Task 1: `Monsters`

Authored by: Brian Lee Jun Siang

Prepared by: Brian Lee Jun Siang

Editorial written by: Brian Lee Jun Siang

Subtask 1

Additional constraints: $k = 1$

There is only 1 mine. We can compute the minimum cost needed if the mine was not used, and if the mine was used separately.

Case 1: If the mine is not used at all, sum up the health of all the monsters.

Case 2: Suppose the mine is at position $x[1]$. Since the mine is used, add 1 to the total cost. Then, the cost needed to kill monster i will be the minimum of $h[i]$ and $|a[i] - x[1]|$. Sum this value up over all monsters.

The final answer is the minimum cost over both cases.

Time complexity: $\mathcal{O}(n)$

Subtask 2

Additional constraints: $k = 2$

This is an extension from subtask 1. Now, Try all possibilities of using none of the mines, only mine 1, only mine 2, or using both mines and take the minimum cost needed over all cases.

Time complexity: $\mathcal{O}(n)$

Subtask 3

Additional constraints: $n, k \leq 18$

The small constraints motivate a brute force solution. Suppose that we fix the subset of mines we would like to use to destroy the monsters. From there, to find the minimum cost needed to kill each monster, loop through all the positions of the mines used and find the closest one to the position of the monster. Suppose that the closest distance for monster i is $closest[i]$. Then for each monster, the cost needed is $min(h[i], closest[i])$.

Since k is small, we can brute force all possible subsets of mines and take the minimum over all cases.

Time complexity: $\mathcal{O}(2^k nk)$

Subtask 4

Additional constraints: $n, k \leq 3000$

Firstly, we sort the monsters and the mines in increasing order of position. Now suppose monster i is killed by mine a while monster j is killed by mine b.

We can observe the following property:

Theorem 1.1. *In an optimal selection of mines, if $i < j$, then $a \leq b$.*

Proof. We can prove this by exchange argument. Assume we have an optimal solution O, where monster i was killed by mine q while monster j was killed by mine p, where $i < j$ and $p > q$. Then the cost to move the monsters to the desired positions will be $|a[i] - x[q]| + |a[j] - x[p]|$.

We can show that it is never worse to transform O into an alternative solution O' where we swap mines to kill monsters i and j. In this case, the new cost to move the monsters will be $|a[i] - x[p]| + |a[j] - x[q]|$, in which $|a[i] - x[p]| + |a[j] - x[q]| \leq |a[i] - x[q]| + |a[j] - x[p]|$ (given $a[i] < a[j]$ and $x[p] < x[q]$).[1] Hence, swapping will never produce a worse result since the same cost is still needed to detonate the mines. □

With this, we can formulate a dynamic programming solution. Let $dp(i, j, flag)$ denote the minimum cost needed to kill the first i monsters using the first j mines, with $flag$ denoting whether the $j - th$ mine has already been used (so an additional cost of 1 will not need to be added if used again).

[1] See here for a detailed proof.

The transition is as follows:

$$dp(i,j,flag) = \begin{cases} 0, & \text{if } i = 0 \\ h[i] + dp(i-1,j,flag), & \text{if } j = 0 \\ min(h[i] + dp(i-1,j,flag), & \\ \quad dp(i,j,1) + |a[i] - x[j]| + (1 - flag), & \text{otherwise} \\ \quad dp(i,j-1,0)) & \end{cases}$$

To explain the otherwise portion, the first line represents killing monster i by reducing its health to 0, the second line represents killing monster i with mine j, while the last line represents skipping over mine j to consider other mines.

Time complexity: $\mathcal{O}(nk)$

Subtask 5

Additional constraints: $h[i] = 10^9$

This subtask means that we can ignore the health of each monster. Let us sort the mines in increasing position.

Then, visualise the monsters and mines on a coordinate line. Observe that:

Theorem 1.2. *Only the closest mines to each monster will be relevant in killing the monster.*

Proof. We will prove this by exchange argument. Suppose that monster i is killed by some mine p which is not the closest mine to monster i in an optimal solution O. If we consider this monster separately, the cost needed would be $|a[i] - x[p]| + 1$ if the mine is not used to kill any other monsters and $|a[i] - x[p]|$ otherwise. However, it will never be worse to swap to an alternative solution O' where monster i is killed by closest mine q instead, as the cost used will be $|a[i] - x[q]| + 1$ if mine q is only used to kill monster i and $|a[i] - x[q]|$ otherwise.

Considering that mine b is the closest mine, it holds that $|a[i] - x[q]| < |a[i] - x[p]|$, and hence $|a[i] - x[q]| + 1 \leq |a[i] - x[p]|$, which, in other words, means that the worst case of killing monster i with mine q will never result in greater cost than even the best case of using mine p.

Hence this concludes the proof that a closest mine should be used to kill the monster (if a mine is used), and hence this leaves each monster with at most 2 relevant mines (closest left, closest right), in the case where the closest left mine and the closest right mine are equidistant from the monster. □

With this observation, we first process the monsters with a single closest mine. For such a monster i with closest mine j, add $|a[i] - x[j]|$ to the answer. If mine j has not been used, add 1 to the answer and set mine j to be used.

This leaves us with the monsters with a closest left and closest right mine. If either the left or right mine has already been used, we can just send the monster to that mine and remove that monster from our consideration, since it will never be more optimal to send the monster to an unused mine over a used one.

Now, we are left with the monsters with an unused closest left and closest right mine. To handle these monsters, sort the monsters in increasing order of position. Then, we can try to pair up these monsters as much as possible. In particular, suppose monster i has closest left mine $x - 1$ and closest right mine x, while monster j has closest left mine x and closest right mine $x + 1$. Then we can pair monsters i and j together by destroying both of them with mine x, effectively saving 1 dollar.

