Fall 2020 mBIT Advanced Editorial

November 14, 2020

This editorial provides the intended solutions to each problem as well as accepted programs in each supported language. In some cases, the given programs may employ different algorithms than the one described in the editorial. For more complex problems, multiple solutions may be given, in which case there will be programs for each solution. Nevertheless, problems are likely to have solutions which are not covered here and we would be interested to hear about any such solutions the reader may devise.

Contents

1 Climbing Trees 2

2 Stone Piles 3

3 Calendars 4

4 The Duplicator 5

5 Locked in the Past 6

6 Night of the Candles 7

7 Gemstones 8

8 The Flock of Rams 9

9 Textile Display 10

10 Tanya's Revenge 11

11 Sphinx Economics 12

12 Building Atlantis 14

§1 Climbing Trees

We can observe that if Joe can visit a tree with height j immediately after a tree with height i, he can also visit all trees with heights between i and j. Thus, we can construct an algorithm as follows:

First, we sort the trees by height. Starting at the lowest tree, we move upwards to the next tree and keep track of how many meters we have ascended. If the next tree is not reachable (or if we are at the tallest tree), store the height climbed. Then repeat until we reach the tallest tree in the forest, making sure to overwrite the stored height climbed if we just climbed a greater distance in the current section.

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

Problem: Claire Zhang
Flavortext: Evan Wang
Editorial: Evan Wang
Code: C++, Java, Python

§2 Stone Piles

There are many ways for Gabe to move the stones in linear time and output complexity. One method is as follows:

Represent each pile as a stack for easy retrieval. This requires that the stones for each pile be read backwards.

1. Move all stones to pile 1. This takes at most N moves.
2. While pile 1 is nonempty, if the top stone is labeled 1 or 2, move it to pile 2. Otherwise, move it to its final stack. This takes exactly N moves.
3. While pile 2 is nonempty, if the top stone is labeled 1, move it to pile 1, and if it is labeled 2, move it to pile 3. This takes at most N moves.
4. Move all the stones labeled 2 that are on top of stack 3 back to stack 2. This takes at most N moves.

Because each step uses no more than N moves, the total number of moves used is bounded by $4N$ or $4 \cdot 10^5$ for $N = 10^5$; the bound in the statement is set higher to discourage reverse-engineering of the solution. In practice, it is usually possible to use significantly fewer moves than $4N$.

Time complexity: $\mathcal{O}(N)$
Output Complexity: $\mathcal{O}(N)$

Problem: Gabriel Wu
Flavortext: Jeffrey Tong
Editorial: Claire Zhang
Code: C++, Java, Python

§3 Calendars

The brute-force approach of computing $dist(A, C)$ after rotating B $0, 1, 2, ...,$ and $N-1$ times to the right runs in $\mathcal{O}(N^2)$, which is too slow. However, if we are able to determine by how much $dist(A, C)$ changes after each rotation in constant time, we can reduce our algorithm's time complexity to $\mathcal{O}(N)$.

First, compute $dist$ at the beginning. Let Δ denote the change in $dist$ that will occur after we rotate C one more time to the right if we pretend that the last number simply moves forward one more space instead of wrapping around. Δ may be computed at the beginning by noting the relative positions of each number in A and B.

During each rotation, we first need to add $\Delta - 1$ to $dist$ to account for the movement of the first $N-1$ elements. Let y be the last element of C; we also add $(pos_A(y) - 0) - (N - 1 - pos_A(y)) = 2pos_A(y) - N + 1$ to $dist$ to account for the wrapping of y. Therefore, we add $\Delta + 2pos_A(y) - N$ in total to $dist$ for this rotation.

