 + 8 = 1624 (subtask 1) and 160024 (subtask 2)

Score ≃ 20/30 + 28/70

O(logN) Solutions

From now on, times shown are modulo(%)-ed by 8

O(logN) Solutions

The solution is divided into two main parts:

1. find the value of t0, i.e. the starting time of the investigation
   so we can tell which direction is the lighthouse from a certain point later on
2. using binary-search-like algorithms to find the value of (x, y)

Each part can be solved in various ways with different performance

O(logN) Solutions - Step 1 (C = 32)

For each of the four corners, detect 8 times

usually, there is a pair of adjacent corners which one lights up at time t and another at time (t + 1)%8

- the edge which this pair of adjacent corners locate, can only light up on time t and (t + 1) % 8
- t0 can be calculated

```
// e.g. if NE and SE light up at t = 3, 4
// the east edge should light up at t = 1, 2
// meaning t0 = (1 - 3) = -2 = 6
```

O(logN) Solutions - Step 1 (C = 32)

For each of the four corners, detect 8 times

usually, there is a pair of adjacent corners which one lights up at time t and another at time (t + 1)%8

in some corner case, the above does not hold:

- the lighthouse is located at (0, 0), in which all 4 pairs meet the condition
- the lighthouse is located on some edge, in which 2 pairs meet the condition

Handle these cases!

O(logN) Solutions - Step 1 (C = 16)

For each of two opposite corners, detect 8 times

By precomputing all possible cases (simulating with small N), it can be shown that t0 can be uniquely deduced by that 16-bit data

11 x 8 cases in total

O(logN) Solutions - Step 2 (C = 224)

Take the edge which the aforementioned pair of corners locate,
WLOG, let's assume it's the edge on east

e.g. if we know that NE corner lights up at t = 2, SE corner lights up at t = 3

then the light up time of the points on the edge is in the pattern of:
2, 2, 2, 2, ..., 2, 2, (both), 3, 3, 3, ..., 3, 3, 3, 3

O(logN) Solutions - Step 2 (C = 224)

2, 2, 2, 2, ..., 2, 2, (both), 3, 3, 3, ..., 3, 3, 3, 3

the (both) point means the lighthouse is located at some point of the same y coordinate, binary search to find the value of y

if (N, mid_y) only lights at t = 2, meaning mid_y < y <= hi_y
if (N, mid_y) only lights at t = 3, meaning lo_y <= y < mid_y
if (N, mid_y) lights at both t = 2, 3, meaning y = mid_y

Knowing the value of y, we can do another binary search on x, this time the sequence is 67, 67, 67, 67, ..., 67, 67, (nothing), 23, 23, ..., 23, 23, 23, 23

O(logN) Solutions - Step 2 (C = 136)

2, 2, 2, 2, ..., 2, 2, (both), 3, 3, 3, ..., 3, 3, 3, 3

instead of using 8 queries to reduce the range by half,
we can use on average 4 queries to reduce the range by half

while (lo_y < hi_y)
  mid_y = (lo_y + hi_y) / 2
  detect (N, mid_y) at t = 2 -> reduce the range by half
  mid_y = (lo_y + hi_y) / 2
  detect (N, mid_y) at t = 3 -> reduce the range by half again!

O(logN) Solutions - Step 2 (C = 120)

After finding the t0, we can do a “binary-search”-like algorithm on 2-D plane

solve(lo_x, hi_x, lo_y, hi_y) =
  detect(mid_x, mid_y) for 8 times
  by using the detection and value of t0, we know which direction the lighthouse is
  then reduce the area to ¼ // if t0 = 0, detect shows (0, 0) lights at t = 3
  // implies that the lighthouse is at top-left