Notice that since the monsters are all at distinct positions, the monsters paired up will be adjacent when sorted by position. This allows for a clean implementation.

Time complexity: $\mathcal{O}(n \log n)$

Subtask 6

Additional constraints: None

To solve the full problem, we need to include health back into the equation. Again, let $closest[i]$ denote the closest distance from monster i to a mine.

First, observe that:

Theorem 1.3. *If $h[i] \leq closest[i]$, it is always optimal to kill monster i by reducing its health to 0.*

Proof. We will prove this by exchange argument. Suppose there is an optimal solution O where monster i is killed by a mine, and $h[i] \leq closest[i]$. Separating monster i, the cost to kill monster i will be $closest[i] + 1$ if the mine is only used to kill monster i and $closest[i]$ otherwise. Whereas, the cost to kill monster i using health will be $h[i]$. We can observe that it is never worse to change to an alternative solution O' which uses health to kill monster i since $h[i] \leq min(closest[i], closest[i] + 1)$ which means that the health method is never worse in any case. □

Next, observe that:

Theorem 1.4. *If $h[i] > closest[i]$, it is never optimal to kill monster i by reducing its health to 0.*

Proof. We will again prove by exchange argument. Suppose there is an optimal solution O where monster i is killed by health and $h[i] > closest[i]$. Considering monster i separately, if it were killed by using a mine instead, the cost to kill monster i will be $closest[i] + 1$ if the mine is only used to kill monster i and $closest[i]$ otherwise. Hence it is never worse to swap to an alternative solution O' where monster i is killed using a mine instead since $max(closest[i], closest[i] + 1) \leq h[i]$ (i. e. it is at least as good or better to use the mine regardless of whether it has been used). □

Hence, with these 2 observations, we can effectively remove health from the equation. We first kill the monsters which should be killed by reducing health, then the remaining monsters can be dealt with using the solution in subtask 5.

Time complexity: $\mathcal{O}(n \log n)$

Task 2: `Thumper`

Authored by: Benson Lin Zhan Li

Prepared by: Ryan Goh Choon Aik

Editorial written by: Benson Lin Zhan Li, Ryan Goh Choon Aik

Subtask 1

Additional constraints: $n, m \leq 2000$

Consider two rabbits, one that is thumping and one that is not. We can simply use some if-else statements to determine the direction that the non-thumping rabbit will move (there are 8 cases).

Now, for each thump, we can iterate over every rabbit other than the rabbit that is executing the thump. Then, we can compute where they go and update their row and column numbers in $\mathcal{O}(1)$ per rabbit. This gives us a $\mathcal{O}(n)$ computation per thump. There are m thumps, so this solution has an overall time complexity of $\mathcal{O}(nm)$.

Time complexity: $\mathcal{O}(nm)$

Subtask 2

Additional constraints: $r[i] = 1$

The given constraint implies that all rabbits lie on a horizontal line. This means that rabbits will only move along the line away from the rabbit that is thumping, and not move in the vertical direction. We can make the following observation:

Lemma 2.1. *The order of the rabbits in the line does not change.*

Proof. We consider only the rabbits to the left of the thumping rabbit. These rabbits will move exactly 2 squares to the left. Since no rabbit ever overtakes another rabbit as they all move in one direction away from the thumping rabbit with the same distance moved, the order remains the same. The case for the rabbits to the right is also similar. □

Hence, it suffices to count the number of thumps to the left and right of a rabbit to determine its final location. This can be done first by sorting the rabbits by column in $\mathcal{O}(n \log n)$ time. Then, we use a frequency array to store the number of thumps the i-th rabbit executed in $\mathcal{O}(m)$ time.

Using prefix and/or suffix sums, we can then count the number of thumps to the left and right of a rabbit in $\mathcal{O}(1)$ time. We then update the coordinates of the rabbit accordingly.

Hence, the total time complexity is $\mathcal{O}(n \log n + m)$.

