Task 1: `Problem Setter`

Authored by: Stuart Lim

Prepared by: Sng James

Editorial written by: Sng James

Subtask 1

Additional constraints: $1 \leq C \leq 1000, P = 1$

There is only one problem. Stuart can check if he can submit the problem to each contest, and submit the problem to the contest that gives the highest satisfaction. However, if submitting to that contest still gives negative satisfaction (i.e. $S_i < D_j$), then he should choose not to submit the problem instead.

Time complexity: $O(C)$

Subtask 2

Additional constraints: $1 \leq C, P \leq 1000$

Stuart can run the solution of subtask 1 on each problem, then sum up the satisfaction values he gains from submitting each problem.

Time complexity: $O(CP)$

Subtask 4

Additional constraints: None

First, sort the contest by their minimum quality M_i. Then, note that each problem can be submitted to a prefix of contests. For each problem, it is always optimal to submit it to the contest that gives the highest S_i.

Thus, we maintain the prefix max of S_i of the contests in the sorted order. Then, for each problem, we binary search for the contest with the largest M_i which we can submit to. The S_i value of the contest we should submit to is the corresponding prefix max.

Note that we still need to check if the prefix max we obtain is at least D_j. Solutions which do not take this into account should pass subtask 3 but fail this subtask.

Time complexity: $O((C + P) \log C)$

Task 2: `Shops`

Authored by: Benson Lin Zhan Li

Prepared by: Yu Zheyuan

Editorial written by: Yu Zheyuan

Subtask 1

Additional constraints: $n \leq 16$

Try all possible assignments of the shop and compute the shortest distance using Floyd Warshall as the number of edges may be large.

Time complexity: $O(2^n n^2)$

Subtask 2

Additional constraints: Graph is a path

The best way to assign shops is obviously to alternate between bunny and duck shops. We calculate the inconvenience for each city by seeing the left and right cities and take the maximum.

Time complexity: $O(n + m)$

Subtask 4

Additional constraints: $w_i = 1$

The answer is always 1. We just need to find a way to assign shops such that each vertex is adjacent to a vertex with a different shop.

We can do this by just running DFS and ensuring that when we visit a new vertex we assign it a different shop from the current one.

Time complexity: $O(n + m)$

Subtask 3

Additional constraints: Graph is a tree

It is easy to show that an assignment where each node has a different shop from all of its neighbors is optimal. Hence, if we assign shops based on the parity of a node's depth in the tree we will achieve such an assignment.

We just need to calculate the inconvenience for each city which can be done easily by just checking all its neighbours.

Subtask 5

It is easy to see that the minimal inconvenience of each city is the smallest edge weight connected to it.

Claim: We can construct an assignment where the inconvenience of each city is minimal

Proof: Let $f(u)$ be any closest neighbor to u. We construct a new graph of n edges containing $u- > f(u)$ for all u. This is a 1-out graph meaning that the structure of the graph is a cycle with trees pointing to each node. We note that the cycle is a 2-cycle. Hence, the graph is bipartite and thus we can 2-color it such that each u has a different color from $f(u)$ and assign shops as such.

If the inconvenience of each city is minimized the maximum is obviously minimized as well. Thus this coloring is optimal

Time complexity: $O(n + m)$

Task 3: `Toxic Gene 2`

Authored by: Benson Lin Zhan Li

Prepared by: Sng James

Editorial written by: Sng James and Yu Zheyuan

Subtask 1

Additional constraints: Bacteria 0 is Toxic

For each bacteria i from 1 to $n-1$, we can query it with bacteria 0. If there are 2 bacteria remaining at the end, bacteria i is Toxic. Otherwise, it is Regular. We use $n-1=999$ queries.

Subtask 2

1000 queries

We can modify the solution from subtask 1. When we query $\{0,i\}$, we don't know what type bacteria i is as we aren't guaranteed that bacteria 0 is Toxic. However, we know whether bacteria 0 and i are of the same type. Doing so, we can split the bacteria into 2 sets.

Afterwards, we can use 1 more query to find out the types of the 2 sets. Suppose a is a bacteria from set 1 and b is a bacteria from set 2. We query $\{a,b,b\}$, and if:

- there is 1 surviving bacteria, set 1 is Toxic
- there are 2 surviving bacteria, set 2 is Toxic

Total queries: $999+1=1000$., which gets 14.23 points.

123 queries

We can use divide and conquer to find one Toxic bacteria. Suppose our current range is $[l,r]$. Let $m=\frac{l+r}{2}$. We query $[l,m]$, and if some bacteria die, we recurse to $[l,m]$. Otherwise, we recurse to $[m,r]$. When the range has size 2, we can query $\{l,l,r\}$ to find a toxic bacteria and

return. This takes around $(O \log_2 n)$ queries. In particular, for $n = 1000$, it always finds a Toxic bacteria in 11 queries. Let the Toxic bacteria we found be T.

Afterwards, we split the 1000 bacteria into groups of size B. Let the current group be $a_0, a_1, ..., a_{B-1}$. We can perform a query with 2^i copies of a_i and 1 copy of T. Note that at the end, only Toxic bacteria will survive. If we let S be the number of surviving bacteria at the end, then bacteria a_i is Toxic iff the i-th bit in $S - 1$ is 1.

For each group, we use $2^{B+1} - 1$ samples, and we can set $B = 8$ for $n = 1000$. Thus, we use $11 + \lceil \frac{1000}{9} \rceil = 123$ queries.

44 queries

We modify the way we handle each group from the above subtask. For each group, we query $\{T, a_0, T, a_1, a_1, a_1, T, a_2, a_2, a_2, a_2, a_2, T, ..., T\}$. In particular, we have $2i + 1$ copies of a_i. Let $v_1, v_2, .., v_k$ be the result of the query, and let v_0 equal to the initial number of bacteria samples. Let $d_i = v_{i-1} - v_i$, which is the number of bacteria that die at iteration i.

Then, a_{i-1} is Regular iff d_i is odd. This is because it is only possible for iteration i to have odd d_i when there is only one remaining sample of a_{i-1}. All other bacteria would either be Toxic, have completely died, or have 2 samples die.

For each group, we use $2 \times \frac{B(B+1)}{2} + 1$ samples, and we can set $B = 31$ for $n = 1000$. Thus, we use $11 + \lceil \frac{1000}{31} \rceil = 44$ queries.

35 queries

We improve the way we query a group. For each group, we query $\{T, a_0, T, a_1, a_1, T, a_2, a_2, a_2, T, ..., T\}$. In particular, we have $i + 1$ copies of a_i. We define v_i and d_i similar to above. Then, for each i, a_{2i} is Regular iff d_i is odd, by a similar reasoning as above.

All that we have to do now is to determine if a_{2i+1} is Toxic or Regular. Consider d_i and d_{i+1}. The only things that cause a difference in the two are if:

- a_{2i} is Regular, causing a difference of 1.
- a_{2i+1} is Regular, causing a difference of 2.
- a_{2i+2} is Regular, causing a difference of 1.

However, we know that a_{2i} is Regular iff d_i is odd. Thus, we can check if a_{2i+1} is Regular by checking if $d_i - d_{i+1} - (d_i \mod 2) - (d_{i+1} \mod 2) = 2$.

For each group, we use $\frac{(B+1)(B+2)}{2}$ samples, and we can set $B = 43$ for $n = 1000$. Thus, we use $11 + \lceil \frac{1000}{43} \rceil = 35$ queries.

27 queries

We change the way we find a Toxic bacteria in the first phase of the solution. We first query $\{0, 0, 999\}$ to see if either is Toxic. If we are unable to find a Toxic bacteria, we know that 0 and 999 are of the same type, and thus there is at least 1 Toxic and 1 Regular in $[0, 998]$.