To compute the next Δ, we notice that it will change between rotations exactly when, for some $1 \leq x \leq N$, $pos_A(x) - pos_C(x)$ changes signs. This can occur in two cases: when $pos_A(x) = pos_C(x)$ and when $pos_C(x)$ wraps around from N to 1. Thus, if k is the number of x values for which $pos_A(x) = pos_C(x)$ after the rotation, Δ will change by $2k - 2$ (-2 since y's contribution to Δ will always switch from $+1$ to -1, except when $pos_A(y) = 1$, in which case the $2k$ immediately reverts it anyways).

Thus, if we precompute k for each rotation in a hashmap during precomputation and track the last number in C, we can recompute $dist$ after each rotation in constant time.

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

Problem: Maxwell Zhang
Flavortext: Jeffrey Tong
Editorial: Claire Zhang
Code: C++, Java, Python

§4 The Duplicator

We can approach this problem with complementary counting. We start by assuming all pairs $1 \leq i < j \leq N$ are valid, with $\binom{N}{2}$ pairs. Then we subtract from our starting count all pairs that do not fit the criteria. We iterate through array A, and keep track of how many of each element has appeared. When we move on to the next index j, we know that the number of invalid indices i that do not satisfy the conditions are the amount of times A_j has appeared previously in the array A (since we could choose any of the previous indices that A_j has appeared as our i index, which would make the condition invalid because $A_i = A_j$), so for each A_j we encounter, we subtract by how many times it has already appeared previously. We then repeat the process for array B.

However, in doing this process, we have counted cases in which $B_i = B_j$ and $A_i = A_j$ twice, so we iterate through both arrays and repeat the process above, but keeping track of how many of each *pair* has been encountered, and instead of subtracting how many times the pair $\langle A_i, B_i \rangle$ has appeared, we add how many times it has appeared, upon encountering the pair.

In practice, we can implement this with three maps (one for A, one for B, and one for the pairs) and iterating once through both arrays.

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

Problem: Timothy Qian
Flavortext: Evan Wang
Editorial: Evan Wang
Code: C++, Java, Python

§5 Locked in the Past

Call an *optimal* set of moves one that uses the minimal number of moves to reach $0, \ldots, 0$. First, we show that there exists an optimal set of moves such that a wheel will only be increased or decreased, never both. Let the total interval size of a set of moves with intervals $[a, b]$ be equivalent to the sum of $b - a$ for all increase or decrease moves $[a, b]$. Consider a wheel that has two moves that has an increase move of wheels $[a, b]$ and a decrease move of wheels $[c, d]$. The first case is if $[a, b]$ is contained within $[c, d]$. Then we can break this move into two nonoverlapping decrease moves. Note that the total interval size sum shrinks. If $[a, b]$ intersects $[c, d]$ such that $a \leq c \leq b \leq d$, then we can break the moves again similarly into two intervals $[a, c], [b, d]$, one with one increase move and one with a decrease move, strictly shrinking the total interval size. These intervals again don't overlap. Therefore, if there are overlapping increase and decrease moves, we can split them so they don't overlap with the same number of moves and strictly smaller total interval size. The total interval size can't decrease forever, so therefore at the end of this process, no two increase or decrease intervals will intersect.

Call an optimal set of moves *good* if it only increases or decreases each wheel. Having established this fact, we note there exists a good set of moves where each wheel rotates up or down by at most NK. To see why, we can obviously just rotate each wheel individually by at most NK to achieve this. Now wheels are called *up* wheels if we only increase them and *down* wheels if we only decrease them. Pooling together these observations, we can use dynamic programming to solve this problem. Let's say a wheel is currently at position x. if it is a down wheel, then we only need to consider rotating it down by $x, x + (K + 1), \ldots, x + (N - 1) \cdot (K + 1)$. Similarly, if it is an up wheel, we only need to consider increasing it by $(K + 1) - x, 2 \cdot (K + 1) - x, \ldots, N \cdot (K + 1) - x$ (with some care taken when x is 0 initially).