Time complexity: $\mathcal{O}(n \log n + m)$

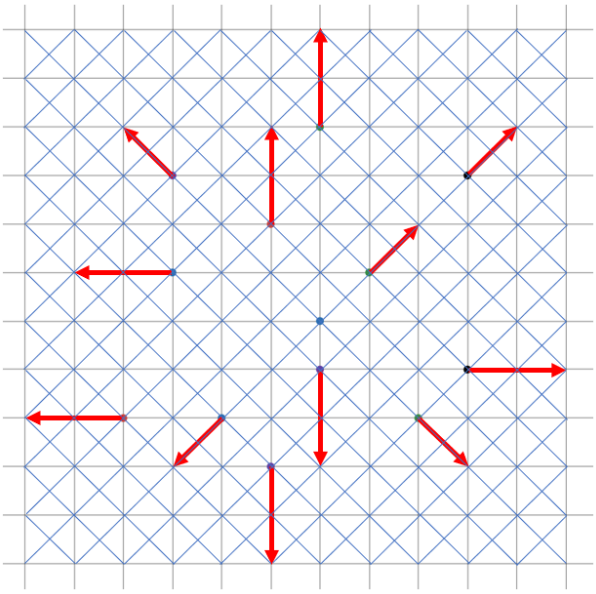
Subtask 3

Additional constraints: $n \leq 2000$

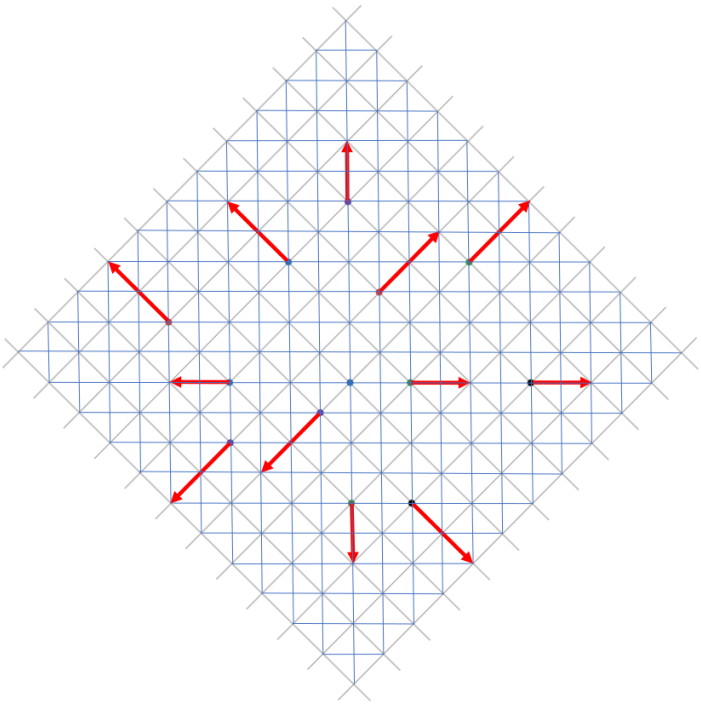
From Subtask 2, we require the following important observation:

Theorem 2.2. *Consider any two rabbits A and B. Define the orientation of B from A to be the direction that B will move in when A thumps. Then, after any sequence of thumps by any of the rabbits (including possibly A and B), the orientation of B from A remains unchanged.*

Proof. We will motivate this observation using sample test case 2. Firstly, we represent the positions of the rabbits as dots on the lattices of a coordinate grid. (A lattice point is a point on the grid with integer coordinates.) We will also draw in all the diagonals at a 45-degree angle on the coordinate grid (this will be useful later on). Let a *UR-diagonal* be a diagonal that slopes up-right (from bottom left to top right). Similarly, let a *DR-diagonal* be a diagonal that slopes down-right (from top left to bottom right). Refer to the following coordinate grid representing the rabbits in sample test case 2, with the diagonals drawn.



Now, we will rotate the entire graph by 45 degrees clockwise. Notice that this creates a new coordinate grid if we consider the points where any two lines intersect as a lattice point.



With this, we can make a few observations:

- Any rabbit to the left of the thumping rabbit in the new grid will move 2 lattice points to the left. In other words, if this rabbit is originally at (r, c) on the new grid, it will be now at $(r - 2, c)$.
- Any rabbit to the right of the thumping rabbit in the new grid will move 2 lattice points to the right. In other words, if this rabbit is originally at (r, c) on the new grid, it will be now at $(r + 2, c)$.

The same applies in the vertical (up-down) directions. Do note that movement in both directions in a single thump is possible, which explains the diagonal movements on the new grid.

Now, consider only the x-coordinates of all the rabbits on the new grid (a similar argument applies for the y-coordinates). From Lemma 2.1, the relative order of the rabbits in the x-direction will remain the same.

Since the relative order of the rabbits in both directions remains the same, the orientation between any two rabbits will remain the same after all the thumping. □

Implementation-wise, we can first compute how many times each rabbit thumps. This takes $\mathcal{O}(m)$ time. Then by Theorem 2.2, we now only need to care about the initial orientation between any pair of rabbits to figure out how the thumps of one rabbit will affect the other. This can be computed in $\mathcal{O}(n^2)$ by iterating over all pairs of rabbits. After that, we can compute the contribution of every other rabbit to each rabbit's position to obtain the final position of each rabbit.

Time complexity: $\mathcal{O}(n^2 + m)$

Subtasks 4 and 5

Additional constraints: $n \leq 100\,000$ (Subtask 4), None (Subtask 5)

We will need to speed up the counting of the thumps in all 4 directions for each rabbit.

To motivate the solution, let us define more terminology.

Again, we will represent the rabbits as points on a coordinate grid, like in Subtask 3. Denote (r, c) be a point on the original coordinate grid. We say that a point (r, c) lies on the k-th UR-diagonal if $r + c = k$. Similarly, we say that (r, c) lies on the k-th DR-diagonal if $r - c = k$ (note that k can be negative here). In other words, a point (r, c) on the coordinate grid can be represented by $(r + c, r - c)$ on the new grid formed when the original grid is rotated by 45 degrees. Moreover, a point (x, y) on the rotated grid (ie. the point lies on the x-th UR-diagonal and the y-th DR diagonal) can be transformed into its original coordinate through $(\frac{x+y}{2}, \frac{x-y}{2})$.

Now, if a rabbit is initially on UR-diagonal s, then the number of times its UR-diagonal number increases is the number of thumps from rabbits whose initial UR diagonal number is lower that s. Similarly, the number of times its UR-diagonal decreases is the number of thumps from rabbits whose initial UR-diagonal number is higher than s. The same logic applies for the DR-diagonals.

Again, we first compute how many thumps each rabbit performs. We then sort them by DR and UR diagonal numbers and use prefix and suffix sums on the number of thumps to efficiently compute how the diagonal numbers of each rabbit changes, in both directions. This algorithm is similar to the one in Subtask 2. The sorting takes $\mathcal{O}(n \log n)$ time, so the whole algorithm's time complexity is $\mathcal{O}(n \log n + m)$.