We then use 2 more queries: $\{(500 \text{ copies of } 0), 0, 1, 2, ..., 499\}$ and $\{(500 \text{ copies of } 0), 0, 500, 501, ..., 998\}$. Suppose the result of the first query is $u_1, u_2, ...u_j$ and the result of the second query is $v_1, v_2, ..., v_k$, where j and k are the sizes of the vectors returned.

If we know $500 < u_j < 1000$ or $500 < v_k < 1000$, we can conclude that bacteria 0 is Toxic.

Otherwise, we know that bacteria 0 is Regular. If $u_j < 500$, we know that there must be at least one Toxic bacteria in $[1, 499]$. Let the first Toxic bacteria be i. Then, we know that $j = 500 + i$ since bacteria i will use i iterations to kill the Regular bacteria in $[0, i - 1]$ and 500 turns to kill the 500 copies of bacteria 0. All other Regular bacteria to the right of i will die in strictly fewer turns.

Otherwise, if $u_j = 1000$, we know that there must be a Toxic bacteria in $[0] \cup [500, 998]$ and use a similar process as above to find it.

This improvement allows us to find one Toxic bacteria in 3 queries. Thus we use $3 + \lceil \frac{1000}{43} \rceil = 27$ queries.

26 queries

We can further improve the way we query a group. For each group, we query $\{(\lceil \frac{B}{2} \rceil \text{ copies of } a_{B-2}), T, a_0, T, a_1, a_1, T, ..., T, (\lceil \frac{B}{2} \rceil + 1 \text{ copies of } a_{B-1})\}$. In particular, we treat a_0 to a_{B-3} the same as before, but only use $\lceil \frac{B}{2} \rceil$ copies of a_{B-2} and $\lceil \frac{B}{2} \rceil + 1$ copies of a_{B-1} in front and behind.

If a_{B-2} and a_{B-1} are both Toxic, the machine will only run for at most $\lceil \frac{B-2}{2} \rceil = \lceil \frac{B}{2} \rceil - 1$ iterations and return at most that many values. Thus, if it returns more values than that, we know that at least one of a_{B-2} and a_{B-1} is Regular.

We know if a_{B-1} is Regular if the query returns $\lceil \frac{B}{2} \rceil + 1$ values. We can then remove its "contribution" to the v_i values by subtracting as necessary.

We know if a_{B-2} is Regular if the query returns $\lceil \frac{B}{2} \rceil$ values (then a_{B-1} is Toxic), or if after

removing the contribution from a_{B-1}, we still have $d_{B/2} > 0$. We can then similarly remove its "contribution" by subtracting as necessary.

Afterwards, we can treat the rest of the $B-2$ values as per normal. For each group, we use $\frac{(B-1)(B)}{2}+2\lceil\frac{B}{2}\rceil+1$ samples, and we can set $B = 44$ for $n = 1000$. Thus we use $3+\lceil\frac{1000}{44}\rceil = 26$ queries.

24 queries

We can further improve how we find a Toxic bacteria. We can query $\{(500 \text{ copies of } 0), 0, 1, 2, ..., 499\}$ first. Suppose the result is $u_1, u_2, ...u_j$. If $u_j < 1000$, we are able to conclude that either bacteria 0 is Toxic, or find a Toxic bacteria in $[1, 499]$. We then proceed to phase 2.

Otherwise, $u_j = 1000$. This tells us that the bacteria in $[0, 499]$ are all the same type. We can then query $\{0, 0, 999\}$ and $\{(500 \text{ copies of } 0), 0, 500, 501, ..., 998\}$ to find 1 Toxic bacteria and conclude the type of bacteria 0, which tells us the types of bacteria $[1, 499]$. We can then use the phase 2 solution on $[500, 999]$.

Thus, we use $\max(1 + \lceil\frac{1000}{44}\rceil, 3 + \lceil\frac{500}{44}\rceil) = 24$ queries.

21 queries

The final improvement is adding the idea for 136 queries to the current solution. Ignoring a_{B-1} and a_{B-2}, our arrangement is currently $T, a_0, T, a_1, a_1, T, a_2, a_2, a_2, T..., T$. Towards the end, we notice that we will need to repeat the same element a lot of times. We introduce a new way of determining a species.

Instead of always putting $T, a_i, ..., T$. We can put $T, a_i, ..., T, a_i, ..., T, a_i, ...$ We repeat a_i after each toxic $2x(x \geq 1)$ times and repeat the whole "$T, a_i, ...,$ " $2^k(k \geq 0)$ times. If a_i is normal, then it will "stop dying" at the x^{th} turn and exactly 2^{k+1} bactera will stop dying hence we are able to uniquely identify it. Lets call the string we get from a pair $S_{x,k}$.

Now, we have a set of strings which we can use to determine a species and each string can be used only once. This set contains of all $S_{x,k}$ and all $T, a_i, a_i, a_i...$ where the number of repetitions of a_i is odd. We sort the set by the string's size and greedily choose until our size of query string is as big as possible without exceeding 1000.

We define d_i similarly as above. We handle the case where we repeated a_i an odd number of times as above. Now we let $b_{i,0}, b_{i,1}, .., b_{i,j}, .b_{i,k}$ be the species corresponding all the strings $S_{i,j}$ which we included. In order to check if $b_{i,j}$ is Regular we can see if the $(j+1)$-th bit in $d_i - d_{i+1} - (d_i \mod 2) - (d_{i+1} \mod 2)$ is on.

Using our greedy, we can actually fit 49 strings. However, the space left would not be enough to use the optimization for 26 queries. It is actually more optimal to just use 48 strings from the set and have some room to spare. We then use the 26 query optimization to have a block size of 50. Thus, we use $\max(1 + \frac{1000}{50}, 3 + \frac{500}{50}) = 21$ queries, which gets full score.

Task 4: `Coin`

Authored by: Ling Yan Hao

Prepared by: Ling Yan Hao

Editorial written by: Ling Yan Hao

Subtask 1

Additional constraints: $1 \leq n \leq 7, 1 \leq m \leq 20$

Given a set of measurements, we can always try all possible weight orderings of the coins and determine all possible ranks of every coin.

Subtask 2

Additional constraints: $1 \leq n \leq 100, 1 \leq m \leq 400$

We will use the following important observation. The two statements are equivalent:

- It is possible to determine that coin x is lighter than coin y based on existing measurements.
- There exists coins $a_1, a_2, \ldots, a_\ell$ such that in the sequence $(x, a_1, a_2, \ldots, a_\ell, y)$, there exists a weighing between every consecutive pair of coins, and the coin in front is lighter.

So now, for every coin x, we can simply use depth first search to find all coins y which can be determined to be lighter than x. Similarly, we can find all coins y which can be determined to be heavier than x. A coin's rank is determined if and only if for every coin y, we can deduce the relative weights of x and y.

Subtask 3

Additional constraints: $1 \leq n \leq 1000, 1 \leq m \leq 4000$

Mostly same as subtask 2. But we can be a little more efficient. We can binary search for the value of k which makes the coin determined.

Subtask 4

Additional constraints: None

Use topological sort to obtain a permutation σ such that $\sigma(x[i]) < \sigma(y[i])$ for all i. Once we do this, we can assume that $x[i] < y[i]$ for all queries.

For all x, let $a[x]$ be the smallest value of y such that there exists a weighing (x, y) ($n + 1$ if such a y does not exist), and let $b[x]$ be the largest value of y such that there exist a weighing (y, x).

Claim: x is determined if and only if, for all $y < x$, $a[y] \leq x$, and for all $y > x$, $b[y] \geq x$.

Proof: suppose $a[y] > x$ for some $y < x$. Consider the sequence $1, 2, \ldots, n$ which removes y and inserts y immediately after x. We can check that this is also a valid ordering, but the rank of coin x in this ordering is now $x - 1$ instead of x. The similar situation holds if $b[y] < x$ for some $y > x$.