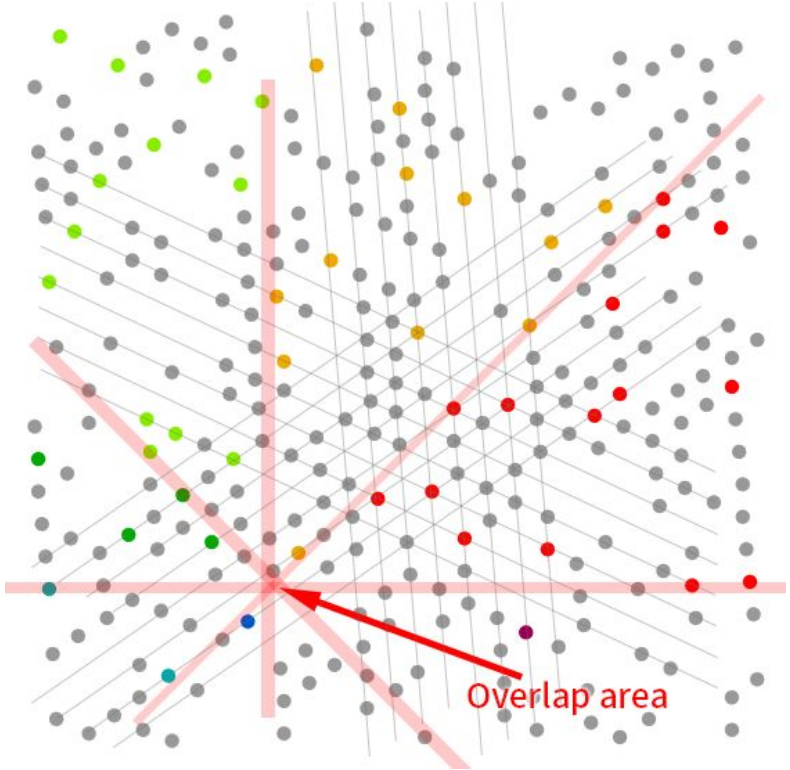
this is sufficient to get 100pts with Step 1 (C = 16)

Other Solutions

Other Solutions

Query points randomly in a rotated triangular lattice with side length ~= 0.04n.

For n = 10000, perform this once more with smaller side length and smaller area.

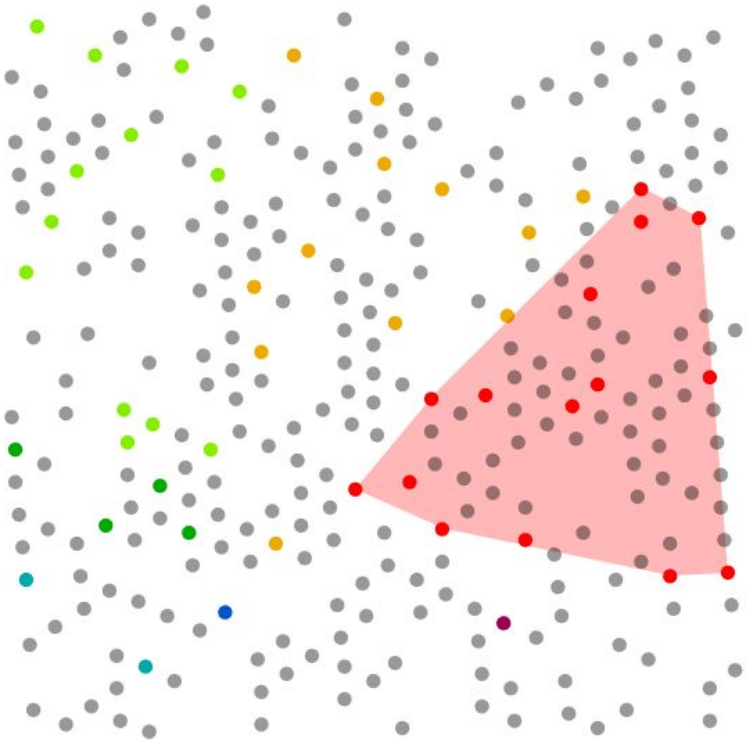


Other Solutions

Make fewer useless queries:

All points bounded by the convex hull have the same color.

Online update the convex hull.



M1962 Planar Game

Alex Poon

2 July 2019

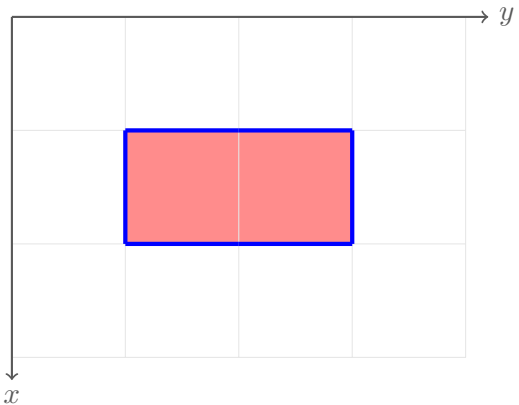
Statement

- Given a $R \times C$ grid and M of the edges of the small squares are toggled on
- In each of the $Q - 1$ queries, one of the edges of the small squares are toggled
- Output the winner of the following two-player game
  before any query and right after each query
  - The players takes turns to toggled off an edges of the small squares such that the number of bounded regions decrease
  - The player who cannot make a move loses