Subtask 4 is for solutions that is $\mathcal{O}(n \log 10^9 + m)$ or $\mathcal{O}(n \log n + m)$ with high constant factors. This may be due to the usage of lazy propagation / lazy node segment trees, `std::map`, or `std::unordered_map`.

Time complexity: $\mathcal{O}(n \log n + m)$

Task 3: `Reachability`

Authored by: Chua Wee Chong

Prepared by: Chua Wee Chong

Editorial written by: Chua Wee Chong

Subtask 1

Additional constraints: $n \leq 8$

For each road, there are 4 possibilities of directed-ness. Since n is very small, we try all 4^n possible renovation plans. To check whether the renovation plan is valid, we can run a naive $O(n)$ search from each node to see how many nodes it can reach. Therefore, we can check each possible renovation plan in $O(n^2)$.

Time complexity: $\mathcal{O}(4^n n^2)$

Subtask 2

Additional constraints: $n \leq 15$

Theorem 3.1. *For two cities (u, v), if $l[u] > l[v]$, then the road must be either one-way from u to v or closed.*

Proof. Consider otherwise. If it is possible to travel from city v to city u, then all cities reachable from city u must also be reachable from city v. That is, $l[u] \leq l[v]$ This contradicts the condition that $l[u] > l[v]$. Hence, the directed-ness of the road must be chosen such that city v cannot reach city u, which only being one-way from city u to city v or being closed fulfills. □

Theorem 3.2. *For two cities (u, v), if $l[u] = l[v]$, then the road must be either two-way from u to v or closed.*

Proof. Consider otherwise. Without loss of generality, suppose the road is one-way only from city u to city v. Then, the cities reachable from city v must a proper subset of the cities that are reachable from u. This is because city u is not reachable from city v, but city v and all cities reachable from it must be reachable from city u. Hence, this would imply that $l[u] > l[v]$. This contradicts the condition that $l[u] = l[v]$. □

With the above two claims, for each road, there are only two logical choices. The other two choices would lead to the renovation plan definitely being invalid. Hence, we only need to try all 2^n renovation plans. To check whether a renovation plan is valid, we can run a check similar to what was suggested in Subtask 1. These observations will be used in the final solution

Time complexity: $\mathcal{O}(2^n n^2)$

Subtask 3

Additional constraints: $l[1] = l[2] = \ldots = l[n]$

In this subtask, we only make use of two-way roads. Consider a leaf u and its parent p. Define a component to be a connected sub-tree of cities. For each city, we can maintain a count of its component size. Initially, all component sizes are 1.

Now, if city u has less than $l[1]$ cities in its component, it must be in the same component as p, as there are no other options. Hence, we can remove u, adding the component size of u to p.

If city u has exactly $l[1]$ cities in its component, it must not be in the same component as p and satisfies the condition for all cities in the component. Hence, we can remove u, without adding to p.

If city u has more than $l[1]$ cities in its component, there cannot be a valid solution.

We perform the above three checks recursively on every leaf, removing checked leaves and adding new leaves created due to the removals, until the tree is empty. We can handle the removals using a set data structure and do the checks in constant time for each city.

This is a classic task.

Time complexity: $\mathcal{O}(n \log n)$

Subtask 4

Additional constraints: If a plan exists, at least one such plan has no two-way roads.

From claims in Subtask 2, for each road, it must either be one-way or closed. This problem reduces to the following:

For each city u, does there exist a set of neighbouring cities such that their sum is equal to $l[u] - 1$? This is the classical knapsack problem, which we can solve individually for each node.

The time complexity of the knapsack problem is $O(nk)$ where n is the number of items and k is the knapsack size. Since our sum of number of items (cities) is bounded by $2n$ and our knapsack size (max value in l) is at most n, therefore our total time complexity is $O(n^2)$

Time complexity: $\mathcal{O}(n^2)$

Subtask 5

Additional constraints: $n \leq 400$

Root the tree arbitrarily, say at city 1. Define $dp(root, num\_reach)$ as whether it is possible to fulfill all l in the sub-tree rooted at $root$ and have $root$ be able to reach num_reach cities.

For every city, initially, only $dp(root, 1)$ is true, as every city can reach itself. Our final answer will be whether $dp(1, l[1])$ is true.

To fulfill sub-task 4, we need to convert the definitions. Let p be a parent city and u be a child city.

Case 1: $l[p] > l[u]$
If $dp(u, l[u])$ is false, then all values of $dp(p, num\_reach)$ must be set to false. This is because there would be no way to fulfill the sub-tree rooted at p.

Otherwise, if $dp(u, l[u])$ is true, then we can convolute it, in a knapsack style. Specifically, the transition is $dp(p, num\_reach) = dp(p, num\_reach - l[u]) \mid dp(p, num\_reach)$.

Case 2: $l[p] < l[u]$
If both $dp(u, l[u])$ (the road is closed) and $dp(u, l[u] - l[p])$ (one-way from city u to city p are false, then all values of $dp(p, num\_reach)$ must be set to false. This is because there would be no way to fulfill the sub-tree rooted at p.

Case 3: $l[p] = l[u]$ **and** $dp(u, l[u])$ **is false**
Since $dp(u, l[u])$ is false, therefore the road from city u to city v must be two-way. The transition is $dp(p, k) = dp(p, i)$ & $dp(u, j)$ for all $i + j = k$ and $1 \leq k \leq n$

Case 4: $l[p] = l[u]$ **and** $dp(u, l[u])$ **is true**
This is similar to case 3. However, now we also have the additional choice of making the road from city u to city v closed as $dp(u, l[u])$ is true.
The transition is $dp(p, k) = dp(p, k) \mid [dp(p, i)$ & $dp(u, j)]$ for all $i + j = k$ and $1 \leq k \leq n$.