Let $u[i][j]$ be the values we must consider for increasing wheel i and $d[i][j]$ be the values for decreasing wheel j, such that $u[i][j] \leq u[i][j + 1], d[i][j] \leq d[i][j + 1]$ as described above. Thus, we maintain a dp state $dp[i][t][j]$, which designates the minimal number of moves to rotate wheels $1, \ldots, i$ into place such that wheel i is an up wheel is $t = 0$ and a down wheel if $t = 1$, and j represents how much we rotated it (the $\mathcal{O}(N)$ possible values for rotation). Now consider the case of computing $dp[i][0][j]$ only since the case of computing $dp[i][1][j]$ is analogous. If wheel $i - 1$ was a down wheel, then $dp[i][0][j] = dp[i - 1][1][k] + u[i][j]$. However, if wheel $i - 1$ was an up wheel, we have $dp[i][0][j] = dp[i - 1][0][k] + \max(0, u[i][j] - u[i - 1][k])$. This is because we can "use" wheel $i - 1$'s increases for wheel i's increases because they are both up wheels. In total, there are $\mathcal{O}(N^2)$ states with a $\mathcal{O}(N)$ transition, leading to an $\mathcal{O}(N^3)$ solution. However, one can notice that $u[i][j] > u[i - 1][k]$ if $j > k$. Thus, these transitions can be optimized to $\mathcal{O}(1)$ using suffix maximums. For more details, see our solution codes.

Time Complexity: $\mathcal{O}(N^2)$

Problem: Gabriel Wu
Flavortext: Timothy Qian
Editorial: Timothy Qian
Code: C++, Java, Python

§6 Night of the Candles

We keep track of four lists, which we label U, D, L, R. In each list, we store the indices of the lit locations that have an unlit candle if you proceed U, D, L, R by one candle respectively. Each time we process a breeze, we process all candles in the respective list for the direction of the breeze, lighting up the candles adjacent to each candle in the list in the corresponding direction. If the candle is already lit, we ignore it. Else, we process the newly lit candle, and check in each of its four adjacent directions to check which candles next to it are not lit, and add the newly candle to the corresponding lists. Overall, each candle gets processed at most 4 times, one for each direction. Therefore, the complexity of this is $\mathcal{O}(NM + B)$.

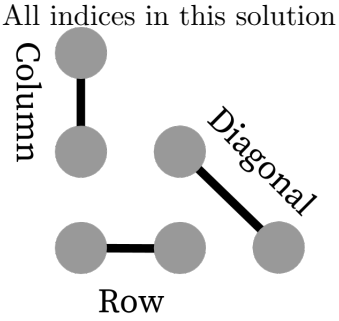
Time complexity : $\mathcal{O}(NM + B)$

Problem: Gabriel Wu
Flavortext: Maxwell Zhang
Editorial: Timothy Qian
Code: C++, Java, Python

§7 Gemstones

The maximum number of strings is $N-1$, attained exactly when one string of each length between 2 and N is used, which is always possible; thus, Maxwell must use all possible lengths. Notice that the string of length N must cover an entire edge of the triangle, leaving a subproblem with size $N-1$. Proceed with row-column DP.

If we rearrange the gems into a triangle with one side parallel to the x-axis and one to the y-axis, as shown, we can classify each string as being on a *row*, *column*, or *diagonal*. All indices in this solution start from 0.



Let $p(\text{“}row\text{”}, i, j, len)$ be the price of the string starting at gem j in row i with length len and define p likewise for “col” and “$diag$”. p can be calculated directly in $\mathcal{O}(N)$.

Define $f(r, c, len)$ as the maximum total price for the subtriangle with bottom-left corner at row r, column c, and side length len, and memoize f's results in an array dp. If

$$f(r,c,len) = \begin{cases} 0 & \text{if } len = 1 \\ dp[r][c][len] & \text{if } dp[r][c][len] \neq -1 \\ \max(& \\ \quad p(\text{“}row\text{”}, r, c, len) + f(r-1, c, len-1), & \\ \quad p(\text{“}col\text{”}, c, r-len+1, len) + f(r, c+1, len-1), & \text{otherwise} \\ \quad p(\text{“}diag\text{”}, r-len+1-c, c, len) + f(r, c, len-1) & \\ ) & \end{cases}$$

, $f(0,0,N)$ will be the final answer, and as f is computed on each DP state (no more than $N \cdot N \cdot N$) exactly once, the computation will be done with $\mathcal{O}(N^4)$ time and $\mathcal{O}(N^3)$ memory.