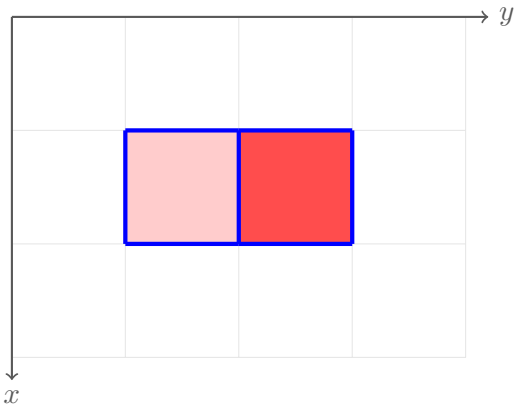
Statement

- Given a $R \times C$ grid and M of the edges of the small squares are toggled on
- In each of the $Q - 1$ queries, one of the edges of the small squares are toggled
- Output the winner of the following two-player game
  before any query and right after each query
  - The players takes turns to toggled off an edges of the small squares such that the number of bounded regions decrease
  - The player who cannot make a move loses
- It is easy to see that the winner is determined by the parity of the number of bounded regions before the game starts

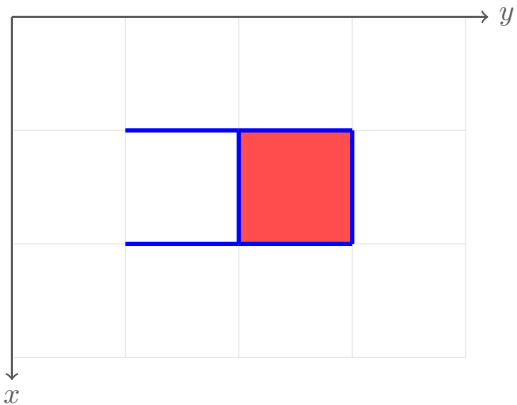
Sample



Sample



Sample



Subtask 1

$1 \leq R \leq 2, 1 \leq C \leq 3$
$1 \leq Q \leq 1000$

- Only two small squares are relevant
- Simply check for the 5 cases
  - No bounded regions
  - The left square is the only bounded region
  - The right square is the only bounded region
  - Both squares are bounded regions (and they are separated)
  - Both squares are in the same bounded regions
- Time complexity: $O(Q)$

Subtask 5

$1 \leq R \leq 2, 1 \leq C \leq 100000$
$1 \leq M, Q \leq 100000$

- There are only a row of small squares
- Use data structure to maintain whether a range of small squares have bounds on its top and its bottom
- Update the number of bounded regions only by the affected ranges
- Time complexity: O$((M + Q) \log C)$

Subtask 4

$1 \leq R, C \leq 700$
$1 \leq M, Q \leq 100000$
All active edges are connected

- Number of bounded regions $=$ number of edges $-$ number of vertices $+1$
- Time complexity: O$(M + Q)$

Subtask 2

$1 \leq R, C \leq 700$
$1 \leq Q \leq 1000$

- Number of bounded regions
  $=$ number of edges $-$ number of vertices $+$
  number of connected components
- Rebuild the graph and use disjoint set union to compute the number of connected components in each query
- Time complexity: O$(Q(M+Q)\alpha(M+Q))$

Subtask 3

$1 \leq R, C \leq 700$
$1 \leq M, Q \leq 100000$
No edges are toggled off during each query

- There is no need to rebuild the graph for each query
- Time complexity: O$((M+Q)\alpha(M+Q))$

Subtask 6

$1 \leq R, C \leq 100000$
$1 \leq M, Q \leq 100000$

- Disjoint set union with undo operation
- Build a segment tree on time axis, each element is a list of active edges
- Process queries by traversing the segment tree (DFS)
- Store the changes made by inserting edges
- Undo the changes right before leaving that node
- Don't use path compression but union by size
  since amortized time complexity doesn't work here
- Time complexity: $O((M+Q) \log Q \log(M+Q))$

Subtask 1-4

$1 \leq R, C \leq 700$
$1 \leq M, Q \leq 100000$

- If you are not familiar with Euler Characteristic ($V - E + F = 2$) and cannot deduce the previous formula for number of bounded regions
- Consider every small square as vertex with a vertex representing the unbounded area to get a graph with $O(RC)$ vertices and $O(RC)$ edges
- The number of bounded regions is the number of connected components in the new graph -1
- Time complexity: $O((M + Q + RC) \log Q \log(RC))$

M1963 Airship

Anson Ho

2 July 2019

Background

https://youtu.be/2T-hFFsDaFQ?t=86

Statement

- Given N integers $l_1, l_2, \ldots, l_N$
- For each of the Q queries
  - Given two integers s_j, k_j
  - Output the smallest and the second smallest $|i - k_j| \times s_j + l_i$ for $i = 1, 2, \ldots, N$

Subtask 1

$1 \leq N \leq 3000$
$1 \leq Q \leq 3000$

- For each scenario, check the cost of each airship
- Then output the smallest two
- Time complexity: O(NQ)