The time complexity of this solution is $\mathcal{O}(n^3)$. Let c be the number of child cities. For cases 1 and 2, the time complexity is $\mathcal{O}(cn)$ for each city. For cases 3 and 4, due to the condition of $i + j = k$, each node would take at most $\mathcal{O}(cn^2)$ time. Summing this over all cities, the total

running time would be $\mathcal{O}(n^3)$, supposing all instances were cases 3 or 4.

Time complexity: $\mathcal{O}(n^3)$

Subtask 6

Additional constraints: None

Our current bottleneck is in cases 3 and 4, where we have to consider all $i + j = k$. It turns out that if we limit j to be the sum of sub-trees of all child cities processed so far every time we perform a new transition, the total time complexity of cases 3 and 4 over the whole tree is $\mathcal{O}(n^2)$.

This is due to a well-known optimisation known as Distribution DP or square-order tree DP.

Time complexity: $\mathcal{O}(n^2)$

Task 4: `Robots`

Authored by: Yaw Chur Zhe, Ling Yan Hao

Prepared by: Ling Yan Hao

Editorial written by: Ng Yu Peng, Yaw Chur Zhe

Subtask 1

Additional constraints: $h, w \leq 16, q \leq 20$

For each query, consider all 2^w possible colourings of the special cells. For each colouring, simulate each robot's movement from the start to the end. Count the number of distinct positions that the robots end up and take the minimum over all colourings.

Time complexity: $\mathcal{O}(2^w(h+w)q)$

Subtask 2

Additional constraints: $a[i] + 1 = b[i]$ for all queries

Since there are exactly two robots per query, we need only determine whether the answer to a query is 1 or 2. The answer to a query is 1 if and only if there exists a special cell in row $a[i]$ or row $b[i]$. Checking this condition can be done in $\mathcal{O}(1)$ with a Boolean array of size h.

Time complexity: $\mathcal{O}(h+w+q)$

Subtask 3

Additional constraints: $x[1] < x[2] < \cdots < x[w]$

For each $a \leq c \leq b$, let row c be an empty row if there does not exist a special cell in row c. Each empty row will certainly contribute to a distinct location, since the robot starting in row c will never move to other rows.

Then, for other rows with a special cell, we can colour the cell blue to move the robots downward. This ensures that robots starting on rows with a special cell will end on a row without a special cell, due to the constraints of the subtask.

However, we need to take note of an additional case. If rows b and $b-1$ both have a special cell, these 2 rows will still result in an additional final cell. This is because there is no empty row to flush the robots down towards. On the other hand, if row $b-1$ was empty while row b has a special cell, the cell on row b could be coloured red so the robot on row b goes towards row $b-1$ instead.

Let the number of empty rows in $[a, b]$ be x. With this, the answer for each query is $x+1$ in the case above and x otherwise. x can be maintained with prefix sums.

Time complexity: $O(h+w+q)$

Subtask 4

Additional constraints: $h, w, q \leq 5000$

Let us think about how to solve a single query efficiently. To do this, we first determine some properties about the robots that end up at the same point. We will also refer to robot i as the robot starting in row i for convenience.

We define some values:

- $endpos_i$ is the ending row of robot i, **in a certain configuration.**
- l_i is the highest (smallest number) ending row robot i can get to, **over all configurations.**
- r_i is the lowest (largest number) ending row robot i can get to, **over all configurations.**

We now state some claims, which we prove in the **Appendix**:

Claim 1: For all i, $l_i \leq i \leq r_i$.

Claim 2: If $i < j$, then $endpos_i \leq endpos_j$.

Claim 3: For any ending row, the starting rows of the robots ending in that row form a contiguous range.

At this point, we see that for a query on $a < b$, we may partition the rows from a to b into intervals $[a_1, b_1], [a_2, b_2], \ldots, [a_k, b_k]$ such that $a_1 = a, b_k = b, a_i \leq b_i$, and $a_{i+1} = b_i + 1$ for each $1 \leq i \leq k-1$, and for each interval $[a_i, b_i]$, all robots starting in a row j with $a_i \leq j \leq b_i$ end up in the same row. Therefore, to achieve the minimum number of distinct cells the robots occupy, we will want to minimise the number of intervals in this partition. This leads us to determine when we can make all robots with indices in some interval $[a_i, b_i]$ end up in the same row, which brings us to the following claim:

Claim 4: Given $a \leq b$, we can make all robots i with $a \leq i \leq b$ end in the same position if and only if $r_a \geq l_b$.

In the **Proof of Claim 4**, notice that if we can make all robots starting in rows from a to b end up in the same row, we can do so by only choosing the colours of the special cells in rows from a to b, and these robots stay within the range of rows from a to b. Therefore, if we can partition $[a, b]$ into the intervals $[a_1, b_1], [a_2, b_2], \ldots, [a_k, b_k]$ such that for each $1 \leq i \leq k$, it is possible to make robots a_i to b_i end up in the same row, then it is possible to make robots from a_i to b_i end up in the same row for all i **simultaneously**.

With that out of the way, we try to build these intervals greedily: given query on the range $[a, b]$, pick the largest $b_1 \leq b$ such that $r_a \geq l_{b_1}$, and take the interval $[a, b_1]$ in the solution. If $b_1 = b$ then we are done, otherwise we continue building the solution for the range $[b_1 + 1, b]$ in the same way. We prove this greedy algorithm is optimal in **Proof of Greedy Algorithm**.