Suppose that $a[y] \leq x$ for all $y < x$. We will show that for all $y < x$, y must be lighter than x. We proceed by induction from $y = x - 1$ down to $y = 1$. The case $y = x - 1$ is clear; $a[x - 1] \leq x$ implies $a[x - 1] = x$ so that y is lighter than x. Now let y be arbitrary, we know that y is lighter than $a[y]$, but we also know that $y < a[y] \leq x$. If $a[y] \leq x-1$, then by induction hypothesis we know that $a[y]$ must be lighter than x. This completes the proof of the claim.

So it turns out that we only need to keep track of the changes in $a[]$ and $b[]$. Define an array $c[1..n]$, which is $c[x]$ is equal to the sum of

- number of $y < x$ with $a[y] > x$
- number of $y > x$ with $b[y] < x$.

Initialize $c[i] = n - 1$ for all i. Whenever we reduce $a[x]$ from y_1 to y_2, do a range decrement of $c[y_2 + 1, \ldots, y_1]$ by 1. We can do a similar decrement when $b[x]$ is increased. The query that makes coin x determined is equal to the first point in time where $c[x]$ becomes zero. We can do all these operations using segment tree.

Getting 50% of points

There is a simpler algorithm which can determine which coins are determined at the end.

Do Kahn's topological sort algorithm, and observe that for each coin x,

1. x was the only item in the queue when it was popped, or
2. there is some other element x' in the queue when we pop x

If (2) happens, x cannot have determined rank. If (1) happens, we can determine the maximum possible rank of x.

By running the same algorithm backwards, either (2) happens at least once (in which x cannot have determined rank), or (1) happens both times, in which we know them minimum and maximum possible rank.

Task 5: `Field`

Authored by: Stuart Lim

Prepared by: Stuart Lim

Editorial written by: Stuart Lim

Subtask 1

Additional constraints: $t = 1, n \leq 100, |a[i]|, |b[i]|, |c[i]|, |d[i]|, |x[j]|, |y[j]| \leq 400$

This is an almost classic single-source shortest-path problem on a grid. Maintaining information for an infinite grid is not possible, but due to the small coordinates used, it suffices to consider only the finite grid where $|x|, |y| \leq 401$. Choosing a grid slightly larger than the area where puddles can form allows us to go around puddles in certain situations such as the one shown in Sample Testcase 2.

Marking blocked points can be done naively by iterating over all points blocked by each puddle. We run breadth-first search (BFS) to find the distance of each unblocked point from $(0, 0)$.

Time complexity: $O(nm_{max}^2 + q)$

Subtask 4

Additional constraints: $t = 2, n \leq 100, q, |a[i]|, |b[i]|, |c[i]|, |d[i]|, m[j] \leq 400$

Since $m[j] \leq 400$, it suffices to consider only the finite grid where $|x|, |y| \leq 400$ because $m[j]$ is too small to reach any location outside of this grid.

Like in Subtask 1, we mark blocked points and run BFS from the origin. To answer each query, we iterate over every point in the grid to count the number of points with distance at most $m[j]$.

Time complexity: $O((n + q)m_{max}^2)$

Subtask 2

Additional constraints: $t = 1, n \leq 100, a[i] \equiv c[i] \equiv 0, b[i] \equiv d[i] \equiv -1 \pmod{10^7}$

Similar to Subtask 1, it is sufficient to consider the finite grid $|x|, |y| \leq 10^9 + 1$ which is slightly larger than the area where puddles are allowed to form. However, it is no longer possible to use BFS.

We split the entire grid into regions of size 10^7-by-10^7. Specifically, for integers u, v, a region contains all points (x, y) satisfying $10^7 u \leq x < 10^7(u + 1)$ and $10^7 v \leq y < 10^7(v + 1)$.

This new grid has three important properties:

1. Each point (x, y) satisfying $|x|, |y| \leq 10^9 + 1$ is in exactly one region. (The new grid contains extra points not satisfying $|x|, |y| \leq 10^9 + 1$ but we can ignore them.)
2. The points in any region are either all blocked or all unblocked.
3. If there exists a path from $(0, 0)$ to some (x, y), then there exists a shortest path from $(0, 0)$ to (x, y) that only enters regions at the corners. Refer to **Appendix A** for a proof.

Using the second property, instead of marking each individual blocked point, we can mark entire regions at once. For each puddle, we iterate over all regions that are blocked by it. In general, this runs in $O(n \cdot (\frac{m_{max}}{10^7})^2)$ time, where $(\frac{m_{max}}{10^7})^2$ estimates the number of regions.

Using the third property, we can run Dijkstra's algorithm to find the shortest distance from $(0, 0)$ to every corner in $O((\frac{m_{max}}{10^7})^2 \log(\frac{m_{max}}{10^7}))$. Let $C[0], C[1], C[2], C[3]$ be the four corners in the same region as (x, y), then the shortest distance from $O = (0, 0)$ to (x, y) is the minimum of $dist(O, C[i]) + dist(C[i], (x, y))$.

Time complexity: $O(n \cdot (\frac{m_{max}}{10^7})^2 + (\frac{m_{max}}{10^7})^2 \log(\frac{m_{max}}{10^7}) + q)$

Subtask 3

Additional constraints: $t = 1, n \leq 100$

We will generalise the idea of constructing a new grid. We construct dividing lines $x = X[u]$ and $y = Y[v]$ where X and Y are sets defined as follows.

$$X = \{-(10^9 + 1), 0, 10^9 + 2, a[1], b[1] + 1, a[2], b[2] + 1, \ldots, a[n], b[n] + 1\}$$
$$Y = \{-(10^9 + 1), 0, 10^9 + 2, c[1], d[1] + 1, c[2], d[2] + 1, \ldots, c[n], d[n] + 1\}$$

Let $X[u]$ and $Y[v]$ denote the u-th and v-th smallest element of X and Y respectively. The grid is divided into $(|X| - 1)(|Y| - 1) = O(n^2)$ rectangular regions containing all grid squares (x, y) where $X[u] \leq x < X[u + 1]$ and $Y[v] \leq y < Y[v + 1]$.

The three properties listed under Subtask 2 still apply. In particular, the proof of the third property in **Appendix A** is valid as $(0, 0)$ is a corner of a region and the proof does not require regions to have a fixed size of 10^7-by-10^7.

Marking blocked regions takes $O(n^3)$ time, while Dijkstra's Algorithm takes $O(n^2 \log n)$ time.

Time complexity: $O(n^3 + q)$

Subtasks 5 and 6

Additional constraints: $t = 2, n \leq 100, q \leq 200$. For Subtask 5 only, $a[i] \equiv c[i] \equiv 0, b[i] \equiv d[i] \equiv -1 \pmod{10^7}$

Since $m[j] \leq 10^9$, it suffices to consider only the finite grid where $|x|, |y| \leq 10^9$. As mentioned under Subtasks 2 and 3, a property of the new grid is that each point (x, y) satisfying $|x|, |y| \leq 10^9$ is in exactly one region. We can answer queries by summing up reachable points across all regions. (Note that the new grid contains other points not satisfying $|x|, |y| \leq 10^9$ as well but these points do not contribute to the sum, so the correctness of the solution is not affected.)

For the two subtasks, handling queries is largely identical, so this section will only discuss regions of a general size $h \times w$. For Subtask 5, $h = w = 10^7$.

Clearly, a blocked region will not contribute to the sum, so we only consider unblocked regions.