To solve subtask 2, we must reduce the time complexity to $\mathcal{O}(N^3)$ by computing p in constant time. Let $row[i][j][k]$ be the price of the string in row i starting from gem j with length k, and define col and $diag$ analogously. These arrays can each be computed in $\mathcal{O}(N^3)$ so that string prices can be accessed in $\mathcal{O}(1)$ in the transition.

To solve subtask 3, only precompute the number of gems of each color in the prefixes of each row, column, and diagonal so that p can be computed via prefix sums, reducing the time and memory complexity of preprocessing to $\mathcal{O}(N^2)$. A final optimization that will probably be necessary is to rewrite dp and f to only use the lengths currently needed in the transition (len and $len-1$), reducing the memory complexity of our DP to $\mathcal{O}(N^2)$.

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

Problem: Gabriel Wu
Flavortext: Jeffrey Tong
Editorial: Jeffrey Tong
Code: C++, Java, Python

§8 The Flock of Rams

There are two cases to consider.

1. Barusu enters through one cell on the border of the grid, takes two rams, then exits through that same border cell.
2. Barusu enters through one border cell, takes one ram, then exits. He then enters through a different border cell, takes another ram, and exits.

We define the distance between two cells as the minimum number of obstacles to break such that there would then exist an unobstructed path between the cells. To handle both cases, we precompute distances from each ram to each cell with K 0-1 BFSs, and precompute the minimal distance from each cell to the border with one multi-source 0-1 BFS. This step takes $\mathcal{O}(NMK)$.

Case 1:

An initial idea could be to consider pairs of rams and the following route:

$$\text{border cell} \to \text{ram 1} \to \text{ram 2} \to \text{border cell}$$

However, simply adding the distances between points is wrong, because parts of the route may overlap, causing us to double count obstacles. Instead, let's consider a *meetup cell*. The route is defined as follows:

$$\text{border cell} \to \text{meetup cell} \to \text{ram 1} \to \text{meetup cell} \to \text{ram 2} \to \text{meetup cell} \to \text{border cell}$$

Importantly, in an optimal route, there exists a meetup cell such that the three paths from the meetup cell to the border cell and two rams do not overlap with each other. If the paths did overlap, we could always shift the meetup cell along the overlapping section until the paths no longer overlap. It is possible for the meetup cell to be the border cell or either of the two rams.

Thus, we can simply iterate over all possible meetup cells and take the minimum of the sum of the distances of the three paths. To avoid $\mathcal{O}(NMK^2)$, we can precompute the closest two rams for each cell. Don't forget to subtract 2 if the meetup cell itself contains an obstacle, so that the obstacle isn't counted 3 times. Overall, this case is handled in $\mathcal{O}(NM)$ time. You may have to store only the closest two rams to each cell to fit the memory limit.

Case 2:

Simply iterate over all pairs of rams and take the pair with minimal sum of distances to the border. Note that we don't have to worry about the two paths from the rams to the borders overlapping, as that is already covered by case 1 (try to see why that's true). Overall, this case is handled in $\mathcal{O}(K^2)$ time.

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

Problem: Gabriel Wu
Flavortext: Timothy Qian
Editorial: Maxwell Zhang
Code: C++, Java, Python

§9 Textile Display

For ease of explanation, I will describe the process instead as an array of textiles with colors in some order, and we remove textiles one by one from left to right. Let's consider the contribution of each color to the answer. Color k has C_k textiles of that color. We want to characterize how much a certain color adds to the total happiness of a given ordering.