How do we implement this greedy algorithm efficiently? Firstly, we may precompute the values of l_i, r_i for all i with DP. We iterate on the special cells from column w to column 1. To compute l_i, r_i for the special cell in column i, suppose the special cell is in row x, then we refer to the l value of the first special cell in row $x - 1$ to the right of this special cell (if any), and refer to the r value of the first special cell in row $x + 1$ to the right of this special cell (if any).

Next, we can utilise the following property to implement our greedy algorithm:

Claim 5: If $i < j$, $l_i \leq l_j, r_i \leq r_j$.

What **Claim 5** implies is that to find the largest $b_1 \leq b$ such that $r_a \geq l_b$, we may just iterate on i from a to b until $l_i > r_a$, and take $b_1 = i - 1$ when this happens.

The precomputation can be done in $\mathcal{O}(w \log w)$ or $\mathcal{O}(w)$ and the greedy algorithm can be implemented in $\mathcal{O}(h)$ per query, giving an overall time complexity of $\mathcal{O}(w \log w + hq)$ or $\mathcal{O}(w + hq)$ which is fast enough.

Time complexity: $\mathcal{O}(w \log w + hq)$ or $\mathcal{O}(w + hq)$

Subtask 5

Additional constraints: None

For the full solution, we must speed up the simulation of the greedy algorithm. For each i, let $j \geq i$ be the maximum row index such that $l_j \leq r_i$, and let $next_i = j + 1$. We can compute the values of $next$ with two pointers due to **Claim 5**, and to simulate the greedy algorithm for a query $a \leq b$, repeatedly replace a with $next_a$ until $a > b$.

Perform 2^k decomposition (or "binary lifting") on the $next$ array to determine how many "jumps" are needed to exceed b, to obtain a time complexity of $\mathcal{O}(\log h)$ per query.

Time complexity: $\mathcal{O}(w \log w + q \log h)$ or $\mathcal{O}(w + q \log h)$

Appendix

Proof of Claim 1

If we colour all special cells red, robots always move upwards on them. Thus robot i will end up at some row $a \leq i$, so $l_i \leq i$. Showing $r_i \geq i$ is similar.

Proof of Claim 2

Let $row_{i,k}$ be the row robot i is at, **when leaving column** k. Suppose for the sake of contradiction that for some $i < j$, $endpos_i > endpos_j$. Consider the **first** column k such that $row_{i,k} \geq row_{j,k}$. Then $row_{i,k-1} < row_{j,k-1}$. Since column k has at most one special square, and robots i and j enter column k in different rows, at most one of $row_{i,k}, row_{j,k}$ can differ from $row_{i,k-1}, row_{j,k-1}$, respectively, and if they differ, they differ by at most 1. Since $row_{i,k-1} < row_{j,k-1}$, we cannot have $row_{i,k} > row_{j,k}$, and since we assumed $row_{i,k} \geq row_{j,k}$, we must have $row_{i,k} = row_{j,k}$. It is easy to see that $row_{i,k'} = row_{j,k'}$ for all $k' \geq k$, which means $endpos_i = endpos_j$, a contradiction.

Proof of Claim 3

For some row x, let a, b, $a \leq b$, be the minimum and maximum indices of robots that end in row x. For each $a \leq i \leq b$, we conclude from **Claim 2** that $endpos_a \leq endpos_i \leq endpos_b = endpos_a$, which forces equality throughout.

Proof of Claim 4

Suppose all robots i with $a \leq i \leq b$ end in the same row, say row x. By definition, $r_a \geq endpos_a = x = endpos_b \geq l_b$, so $r_a \geq l_b$.

Now suppose that $r_a \geq l_b$. We propose an assignment of colours to the special squares so that $endpos_a = endpos_b$. The case when $a = b$ is trivial, so suppose $a < b$.

In the configuration where we colour all special cells blue, let the special cells that robot a passes through be $(a, i_a), (a+1, i_{a+1}), \ldots, (r_a - 1, i_{r_a-1})$ (it only passes through one special cell in each row, since it always goes down when it gets to a special cell). Let this set of cells be A.

Similarly, in the configuration where we colour all special cells red, let the special cells that robot b passes through be $(b, j_b), (b-1, j_{b-1}), \ldots, (l_b + 1, j_{l_b+1})$. Let this set of cells be B.

Note that if we colour all special cells in A blue, then regardless of what the colours of other special cells are, robot a will pass through all the special cells in A, and only the special cells in A, ending in row r_a. Similarly, by colouring all special cells in B red, robot b passes through all the special cells in B and only the special cells in B, ending in row l_b.

If A and B have no common cells, then by colouring all cells in A blue and all cells in B red, robot a ends up in row r_a and robot b in row l_b. If $r_a = l_b$, we are done. Otherwise, we know $r_a \geq l_b$, so we have $r_a > l_b$. But then $endpos_a = r_a > l_b = endpos_b$, a contradiction to $endpos_a \leq endpos_b$!

On the other hand, if A and B have some common cell, say $(k, i_k) = (k, j_k)$, then we may colour the cells $(a, i_a), (a+1, i_{a+1}), \ldots, (k-1, i_{k-1})$ blue, and $(k+1, j_{k+1}), \ldots, (b, j_b)$ red, so that both robots a and b will reach the special cell $(k, i_k) = (k, j_k)$. Their paths after reaching that cell must be identical, and thus they will end up in the same ending row.