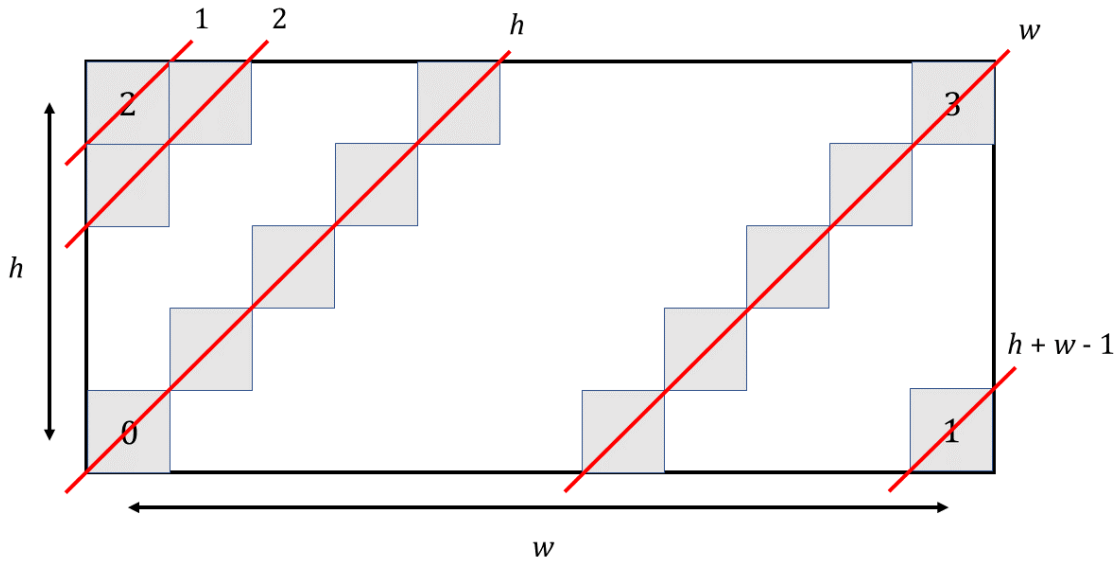

Figure 1: Example rectangular region with corners, dimensions and diagonals labelled

Let $C[0], C[1], C[2], C[3]$ be the corners of a region as shown in Figure 1. The placement and indexing is chosen such that the bit with value 1 indicates if the corner lies on the $-x$ or $+x$ border, whereas the bit with value 2 indicates if the corner lies on the $-y$ or $+y$ border. As shorthand, let $dist[0]$, $dist[1]$, $dist[2]$, $dist[3]$ be their respective distances from the origin. The region also has $h + w - 1$ diagonals of points each parallel to the lines $y = x$ and $y = -x$. Figure 1 shows the diagonals parallel to the line $y = x$.

We will try to use the principle of inclusion-exclusion (PIE) to count the number of points that can be reached. Let $S[0], S[1], S[2], S[3]$ be the sets of points that can be reached in $m[j]$ moves while also passing through the respective corners before that.

Unfortunately, it is difficult to find the intersection of three or more sets. However, consider what happens if $S[0] \cap S[3]$ is nonempty. At least one out of $h + w - 1$ diagonals is reachable via both $C[0]$ and $C[3]$, while the rest are reachable via at least one corner, so we know the total number of reachable points is $h \cdot w$ without evaluating PIE.

To identify this case, let $g[i]$ be the number of nearest diagonals reachable from corner $C[i]$ after passing through $C[i]$. Notice that if $dist[i] \leq m[j]$, then $g[i] = \min(h+w-1, m[j]-dist[i]+1)$, otherwise $g[i] = 0$. Now $S[0] \cap S[3]$ is nonempty if

$$g[0] + g[3] \geq h + w.$$

The right hand side is $h + w$ instead of $h + w - 1$ to account for a diagonal shared by $S[0]$ and $S[3]$. A similar second inequality can be obtained with $g[1]$ and $g[2]$. $S[1] \cap S[2]$ is nonempty if the second inequality is true. In summary, if at least one of the inequalities is true, then all points in the region are reachable and the answer is $h \cdot w$.

Appendix B shows that at least one of the inequalities is true if and only if

$$m[j] \geq \frac{h + w - 2 + \min(dist[0] + dist[3], dist[1] + dist[2])}{2}$$

Otherwise, $S[0] \cap S[3] = S[1] \cap S[2] = \varnothing$. Any intersections of three or more of the sets is also empty because it necessarily contains a pair of sets from opposite corners. Hence, the standard PIE expression reduces to

$$|S[0]| + |S[1]| + |S[2]| + |S[3]| - |S[0] \cap S[1]| - |S[1] \cap S[2]| - |S[2] \cap S[3]| - |S[3] \cap S[0]|.$$

If $m[j] < dist[i]$, then $C[i]$ cannot be reached, so $|S[i]| = 0$. Otherwise, $m[j] \geq dist[i]$, then as we have seen above, the $m[j] - dist[i] + 1$ diagonals nearest to $C[i]$ can be reached, and $|S[i]|$

is the sum of lengths of $m[j] - dist[i] + 1$ diagonals. Let $L(h, w, i)$ be the length of the i-th diagonal of a $h \times w$ rectangular region. We have

$$L(h,w,i) = \begin{cases} i & \text{if } i < \min(h,w) \\ \min(h,w) & \text{if } \min(h,w) \le i \le \max(h,w) \\ h+w-i & \text{if } \max(h,w) < i \end{cases}$$

Using the identity

$$1 + 2 + \cdots + i = \frac{i(i+1)}{2},$$

the required sum can be calculated in constant time.

Now we consider a side of the region, for example the $-y$ border containing w points with corners $C[0]$ and $C[1]$ at the ends. Let M be the point in the region that can be reached via $C[0]$ and $C[1]$ in the smallest number of moves. Mathematically, M minimises

$$\max(dist(O, C[0]) + dist(C[0], M), dist(O, C[1]) + dist(C[1], M)).$$

It can be proven that M is unique and is the k-th point along the $-y$ border counting from $C[0]$ where

$$k = \frac{w + 1 - dist[0] + dist[1]}{2}$$

Furthermore, $S[0] \cap S[1]$ is equal to the set of points with distance at most $m[j] - dist(O, M)$ from M. Refer to **Appendix C**.

We can reuse the method of counting by diagonals. If M is a corner, we can use the same method described above for $|S[i]|$. Otherwise, we can split the region into two smaller rectangles. For the $-y$ and $+y$ borders, we can split the region beside M as shown in Figure 2, where $w_1 = k$ and $w_2 = w - k$. M and M' are the corners of the rectangles.

We can sum up the lengths of $m[j] - dist(O, M) + 1$ diagonals in the first rectangle using $L(h, w_1, i)$. Since one move is spent moving from M to M', we sum up the lengths of $m[j] - dist(O, M)$ diagonals in the second rectangle using $L(h, w_2, i)$. It is possible that $m[j] - dist(O, M) = 0$, in which case the sum for the second rectangle is 0.

All of the above checks and calculations can be done in constant time for a single region. A single query can be answered in $O((\frac{m_{max}}{10^7})^2)$ or $O(n^2)$ time.