Subtask 2

$1 \leq N \leq 3000$
$1 \leq Q \leq 500000$

- Poor implementation of full solution
- Time complexity: $O(N^2 + Q \log N)$ or $O((N^2 + Q) \log N)$

Subtask 3

$1 \leq N \leq 500000$
$1 \leq Q \leq 500000$
$s_1 = s_2 = \cdots = s_Q = s$

- Consider the airships on the left and the right of Carl separately
- Then the cost $|i - k_j| \times s_j + l_i$
  becomes either $(si + l_i) - sk_j$ or $(-si + l_i) + sk_j$
- Consider the airships and the queries from one side to another
  (then repeat in another direction)
- Maintain the smallest two values of $si + l_i$ (and $-si + l_i$ respectively)

Subtask 3

$1 \leq N \leq 500000$
$1 \leq Q \leq 500000$
$s_1 = s_2 = \cdots = s_Q = s$

- Then the answer of each query is the smallest two among the four computed values
- Time complexity: $O(N + Q)$

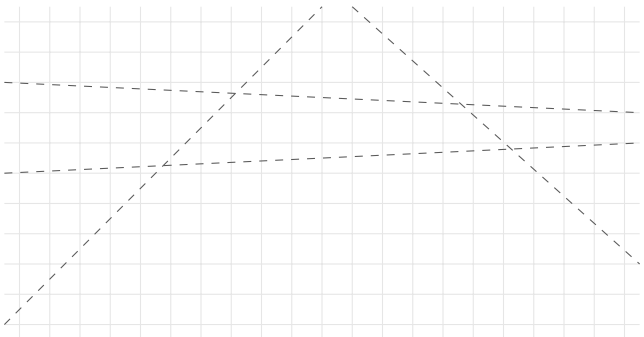
Subtask 4

$1 \leq N \leq 500000$
$1 \leq Q \leq 500000$

- Consider $si + l_i$ (and $-si + l_i$ respectively) as linear function with variable s
- Use the data structure which is usually known as convex hull trick
- It supports online inserting linear functions and querying minimum value given a value of s in $O(\log N)$ per operation
- At this point, the minimum value in each scenario can be computed

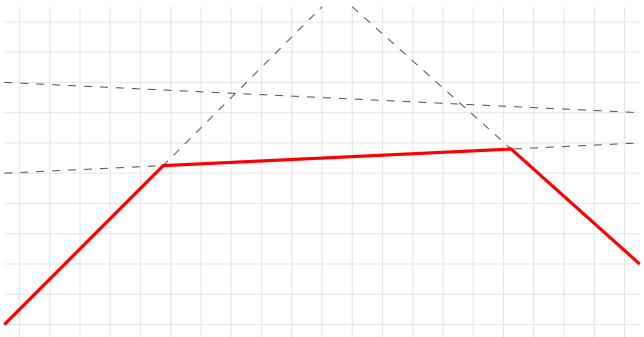
Subtask 4

$1 \leq N \leq 500000$
$1 \leq Q \leq 500000$



Subtask 4

$1 \leq N \leq 500000$
$1 \leq Q \leq 500000$



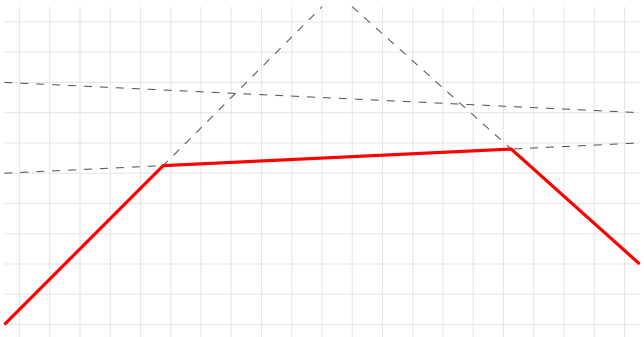
Subtask 4

$1 \leq N \leq 500000$
$1 \leq Q \leq 500000$

- Observation: the second smallest value is never obtained from the linear functions in the convex hull except the two neighbours of the linear function that gives the smallest value