Thus in all cases, we have a configuration that gives us $endpos_a = endpos_b$. Using **Claim 2** again, for each $a \leq i \leq b$, we have $endpos_a \leq endpos_i \leq endpos_b$, and since $endpos_a = endpos_b$ equality holds everywhere.

Proof of Greedy Algorithm

Let $ans_{i,j}$ be the minimum number of distinct final cells the robots from i to j can occupy, which we define as 0 if $i > j$. Clearly if $i_1 \leq i_2$, then $ans_{i_1,j} \geq ans_{i_2,j}$. Let $[a, b_1']$ be the interval that a is in inside the optimal solution. By definition of b_1, $b_1' \leq b_1$. Then $ans_{i,j} = 1 + ans_{b_1'+1,b} \geq 1 + ans_{b_1,b}$, which shows that by taking the interval $[a, b_1]$ along with the optimal solution for the range $[b_1 + 1, b]$, we still have an optimal solution for the range $[a, b]$.

Proof of Claim 5

We will prove $l_i \leq l_j$, as $r_i \geq r_j$ is similar. Consider a configuration that lets robot j reach row l_j as the ending row. Using the same notation as in the **proof of Claim 2**, if we ever have $row_{i,k} = row_{j,k}$ then they will end up in the same row, which shows $l_i \leq l_j$ since robot j ends up in row l_j. Otherwise, since $row_{i,0} = i < j = row_{j,0}$, following the same logic as in the

proof of Claim 2 we can show that we can never have $row_{i,k} > row_{j,k}$. Then $row_{i,k} \leq row_{j,k}$ always, so robot i ends up in some row $\leq l_j$, so $l_i \leq l_j$ as desired.

Task 5: `Flooding`

Authored by: Yaw Chur Zhe

Prepared by: Yaw Chur Zhe

Editorial written by: Yaw Chur Zhe

In this editorial, let $s((r, c))$ be the set of cells that would be flooded if (r, c) were the initially flooded cell. In particular, $f((r, c)) = |s((r, c))|$.

Assume for convenience that the grid is surrounded by buildings (if not, pad the grid with buildings).

Subtask 1

Additional constraints: $h = 1$

It is easy to see that $s((1, c))$ forms a maximal range that includes $(1, c)$. By iterating through the grid from left to right and maintaining the length of the maximal range, it is possible to solve this subtask in linear time.

Time complexity: $\mathcal{O}(w)$

Subtask 2

Additional constraints: $h, w \leq 80$

It is possible to simulate the flooding process with a Breadth-First Search (BFS). To compute $f((r, c))$:

1. Push cell (r, c) into a queue.
2. Repeatedly, pop the front cell (a, b) from the queue, and if there exists a cell adjacent to (a, b) that becomes flooded. If so, add that cell to the queue.

Since this BFS runs in $\mathcal{O}(hw)$, performing this BFS for all empty cells (r, c) yields a $\mathcal{O}(h^2w^2)$ solution.

Time complexity: $\mathcal{O}(h^2w^2)$

Subtask 3

Additional constraints: $h, w \leq 500$

A crucial observation is the following:

Theorem 5.1. *For all (r, c), $s((r, c))$ forms a subrectangle of the grid.*

Proof. One of the many ways to prove this theorem is via a constructive algorithm:

1. Initially, let $s((r, c)) := \{(r, c)\}$.
2. Repeat the following steps:
    - If $s((r, c))$ is surrounded by buildings on all sides, then $s((r, c))$ cannot possibly expand further. Break out of this loop.
    - Otherwise, there exists an empty cell adjacent to $s((r, c))$, which becomes flooded. Assume without loss of generality that this cell is on the top edge of $s((r, c))$. Water flows into this cell, and then flows into the row of that cell. The overall effect is that $s((r, c))$ expands upwards by one row.

Since $s((r, c))$ is a subrectangle of the grid at all times in this algorithm, the theorem is proven. □

The constructive algorithm can be used to compute $f((r, c))$. With appropriate preprocessing, it is possible to perform the check in step 2 in $\mathcal{O}(1)$ time (using a prefix sum, for instance). In this manner, computing $f((r, c))$ can be done in $\mathcal{O}(h + w)$, since $s((r, c))$ can only expand h times up or down and w times left or right.

By repeating this for all empty cells, we obtain a $\mathcal{O}(hw(h + w))$ solution.

Time complexity: $\mathcal{O}(hw(h + w))$

Subtask 4

Additional constraints: $h, w \leq 2000$

From this point onwards, we require further observations concerning the structure of $s((r, c))$.

Lemma 5.2. *For all rows and columns of $s((r, c))$, there must exist at least one empty cell.*

Proof. Assume otherwise, i.e. there exists some row or column that consists completely of buildings. It is not possible for water to flow past this row or column, hence contradicting the initial assumption. □

Corollary 5.3. *$s((r, c))$ is surrounded completely by buildings.*

Proof. This follows directly from the constructive algorithm provided in the proof of Theorem 5.1. □

Call a subrectangle *good* if it satisfies the properties in Lemma 5.2 and Corollary 5.3 (even if the subrectangle is not equal to $s((r, c))$ for any (r, c)). It turns out that it is possible to generate all such good subrectangles quickly (**Phase 1**). Then, all that remains is to determine which good subrectangles correspond to which cells (**Phase 2**).