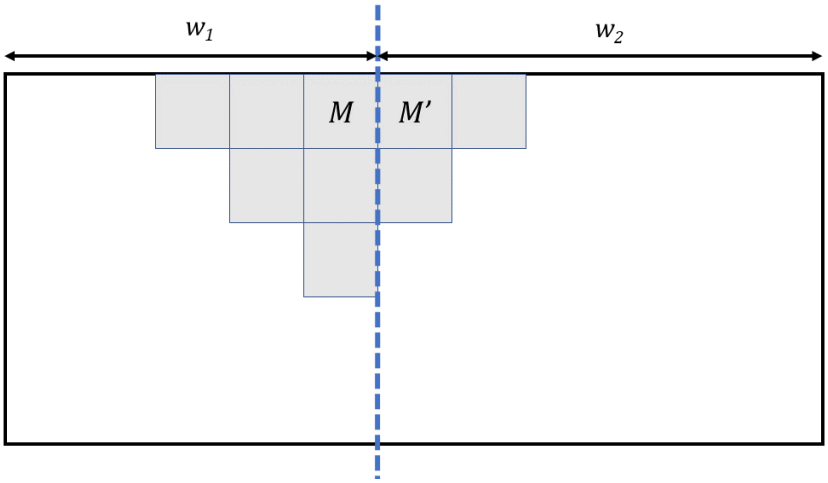

Figure 2: The region is split perpendicular to the $+y$ border

Time complexity: $O(n(\frac{m_{max}}{10^7})^2 + (\frac{m_{max}}{10^7})^2 \log(\frac{m_{max}}{10^7}) + q(\frac{m_{max}}{10^7})^2)$ for Subtask 5, $O(n^3 + n^2 q)$ for Subtask 6

Subtask 7

Additional constraints: $t = 2, n \leq 100, q \leq 5000$

Observe that regions can be split into one of the three categories below.

1. *Unstarted:* No point in the region can be reached when $m[j] < \min(dist[0], dist[1], dist[2], dist[3])$.
2. *Active:* At least one point in the region can be reached and sets corresponding to diagonally opposite corners do not intersect.
3. *Finished:* All points in the region can be reached when $m[j] \geq \frac{1}{2}(h+w-2+\min(dist[0]+dist[3], dist[1]+dist[2]))$.

As the number of moves increases, the state of a region can only change from unstarted to active, then from active to finished. Hence, we will try to maintain the number of grid squares that can be reached as the number of moves increases. We will answer queries offline in increasing order of $m[j]$.

We will maintain a variable fc that stores the sum of reachable points in finished regions. We will also maintain a set of all active regions A and calculate their individual contributions during each query in the same way as the previous subtask.

Define three types of events as follows.

- *start*: A region's state changes from unstarted to active. Insert the region into the set A. There are $O(n^2)$ such events, each running in $O(\log n)$ time using a STL set in C++.
- *finish*: A region's state changes from active to finished. Delete the region from the set A and increase fc by the number of points in the region. There are $O(n^2)$ such events, each running in $O(\log n)$ time.
- *query*: Answer a query by calculating the individual contribution of each region in A using PIE, summing them up and adding fc. There are q such events, each running in $O(|A|)$ time.

The author was not able to find an upper bound on $|A|$. It can be estimated to be about $8n$, but is actually about $9n$ in actual testcase data. Refer to **Appendix E**.

The number of moves for *start* and *finish* events are calculated based on $dist$ values which we find after running Dijkstra's Algorithm. We process events in increasing order of the number of moves. In case of ties, process *start* events first, followed by *finish* and *query* events. Sorting all events runs in $O((n^2+q)\log(n^2+q))$ time.

Time complexity: $O(n^3 + (n^2+q)\log(n^2+q) + nq)$ assuming $|A| = O(n)$

Subtask 8

Additional constraints: $t = 2, n \leq 100$

Recall that for each active region, each term in the PIE expression (e.g. $|S[0]|$, $|S[0] \cap S[1]|$) can be written as either one or two summations of the form

$$L(h,w,1) + L(h,w,2) + L(h,w,3) + \ldots$$

depending on whether the region needs to be split into two rectangles. We look back at the definition of $L(h,w,i)$. Define $L(h,w,0) = 0$ and

$$\Delta L(h,w,i) = L(h,w,i) - L(h,w,i-1)$$

for $1 \leq i \leq h+w-1$. Notice that $\Delta L(h,w,i)$ is initially equal to 1, but decreases to 0 when $i = \min(h,w)+1$, and again to -1 when $i = \max(h,w)+1$. Crucially, ΔL does not change frequently.

We maintain the number of reachable points in each active region, $ac[u][v]$, and two orders of rate of change, $\Delta ac[u][v]$ and $\Delta^2 ac[u][v]$ as the number of moves m increases. $\Delta ac[u][v]$ effectively counts the number of points in the active region with distance equal to m, which is the sum of a single $L(h, w, i)$ of each summation, taking the sign into account. $\Delta^2 ac[u][v]$ sums up the rates of changes of diagonal lengths, $\Delta L(h, w, i)$, across all summations, also taking sign into account. The three variables are technically functions of m, but updating them at every step is not feasible. Let $eff[u][v]$ be the last value of m for which we updated $ac[u][v]$, $\Delta ac[u][v]$, $\Delta^2 ac[u][v]$. Suppose we increase the number of moves by Δm without modifying $\Delta^2 ac[u][v]$, then $ac[u][v]$ and $\Delta ac[u][v]$ will change simultaneously as shown (indices $[u][v]$ omitted):

$$\begin{aligned} ac &\leftarrow ac + (\Delta ac + \Delta^2 ac) + (\Delta ac + 2 \cdot \Delta^2 ac) + \cdots + (\Delta ac + \Delta m \cdot \Delta^2 ac) \\ &= ac + \Delta m \cdot \Delta ac + \frac{\Delta m(\Delta m + 1)}{2} \cdot \Delta^2 ac \\ \Delta ac &\leftarrow \Delta ac + \Delta m \cdot \Delta^2 ac \end{aligned}$$

We also maintain ac_{global}, Δac_{global} and $\Delta^2 ac_{global}$, which are the sums of ac, Δac and $\Delta^2 ac$ over all active regions. The last value of m for which the three values are updated is eff_{global}. Without changes to $\Delta^2 ac_{global}$, when m increases, the other two variables change in the same way as their single-region counterparts. Finally, the sum of contributions of finished regions, fc, must also be maintained.

Similar to Subtasks 3 and 5, after running Dijkstra's Algorithm, we create events based on our $dist$ results and handle events in increasing order of number of moves m, but the events are different in this subtask. These are explained below. Note that before each event, we always update ac_{global}, Δac_{global}, $\Delta^2 ac_{global}$ and eff_{global}.

- *finish*: A region is finished.
  - Update $ac[u][v]$, $\Delta ac[u][v]$, $\Delta^2 ac[u][v]$ and $eff[u][v]$ for this region only. Now the global and regional last-updated times eff_{global} and $eff[u][v]$ are equal.
  - Subtract $ac[u][v]$, $\Delta ac[u][v]$, $\Delta^2 ac[u][v]$ from ac_{global}, Δac_{global}, $\Delta^2 ac_{global}$ respectively.
  - Add the number of points in the region to fc.
  - The number of points might **not** be equal to $ac[u][v]$ because our method to calculate the number of reachable points only works for active regions. The reason we update $ac[u][v]$, $\Delta ac[u][v]$, $\Delta^2 ac[u][v]$ is to determine how much to remove from ac_{global}, Δac_{global}, $\Delta^2 ac_{global}$.
  - There is 1 *finish* event for each region.
- *update-delta*: $\Delta^2 ac$ values change as ΔL values change

  - Update $ac[u][v]$, $\Delta ac[u][v]$, $\Delta^2 ac[u][v]$ and $eff[u][v]$ for this region only. Now the global and regional last-updated times eff_{global} and $eff[u][v]$ are equal.
  - Increase each of ac_{global}, Δac_{global}, $\Delta^2 ac_{global}$, $ac[u][v]$, $\Delta ac[u][v]$, $\Delta^2 ac[u][v]$ by the change in ΔL.
  - Since $\Delta^2 ac_{global}$ and $\Delta^2 ac[u][v]$ change, we adjust ac_{global}, Δac_{global}, $\Delta^2 ac_{global}$, $ac[u][v]$, $\Delta ac[u][v]$, $\Delta^2 ac[u][v]$ manually after updating them to m moves. However, we do not need to update the other regions.
  - The above updates should not happen when the region is already finished. Either check if the region is finished first or avoid creating *update-delta* events with the same or larger number of moves as the *finish* event in the same region.
  - The PIE expression consists of at most 12 summations. Each summation has at most 3 changes to ΔL, producing at most 36 *update-delta* events per region.