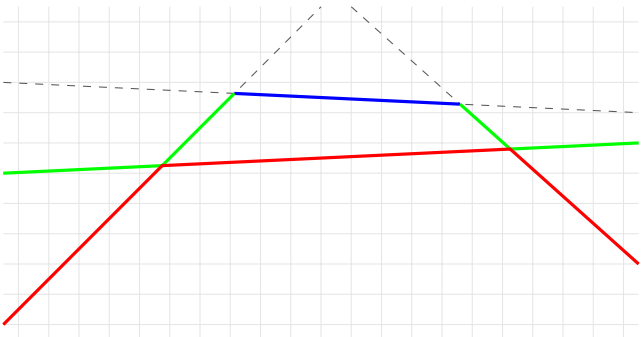
Subtask 4

$1 \leq N \leq 500000$
$1 \leq Q \leq 500000$



Subtask 4

$1 \leq N \leq 500000$
$1 \leq Q \leq 500000$



Subtask 4

$1 \leq N \leq 500000$
$1 \leq Q \leq 500000$

- Update the answer by the two neighbours of the linear function that gives the smallest value
- Also, use another instance of the same data structure to maintain the linear functions which are not used in the originally one
- Update the answer by querying on the second data structure
- Time complexity: $O((N+Q)\log N)$

Implementation

https://wcipeg.com/wiki/Convex_hull_trick#Fully_dynamic_variant

M1964 Injection

Charlie Li

Problem statement

- Given two arrays a[1..N] and b[1..M], find the minimum cost of f where f is an injective function from [1..N] to [1..M] such that a[i] = b[f(i)]

- The cost is defined to be the sum of all $|i - f(i)|$

Subtasks

- For all cases: 1 <= N <= M <= 500000
- Subtask 1(10 points): N = M
- Subtask 2(5 points): All a[i] are distinct, all b[i] are distinct
- Subtask 3(6 points): N, M <= 8
- Subtask 4(7 points): N, M <= 16
- Subtask 5(21 points): N, M <= 300
- Subtask 6(17 points): N, M <= 5000
- Subtask 7(34 points): No additional constraints

Subtask 2

- Subtask 2(5 points): All a[i] are distinct, all b[i] are distinct

- In this circumstances, there is only one possible function satisfying the conditions, so our task is to find the function.
- You can easily solve the problem by using any O(lg N) time algorithm to find the position of an element in b[i].

- You can do this either by first sorting b[i] and the do a binary search every time or use map or other data structure to achieve this.

- Time complexity: O(N lg N)

Subtask 3

- Subtask 3(6 points): N, M <= 8

- You can use exhaustion to do this
- If you use C++, you can use next_permutation

- Time complexity: O(N!)

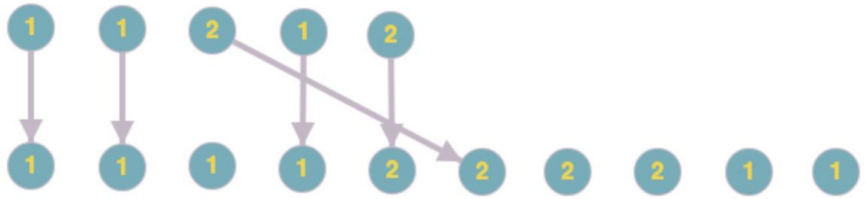
Subtask 4

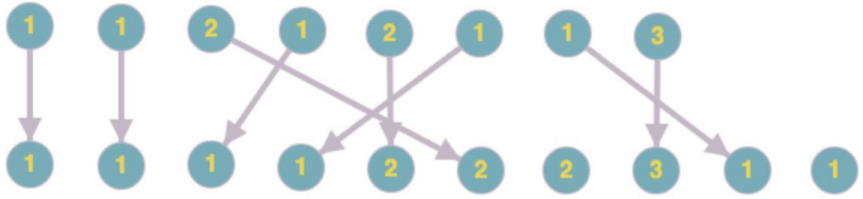
- Subtask 4(7 points): N, M <= 16
- To improve the runtime for exhaustion, you can use bitwise DP
- Time complexity: $O(N^2 2^N)$

Subtask 5

- Subtask 5(21 points): N, M <= 300

- It is natural to connect function with bipartite graph as the function's graph must be bipartite! (Look at the graph in sample 1 and 2!)





Subtask 5

- So we can build a bipartite graph using the input

- The nodes are labelled a(1..N) and b(1..M), so there are N+M nodes
- if a[i] == b[j], connect node a(i) and b(j) with weight abs(i - j)
- Then the answer would be the minimum weight perfect matching or -1 if perfect matching does not exists

- Time complexity: O((N+M)NM)
- Google it for implementation details if you are interested in

Observation 0

- For distinct elements, they have no effect on each other
- So we may calculate one by one the cost of mapping the occurrences of a particular element in a[] to their occurrences in b[]
- We may need to transform the problem a little bit