Phase 1

Fix the top row t of the good subrectangle. Let $i_1, i_2, \ldots, i_k$ be the indices of columns where (t, i_j) has a building. Consider a good subrectangle with left column $l = i_j + 1$ and right column $r = i_{j+1} - 1$ (if $l > r$, ignore the following). We can uniquely determine its bottom row b (if it exists) due to Lemma 5.2. Specifically, b is equal to the minimum row where cells $(b + 1, l), (b + 1, l + 1), \ldots, (b + 1, r)$ all have buildings, provided $b \geq t$. Computing b can be done efficiently with a binary search. You must additionally ensure that the top, left, and right borders of this good subrectangle are surrounded by buildings, as per Corollary 5.3.

Now, we have identified all good subrectangles with left and right columns adjacent in the sequence $i_1, i_2, \ldots, i_k$. Consider the index i_j where the number of contiguous buildings directly below (t, i_j) is minimised. In other words, the distance from (t, i_j) to the nearest empty cell below it is minimised. Observe that there are no other good rectangles where i_j is the left or right column (under the fixed top row t), due to Lemma 5.2. Hence, we can safely remove i_j and try to identify a good subrectangle with $l = i_{j-1}$ and $r = i_{j+1}$. By repeatedly removing the index with the minimum number of contiguous buildings directly below it, we can identify all good subrectangles for a fixed top row t. It is also obvious that $\mathcal{O}(w)$ good subrectangles can be identified in this manner, since $\mathcal{O}(w)$ good subrectangles are identified initially, and additional good subrectangles will only be identified if an element from $i_1, i_2, \ldots, i_k$ is removed.

By repeating the above process for all top rows t, we are able to identify all good subrectangles. Crucially (and surprisingly), there are $\mathcal{O}(hw)$ of these good subrectangles.

Phase 2

To match good subrectangles to their corresponding cells, we observe the following fact:

Lemma 5.4. *$s((r, c))$ is equal to the smallest good subrectangle containing (r, c).*[2]

Proof. Consider all the good subrectangles that contain cell (r, c). Let X be the intersection of all of these subrectangles (X cannot be empty since it must contain the cell (r, c) itself). One can verify that X is also a good subrectangle:

1. The intersection of many subrectangles that are surrounded by buildings will also yield a subrectangle surrounded by buildings.
2. It is not possible for X to contain a row or column filled with buildings, since it would imply the existence of a smaller good subrectangle that is a subrectangle of X (by "cutting" the rectangle along the filled row or column).

It is not possible for water to flow out of X if (r, c) were initially flooded (since X is surrounded by buildings). Hence $s((r, c)) \subseteq X$. However, since X is the intersection of all good subrectangles containing (r, c), $X \subseteq s((r, c))$. Hence $s((r, c)) = X$. Further observe that X is exactly equal to the smallest good subrectangle containing (r, c), hence proving the lemma. □

With the knowledge of Lemma 5.4, a straightforward way to complete the solution would be to use standard sweep-line techniques and a segment tree to simulate several 2D range-set operations. This yields a time complexity of $\mathcal{O}(hw \log h \log w)$, which is sufficient to pass this subtask.

Time complexity: $\mathcal{O}(hw \log h \log w)$

Subtask 5

Additional constraints: None

To obtain the full solution, **Phase 2** of the algorithm must be sped up. At least two distinct ways to achieve this have been identified.

Method 1

This method relies on the following observation:

Lemma 5.5. *When identifying good subrectangles in Phase 1 in **decreasing** t, the smallest good subrectangle containing (r, c) is equal to the first good subrectangle identified containing (r, c).*

[2]Note that there must exist at least one good subrectangle containing (r, c), since (r, c) is empty.

Proof. Omitted. □

In other words, we may perform **Phase 1** and **Phase 2** in parallel: once a good subrectangle is identified, we can immediately determine the answer for all cells in that subrectangle.

Sweep t from h to 1. Maintain a segment tree of size w indexed by columns, where the value of the segment tree at a particular column is equal to the minimum row of the cell in that column whose answer has not been determined yet, with the additional constraint that the row must be at least t. When a good subrectangle t, l, r, b has been identified, repeatedly do a range minimum query on columns l to r in this segment tree. If it yields a row $\leq b$, set the answer of that cell to $(b-t+1)(r-l+1)$ and delete it from the segment tree (then updating the value of the segment tree at that column). When transitioning from t to $t-1$, update all values in the segment tree to $t-1$. This is clearly amortised $\mathcal{O}(hw)$, since the answer for each cell can only be determined once.

Time complexity: $\mathcal{O}(hw(\log h + \log w))$

Method 2

The 2D range-set problem from **Phase 2** can be reframed as multiple independent 1D range-set problems (one for each row), which can then be solved with Union-Find Disjoint Sets. However, naively splitting each good subrectangle into its constituent rows may not be efficient enough. One possible optimisation is to "ignore" rows that have already been filled; maintain the maximum fully filled row from previously identified subrectangles, and only fill rows after that. There exists a proof showing that the number of 1D range-set operations generated by this process is $\mathcal{O}(hw)$.

Time complexity: $\mathcal{O}(hw(\log h + \log w))$

Author's Comments

I am aware of other solutions with unproven time complexities:

- Optimising the solution for Subtask 3 using a Union-Find Disjoint Set, eagerly "merging" subrectangles whenever possible. This solution appears to be empirically efficient, and is likely able to score 60 or 100 points with enough constant-time optimisation.
- Naively performing **Method 1**. In other words, instead of using a segment tree to determine if there exists a cell whose answer has yet to be determined, simply iterate from l to r one-by-one to clear all such cells. Such a solution comfortably scores 100 points. I

am not sure how to bound the complexity of this solution, but I conjecture that it is fast because the sum of widths (or heights) of good subrectangles is somehow bounded.

Proving that these solutions are efficient remains an open problem. I welcome further discussion on this topic.