- *query*: Answer a query by returning $ac_{global} + fc$. There are q such events.

In case of ties, *finish* and *update-delta* events should be processed before *query* events. Each event can be handled in constant time.

Time complexity: $O(n^3 + (n^2 + q)\log(n^2 + q))$

Subtask 9

Additional constraints: $t = 2$

Due to the large number of *update-delta* events, the process of sorting and handling events has a large constant factor. We can reduce this constant by reducing the number of *update-delta* events we create. It turns out that many *update-delta* events for the same region happen at the same number of moves. We can merge such *update-delta* events by summing up the changes in ΔL. We can further remove events that sum up to a change of 0.

Other optimisations include:

- Marking blocked regions in $O(n^2)$ instead of $O(n^3)$ (refer to **Appendix D**)
- Avoid running Dijkstra's algorithm and creating events beyond 10^9 moves and compress their parameters into a single 64-bit integer per event

Time complexity: $O((n^2 + q)\log(n^2 + q))$

Appendix A: Property of Shortest Paths

We prove that if there exists a path from $(0, 0)$ to (x, y), then there exists a shortest path from $(0, 0)$ to (x, y) that only enters regions at the corners.

Let P be a shortest path from $(0, 0)$ to (x, y). Let $R[1], R[2], \ldots, R[k]$ be the regions that P visits in order. $R[i] \neq R[i + 1]$ for $1 \leq i \leq k - 1$. Let $(0, 0) = V[1], V[2], \ldots, V[k]$ be the first points which the path P visits in the regions $R[1], R[2], \ldots, R[k]$ respectively. Also let $V[k + 1] = (x, y)$.

Note that P does not visit the same region more than once. In other words, all $R[i]$ are pairwise distinct. Assume for the sake of contradiction that there exist i, j such that $i + 2 \leq j$ and $R[i] = R[j]$. Since region $R[i] = R[j]$ is completely unblocked, it is possible to move directly from $V[i]$ to $V[j]$ within the region, using fewer moves than visiting $V[i + 1], \ldots, V[j - 1]$ along the way. Hence, we found a path shorter than P, which is a contradiction.

The length of P can be written as

$$dist(V[1], V[2]) + dist(V[2], V[3]) + \cdots + dist(V[k], V[k + 1])$$

where $dist(V[i], V[i + 1])$ denotes the minimum number of moves required to move from $V[i]$ to $V[i + 1]$. By definition of P, this is the shortest distance between $(0, 0)$ and (x, y). Denoting $V[i] = (x[i], y[i])$, we have

$$dist(V[i], V[i + 1]) = |x[i] - x[i + 1]| + |y[i] - y[i + 1]|$$

because the regions $R[i]$ and $R[i + 1]$ (or just $R[k]$ if $i = k$) are completely unblocked, which allows a direct path between the two points.

Notice that by the definition of the new grid, $(0, 0)$ is a corner of the region $R[1]$. With reference to P, we construct another path P' such that, starting from $(0, 0)$ in $R[1]$, we repeatedly move in a straight line to the nearest corner in the next region until reaching a corner in $R[k]$, then take any shortest path from that corner to $V[k + 1] = (x, y)$. For $i = 2, 3, \ldots, k$ in order, we essentially want to move $V[i]$ to the corner of $R[i]$ described by the rule above, one at a time. Hence, P' satisfies the condition that regions are only entered at the corners. We also show that the distance cannot increase.

Suppose we are at step i for some i. We have already moved $V[2], \ldots, V[i - 1]$ to corners of $R[2], \ldots, R[i - 1]$. Now we want to move $V[i]$. Without loss of generality, suppose we move in the $+x$ direction from $R[i - 1]$ to $R[i]$. Other orientations are similar by rotation.

- $V[i]$ will necessarily be on the $-x$ border of $R[i]$, so we cannot change the x-coordinate

of $V[i]$.

- Since $V[i-1]$ must lie on either the $+y$ or $-y$ border of $R[i-1]$, by using the same y-coordinate, $V[i]$ will lie on either the $+y$ or $-y$ border of $R[i]$.

$V[i] = (x[i], y[i])$ would be moved to $V'[i] = (x[i], y[i-1])$. It lies on two borders of $R[i]$, therefore it is a corner of $R[i]$. We see that only two terms $dist(V[i-1], V[i])$ and $dist(V[i], V[i+1])$ are affected as follows. The triangle inequality is used.

$$
\begin{aligned}
&dist(V[i-1], V'[i]) + dist(V'[i], V[i+1]) \\
&= (|x[i-1] - x[i]| + |y[i-1] - y[i-1]|) + (|x[i] - x[i+1]| + |y[i-1] - y[i+1]|) \\
&= |x[i-1] - x[i]| + |x[i] - x[i+1]| + |y[i-1] - y[i+1]| \\
&\leq |x[i-1] - x[i]| + |x[i] - x[i+1]| + |y[i-1] - y[i]| + |y[i] - y[i+1]| \\
&= (|x[i-1] - x[i]| + |y[i-1] - y[i]|) + (|x[i] - x[i+1]| + |y[i] - y[i+1]|) \\
&= dist(V[i-1], V[i]) + dist(V[i], V[i+1])
\end{aligned}
$$

This shows that at each step, moving $V[i]$ as described will never increase the total length of the path. Therefore the length of P' is at most the length of P. On the other hand, by definition P is a shortest path, so the length of P' is at least the length of P. Therefore P and P' have the same length, giving the result that P' is also a shortest path.

Appendix B: Inequality for Diagonally Opposite Corners

Let $g'[i] = m[j] - dist[i] + 1$. We already defined $g[i]$ as the number of nearest diagonals reachable from corner $C[i]$ after passing through $C[i]$. $g[i]$ can be written as follows:

$$
g[i] = \begin{cases} 0 & \text{if } g'[i] \leq 0 \\ g'[i] & \text{if } 1 \leq g'[i] \leq h + w - 1 \\ h + w - 1 & \text{if } h + w \leq g'[i] \end{cases}
$$

We will show that $g[0] + g[3] \geq h + w$ if and only if $g'[0] + g'[3] \geq h + w$.

- Suppose $g[0] = 0$, so $C[0]$ is not reachable. $C[0]$ lies on the furthest diagonal from $C[3]$, which is the $(h + w - 1)$-th nearest diagonal, therefore $g[3] < h + w - 1$. Adding $g[0]$ and $g[3]$, we conclude that the first inequality is false. Since $g'[0] \leq g[0]$ and $g'[3] \leq g[3]$, we conclude that the second inequality is also false. Similarly, if $g[3] = 0$ then both inequalities are false.

- Now suppose $g[0] = h + w - 1$. By the same line of reasoning, $g[3] > 0$. Adding $g[0]$ and $g[3]$, we conclude that the first inequality is true. Since $g'[0] \geq g[0]$ and $g'[3] \geq g[3]$, we conclude that the second inequality is also true. Similarly, if $g[3] = h + w - 1$ then both inequalities are true.
- The only remaining case is when both $g[0]$ and $g[3]$ are between 0 and $h+w-1$ exclusive. $g[0] = g'[0]$ and $g[3] = g'[3]$ so the two inequalities are either both true or both false.

The inequality $g'[0] + g'[3] \geq h + w$ can be rewritten as

$$m[j] \geq \frac{h + w - 2 + dist[0] + dist[3]}{2}$$

Similarly, $g[1] + g[2] \geq h + w$ if and only if $g'[1] + g'[2] \geq h + w$ and we rewrite the latter inequality into a similar form.