Observation 0

- For example, let's look at sample 1:
  1 1 2 1 2
  1 1 1 1 2 2 2 2 1 1

- Extracting the 1's occurrence:
  1 1 2 1 2
  1 1 1 1 2 2 2 2 1 1

  1 2 4
  1 2 3 4 9 10

Observation 0

- For example, let's look at sample 1:
  1 1 2 1 2
  1 1 1 1 2 2 2 2 1 1

- Extracting the 2's occurrence:
  1 1 2 1 2
  1 1 1 1 2 2 2 2 1 1

  3 5
  5 6 7 8

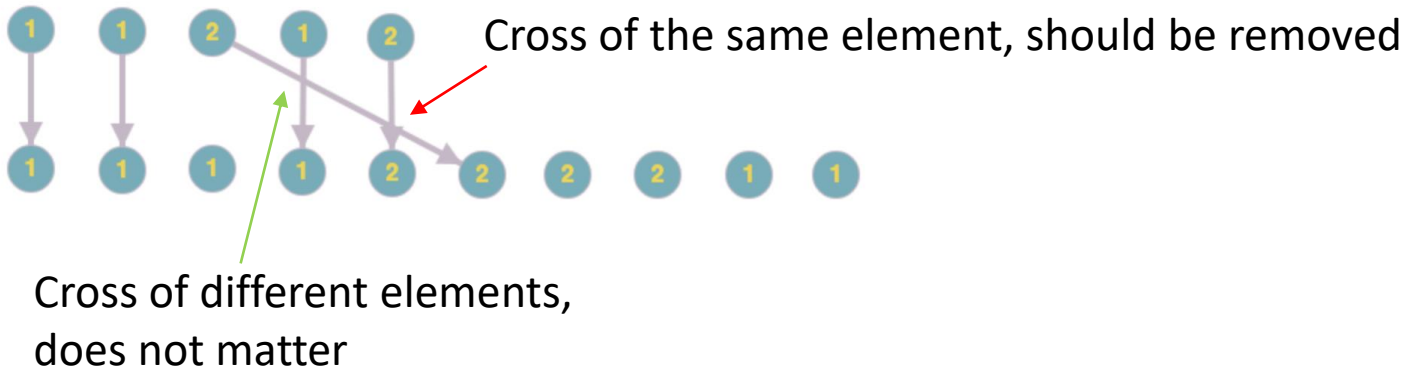
Observation 0

- So we may calculate the answers element-wise and the answer to the problem is the sum of all those answers
  (or -1 is there is any element cannot be mapped)

- So from now on, we may only consider the transformed data for one element only

Observation 1

- The resulting mapping graph of the same element should not have “crosses” in it since we can remove the “crosses” so that it is still as least as good as the one with “crosses”



Observation 1

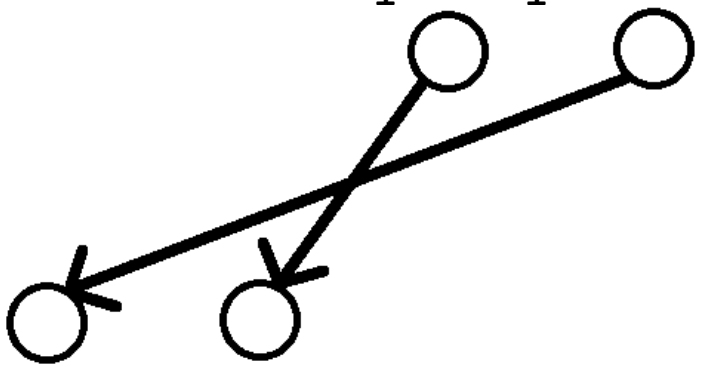
- More formally speaking, a mapping forms a cross if there is some $l_1 < l_2$ where $f(l_1) = r_1$, $f(l_2) = r_2$ and $r_2 < r_1$

- To remove the cross, we just need to change the mapping to $f(l_1) = r_2$, $f(l_2) = r_1$

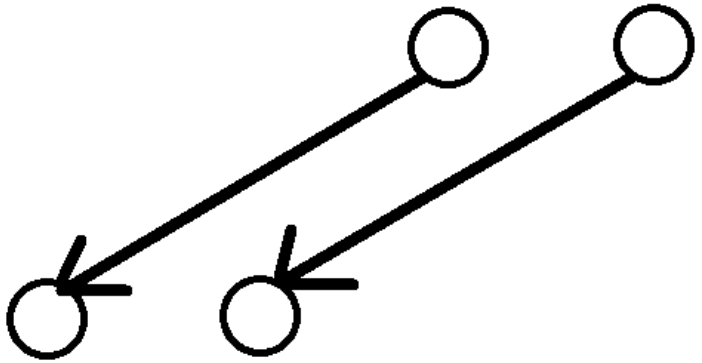
- To see that removing the cross does not increase the cost, there are totally 4 cases we need to investigate