Therefore, at least one of $g[0] + g[3] \geq h + w$ and $g[1] + g[2] \geq h + w$ is true if and only if

$$m[j] \geq \frac{h + w - 2 + \min(dist[0] + dist[3], dist[1] + dist[2])}{2}$$

Appendix C: Points Reachable From Adjacent Corners

We are interested in the point(s) M that can be reached by two corners along the same border, say $C[0]$ and $C[1]$, minimising

$$\max(dist(O, C[0]) + dist(C[0], M), dist(O, C[1]) + dist(C[1], M)).$$

It can be proven that M is unique.

First, observe that paths from $(0, 0)$ to (x, y) either all have even lengths or all have odd lengths. Colour the grid in a black and white chessboard-like pattern such that $(0, 0)$ is black. After each move, the colour of the current location (a, b) changes, so after m moves, the current location (a, b) is black if and only if m is even. The colour of (x, y) therefore determines the parity of m that is required to land on (x, y).

Next, we note that the first point that can be reached by both $C[0]$ and $C[1]$ necessarily lies along the border they are on, because all other points require some number of extra moves perpendicular to the border. This allows us to consider just the points along the border. Label

them from $V_1 = C[0]$ to $V_w = C[1]$. Let a_k be the number of moves for square V_k to be reachable from both $C[0]$ and $C[1]$, then

$$a_k = \max(dist[0] - 1 + k, dist[1] + w - k).$$

Since the two terms describe different ways to move to the same location, they have the same parity and their signed difference

$$(dist[0] - 1 + k) - (dist[1] + w - k)$$

is always even. Furthermore, if k increases by 1, then the first term increases by 1 and the second decreases by 1. Clearly a_k is minimum when the signed difference is 0.

The signed difference can be rearranged to get

$$(dist[0] - dist[1]) - (w + 1) + 2k.$$

Note that $|dist[0] - dist[1]| \leq w - 1$, where the extreme values are obtained when a shortest path to $C[0]$ passes through $C[1]$ or vice versa. We have

$$\begin{aligned} -(w-1) - (w+1) \leq (dist[0] - dist[1]) - (w+1) &\leq (w-1) - (w+1) \\ -2w \leq (dist[0] - dist[1]) - (w+1) &\leq -2 \end{aligned}$$

Since $1 \leq k \leq w$ by definition, it is always possible to make the signed difference equal to 0 by setting

$$k = \frac{(w+1) - (dist[0] - dist[1])}{2}.$$

Hence, we have found the unique value of k that minimises a_k. M is the point represented by this value of k, V_k. As described above, the minimum number of moves needed for V_k to be reachable from $C[0]$ and $C[1]$ is $dist(O, V_k) = dist[0] - 1 + k = dist[1] + w - k$. If $m[j] < dist(O, V_k)$, then $S[0] \cap S[1] = \varnothing$ and we are done. Otherwise, V_k is the point with smallest distance in $S[0] \cap S[1]$.

Recall that we numbered points along the border from V_1 at $C[0]$ to V_w at $C[1]$. For some point J, suppose that it has a perpendicular distance of p from V_j on the border. J is in $S[0] \cap S[1]$ if and only if

$$\begin{aligned}
\max(dist[0]-1+j, dist[1]+w-j)+p &\le m[j] \\
\max(dist[0]-1+k-(k-j), dist[1]+w-k+(k-j))+p &\le m[j] \\
\max(dist(O,V_k)-(k-j), dist(O,V_k)+(k-j))+p &\le m[j] \\
dist(O,V_k)+|k-j|+p &\le m[j] \\
dist(O,V_k)+dist(V_k,V_j)+dist(V_j,J) &\le m[j] \\
dist(O,V_k)+dist(V_k,J) &\le m[j] \\
dist(V_k,J) &\le m[j]-dist(O,V_k)
\end{aligned}$$

Appendix D: Marking Blocked Regions Faster

Recall that each puddle covers a rectangular area of regions. Specifically, puddle k covers all regions (i, j) where $i_1 \le i < i_2$, $j_1 \le j < j_2$ for some i_1, i_2, j_1, j_2 to be found. Using the definition of the grid, $X[i_1] = a[k], X[i_2] = b[k] + 1, Y[j_1] = c[k], Y[j_2] = d[k] + 1$, so each of i_1, i_2, j_1, j_2 can be found using binary search on the sets X and Y in $O(\log n)$ time.

Let $p[i][j]$ be the number of puddles covering the region (i, j). Clearly, region (i, j) is blocked if and only if $p[i][j] > 0$. Define z as the inverse prefix sum array of p. In other words, taking the prefix sum of z would produce p. Then incrementing all $p[i][j]$ in the ranges $i_1 \le i < i_2$, $j_1 \le j < j_2$ is equivalent to

- Incrementing $z[i_1][j_1]$ and $z[i_2][j_2]$; and
- Decrementing $z[i_1][j_2]$ and $z[i_2][j_1]$.

We make $O(n)$ updates to z, then construct the prefix sum p in $O(n^2)$ time, so the total time complexity of this step is $O(n^2)$.

Appendix E: Estimate of the Number of Active Regions and Checking Actual Testcases

Note that this estimation process is only required for Subtask 7. The full solution does not rely on any information in this section.

Call the frontier a set of points with equal distance m from the origin. By observing frontiers while running BFS on grids, one can see that a frontier consists mostly, if not entirely, of points along diagonal lines parallel to $y = \pm x$. A frontier intersects a region if and only if the region

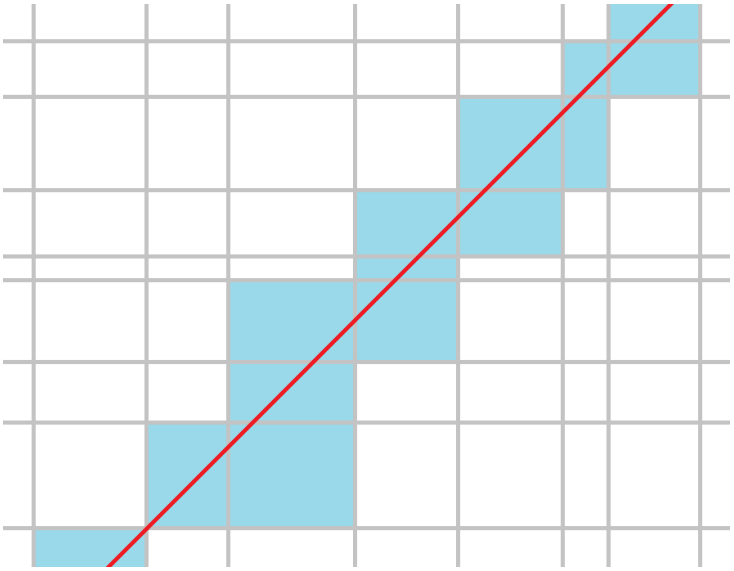

Figure 3: A diagonal segment of a frontier (red) intersects regions (turquoise)

contains a point that has a distance of m from the origin. Unfortunately, such a definition includes regions that just became finished at m moves and excludes active regions that have all internal points reachable in strictly fewer than m moves. For the sake of proceeding with the estimation, we assume such cases are rare and/or cancel out.

For a frontier segment parallel to $y = x$ such as shown in Figure 3, the truly active regions form a path in the $+x$ and $+y$ directions, where the region following (u, v) may be $(u + 1, v)$, $(u + 1, v + 1)$ or $(u, v + 1)$. If the frontier overlaps a rectangular area of $h \times w$ regions, it would intersect at most $h + w$ of them.

We can imagine the frontier to spread outward in each of the four quadrants separately in their own directions. Suppose among the approximately $2n$ dividing lines $x = X[u]$, r of them have $x > 0$. Similarly, among the approximately $2n$ dividing lines $y = Y[v]$, s of them have $y > 0$. The number of regions intersected by the frontier would be

$$(r + s) + ((2n - r) + s) + ((2n - r) + (2n - s)) + (r + (2n - s)) = 8n$$

By the assumptions listed previously, this is a reasonable estimate of the number of active regions.

This estimate is good enough for this task. Comparing with the actual set of testcases, the ratio of maximum active regions over all numbers of moves to n is 9.01 at maximum, with a mean of 4.75 and standard deviation of 2.99. Figure 4 shows the distribution over all testcases. Note that the testcases include handwritten and randomly generated data using several different generators and different parameters, which could explain the wide variation.

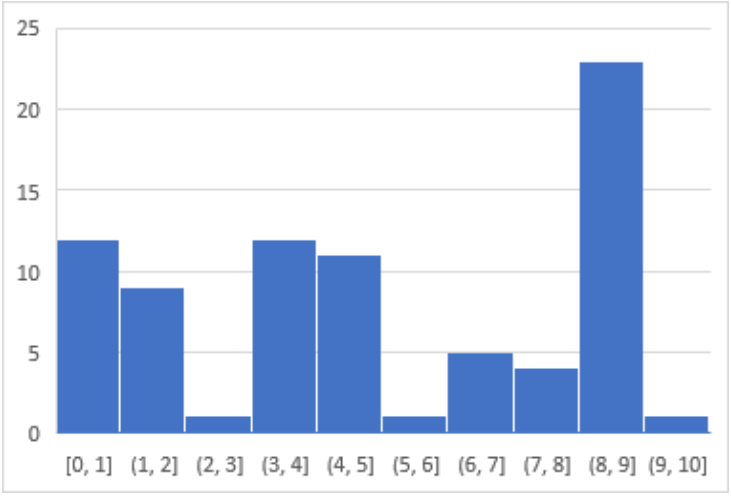

Figure 4: Histogram of ratio of maximum active regions to n in used testcases

ID	N= 10	20	30	40	60	80	100	150	200	250	300	400	500	600	700	800	900	1000
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	7.90	6.80	6.43	6.15	5.13	4.58	4.34	3.35	2.76	2.50	2.06	1.89	1.56	1.38	1.24	1.15	1.03	0.95
3	5.80	4.80	3.93	3.30	2.67	2.00	1.96	1.41										
4	3.60	1.30	0.47	0.25	0.20	0.30	0.06	0.04										
5	8.40	8.20	8.13	8.10	8.07	8.05	8.04	8.03	8.02	8.02	8.01	8.01	8.01	8.01	8.01	8.01	8.00	8.00
6	1.30	1.35	1.27	0.38	0.13	0.13	0.21	0.19	0.09	0.08	0.07	0.09	0.06	0.04	0.07	0.03	0.02	0.02
7	4.40	4.85	4.20	4.10	4.07	4.08	4.04	4.04	4.04	4.02	4.01	4.01	4.01	4.01	4.01	4.01	4.00	4.00
8	2.30	1.55	0.27	0.50	0.05	0.11	0.08	0.08										
9	8.40	8.20	8.13	8.10	8.07	8.05	8.04	8.03	8.02	8.02	8.01	8.01	8.01	8.01	8.01	8.01	8.00	8.00
10	8.50	8.00	8.00	7.45	7.43	7.43	7.24	6.67	6.35	5.86	5.67	5.04	4.62	4.19	3.85	3.49	3.15	2.95
11	8.40	7.70	7.93	7.70	7.43	7.38	6.98	6.53	5.86	5.56	5.27	4.66	4.20	3.85	3.52	3.16	2.90	2.71
12	8.20	7.35	6.80	6.30	5.53	4.90	4.02	2.79										
13	7.80	6.30	5.63	4.90	4.50	3.66	3.32	2.71										
14	8.40	8.20	8.13	8.10	8.08	8.05	8.02	8.01	8.05	7.96	8.05	7.85	7.92	7.95	7.92	7.91	7.91	7.90
15	8.40	8.20	8.13	8.10	8.07	8.05	8.04	8.03	8.02	8.02	8.01	8.01	8.01	8.01	8.01	8.01	8.00	8.00
16	8.40	8.20	8.13	8.10	8.03	8.03	8.03	8.03	7.97	7.98	8.00	8.00	8.00	7.90	7.90	7.94	7.86	7.84
17	8.30	8.00	7.77	8.05	8.00	7.69	7.92	7.60	7.80	7.50	7.44	7.09	6.99	6.41	6.93	6.28	6.73	6.09
18	8.40	8.20	8.13	8.10	8.07	8.05	8.04	8.03	8.02	8.01	8.01	8.01	8.01	8.01	8.00	8.01	8.00	8.01
19	3.60	3.60	3.77	3.85	3.82	3.91	3.92	3.95										
20	3.60	3.85	3.77	3.78	3.88	3.91	3.90	3.96										
21	3.60	3.45	3.73	3.85	3.82	3.94	3.93	3.95										
22	3.40	3.65	3.87	3.83	3.80	3.86	3.92	3.95	3.97	3.96	3.97	3.98	3.98	3.99	3.99	3.99	3.99	4.00
23	1.50	1.35	1.30	1.40	1.35	1.45	1.29	1.35										
24	1.90	1.50	1.73	1.60	1.32	1.44	1.37	1.32										
25	1.20	1.10	1.07	1.05	1.03	1.03	1.02	1.01	1.01	1.01	1.01	1.01	1.00	1.00	1.00	1.00	1.00	1.00
26	1.20	1.10	1.07	1.05	1.03	1.03	1.02	1.01	1.01	1.01	1.01	1.01	1.00	1.00	1.00	1.00	1.00	1.00
27	4.60	4.20	4.20	4.10	4.07	4.05	4.04	4.04	4.02	4.02	4.01	4.01	4.01	4.01	4.01			
28	1.60	1.55	1.40	1.35	1.30	1.36	1.38	1.29										
29	8.40	8.20	8.13	8.10	8.07	8.05	8.04	8.03	8.02	8.02	8.01	8.01	8.01	8.01	8.01	8.00	8.00	8.00
30	8.20	8.10	7.87	7.65	7.37	7.30	7.24	6.69	6.27	6.03	5.82	5.18	4.70	4.25	3.80	3.50	3.20	2.96
31	5.40	5.20	4.93	5.15	5.30	4.80	5.10	4.99	5.01	4.85	4.92	4.95	5.00	4.88	4.84	4.93	4.84	4.87
32	4.70	4.45	4.27	4.33	4.13	4.31	4.48	4.12	4.31	4.26	4.37	4.51	4.07	4.15	4.36	4.18	4.09	4.07
33	5.00	4.75	4.37	4.30	4.20	4.50	4.40	4.15	4.07	4.13	4.33	4.10	4.16	4.29	4.22	4.34	4.15	4.06
34	4.70	4.25	4.13	4.18	4.02	3.96	3.71	3.61	3.46	3.46	3.28	2.96	2.73	2.59	2.31	2.27	2.09	2.02
35	4.40	4.00	3.90	4.00	3.87	3.88	3.76	3.83	3.81	3.80	3.78	3.81	3.80	3.79	3.77	3.79	3.79	3.78

Figure 5: Ratio of maximum active regions to n as n varies and other parameters are kept constant. Larger values are green and smaller values are red

The randomly generated tests used in the task can be split into 35 different types ignoring differences in n. Figure 5 shows the change in ratio of maximum active regions to n for $10 \leq n \leq 1000$. For each type, the ratio mostly either decreases or stays the same as n increases. The maximum ratio of 8.50 is attained when $n = 10$.

Of course, since the author could not produce a provable upper bound for the number of active regions, there was not a clear way to generate testcases with the intention of maximising the number of active regions. It is entirely possible that a testcase exists with a ratio much larger than 9.