Observation 1

- Case 1 (No difference): $r_1 <= l_1$
  - Before:

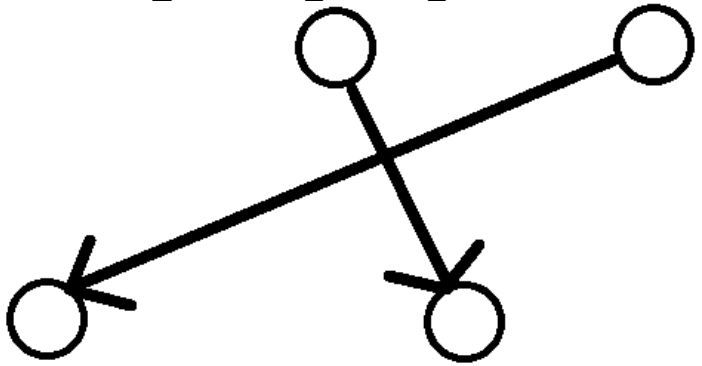


  - After:

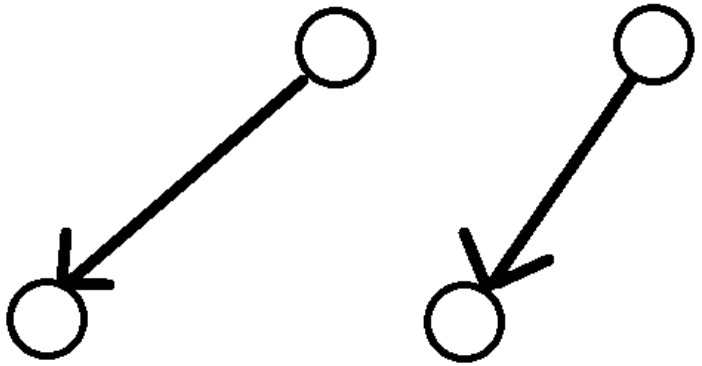


Observation 1

- Case 2 (Better): $r_2 <= l_1 < r_1$
  - Before:

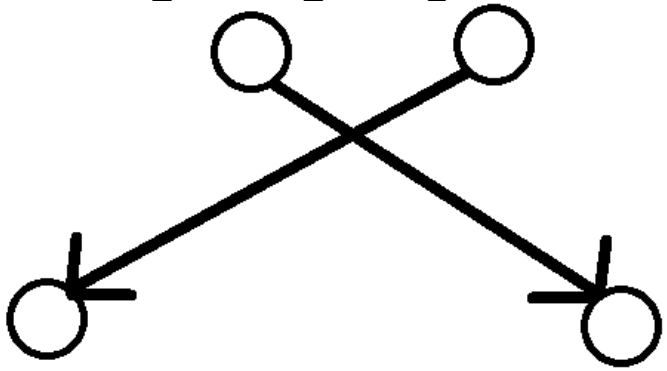


  - After:

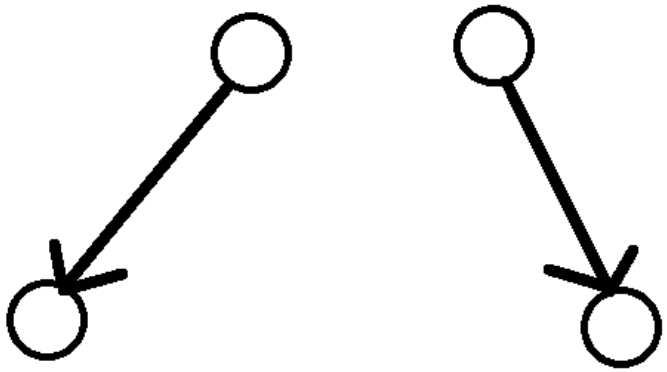


Observation 1

- Case 3 (Better): $r_2 <= l_2 < r_1$
  - Before:

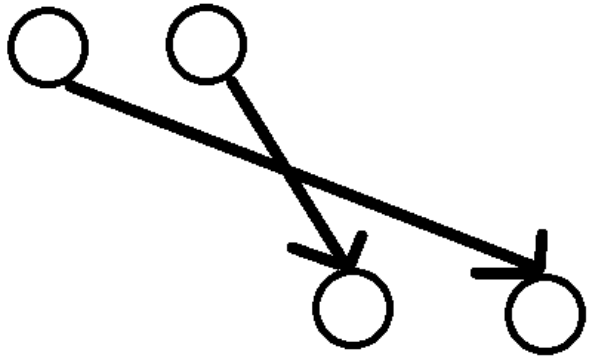


  - After:

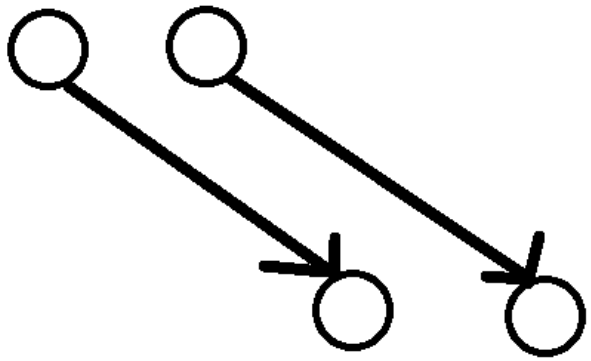


Observation 1

- Case 4 (No difference): $l_2 <= r_2$
  - Before:



  - After:



Observation 1

- So it is true that there will be at least one optimal solution without crosses.
  - i.e. If $i < j$, then $f(i) < f(j)$ (within the same element)

- Base on this observation, we can do a lot more.

Subtask 1

- Subtask 1(10 points): N = M

- (Using the transformed data)
- If any solution exists, there will be one and only one f such that it has no crosses within the same element.
- We can map the elements by ascending order.

- Time complexity: O(N)

Subtask 6

- Subtask 6(17 points): N, M <= 5000

- (Using the transformed data)
- Base on observation 1, we can use DP to solve this problem.

Subtask 6

- Let pa[i] be the position of the i^{th} occurrence in a[], similar for pb[i]

- State: dp[i][j] = minimum cost to map first i occurrence in a[] to first j occurrence in b[]
- Base case: dp[0][j] = 0
- Transition: dp[i][j] = min(dp[i][j-1], dp[i-1][j-1] + abs(pa[i] – pb[i]))

- Time complexity: O(NM)

Subtask 7

- Subtask 7(34 points): No additional constraints

- Can we optimize the previous DP?
- Yes! The intended solution is DP!
- But how?

Subtask 7

- First step, we need to reduce the number of states as its now the bottle neck.

- Let's merge two arrays into one!

Subtask 7

- Consider the 1's in sample 1.
  1 2 4
  1 2 3 4 9 10

  1 2   4
   1 2 3   4 9 10

1 1 2 2 3 4 4 9 10

- Let's call this array c[]

Subtask 7

1 1 2 2 3 4 4 9 10

- The red be the numbers from a[] and black numbers are from b[]
  - For numbers at the same position it does not matter if we put the one from a[] first or b[] first.
- Now we can define the new state as
  dp[i] = minimum cost to map those red numbers to those black numbers within the range [1..i]

Subtask 7

- Before finding the transition formula, let's fill in the DP table first to understand the definition of the state

1	1	2	2	3	4	4	9	10
inf	0	inf	0	0	1	0	0	0

Subtask 7

- From observation 1, we know that if there are the same number of red and black numbers in c[], then we can find the cost easily

- So the transition formula looks like this:
  - dp[i] = min(dp[i-1], dp[j-1] + cost of matching c[j..i])
    where j is the largest index such that there are the same number of red and black numbers in c[j..i]
- If such j does not exist, then dp[i] = dp[i-1]

Subtask 7

- This transition is O(N) if calculated naively (with observation 1) makes the overall time complexity $O(N^2)$

- To improve the complexity of this transition, we have two challenges to tackle.
  - Find the j quickly
  - Calculate the cost quickly

Subtask 7

- How to find the j quickly?
- Maintain the running difference of red and black numbers and use an array or any data structure you like to store its latest occurrence
- O(1) or O(lg n)

Subtask 7

- How to calculate the cost quickly?
- Given that j is the largest index such that c[j..i] contains the same number of red and black numbers
- Then we know that the red numbers are either all mapped to its left or all mapped to its right

Subtask 7

- E.g.:

| 1 1 | 2 2 3 4 4 9 10 : All right

1 | 1 2 | 2 3 4 4 9 10 : All left

| 1 1 2 2 | 3 4 4 9 10 : All right

1 1 2 2 | 3 4 | 4 9 10 : All left

1 1 2 2 3 | 4 4 | 9 10 : All right

Subtask 7

- Why is this true?

- Suppose this is not true, then we can find some red numbers at index l and r such that l is mapped to left and r is mapped to right (or vise versa), then there must be some j' in the range (l, r) such that c[j'..i] contain the same number of red and black numbers

Subtask 7

- How to calculate the cost quickly?

- We now know that the red numbers are either all mapped to its left or all mapped to its right

- So the cost can be calculated as: abs(sum of red – sum of balck) in the range
- This can be done using some data structures in O(lg N) or even O(1) by partial sum

Subtask 7

- Full Solution time complexity: O(N) or O(N lg N)
- 完