One can observe that only the last occurrence of a color matters, as a color contributes to the impression factor as long as it holds at least one occurrence. Since textiles are removed from the left, if the last occurrence is at index i (one-indexing), then it contributes $+i$ to the total happiness to the ordering. Specifically, it contributes $+1$ to the impression factor after $0, 1, \ldots, i-1$ textiles have been removed.

Now let's ask a different question: for how many orderings does the color k contribute $+1$ to the impression factor after removing $i-1$ textiles? Well, the only cases where it doesn't contribute $+1$ is if the last occurrence (or equivalently, all occurrences) of the color are before index i. There are $\binom{i-1}{C_k} \cdot C_k! \cdot (N-C_k)!$ possible ways to do that (we multiply by $C_k!$ and $(N-C_k)!$ because the textiles are distinguishable). The number of orderings where color k does contribute is thus $\left[\binom{N}{C_k} - \binom{i-1}{C_k}\right] \cdot C_k! \cdot (N-C_k)!$.

Ok, it's time to piece it all together. For color k, the amount it contributes to the answer is the sum of the amount it contributes before removing each textile:

$$\begin{aligned}
\sum_{i=1}^{N} \left(\left[\binom{N}{C_k} - \binom{i-1}{C_k} \right] \cdot C_k! \cdot (N-C_k)! \right) &= C_k! \cdot (N-C_k)! \cdot \sum_{i=1}^{N} \left[\binom{N}{C_k} - \binom{i-1}{C_k} \right] \\
&= C_k! \cdot (N-C_k)! \cdot \left[N \cdot \binom{N}{C_k} - \sum_{i=1}^{N} \binom{i-1}{C_k} \right] \\
&= C_k! \cdot (N-C_k)! \cdot \left[N \cdot \binom{N}{C_k} - \binom{N}{C_k+1} \right]
\end{aligned}$$

We use hockey-stick identity to simplify the summation expression.

After precomputing factorials and inverse factorials in $\mathcal{O}(N)$ time, we can add the contribution of each color in constant time.

Time complexity: $\mathcal{O}(N+M)$

Problem: Maxwell Zhang
Flavortext: Gabriel Wu
Editorial: Maxwell Zhang
Code: C++, Java, Python

§10 Tanya's Revenge

To solve this problem, we perform dynamic programming on a tree. Consider the state $dp[i][t][j]$, where this describes the maximum battle readiness if we have already directed the paths in the subtree of village i. The remaining two states represent two possibilities:

- If $t = 0$, then we have $dp[i][t][j]$ means our current state has j directed paths going upwards through village i, using only the edges in the subtree of village i.
- If $t = 1$, then we have $dp[i][t][j]$ means our current state has j directed paths going down through village i that end at battle forts, using only the edges in the subtree of village i.

One might think that we have to store both the directed paths going upwards through village i and the number of paths going downwards through village i ending at a battle fort. However, depending on whether the edges that connects village i to its parent is oriented, we only need to consider one of these cases. We perform a depth first search starting from the headquarters at village 1. We evaluate $dp[i][t][j]$ for all children of a village before we process it. To compute $dp[i][t][j]$ after we've computed $dp[c][t][j]$ for all the children of village i, we perform a knapsack DP to combine two subtrees. This knapsack DP may seem like it is $\mathcal{O}(N^3)$. However, we note that when combining two subtrees of size x, y, we only iterate at most xy times. The number of iterations in our knapsack DP can be seen to biject to the number of paths between any two villages, which is in fact $\mathcal{O}(N^2)$. thus our overall solution is in fact $\mathcal{O}(N^2)$.

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

Problem: Colin Galen
Flavortext: Timothy Qian
Editorial: Timothy Qian
Code: C++, Java, Python

§11 Sphinx Economics

We first define the state $f(i, j)$, which represents the maximum amount of money Faris can guarantee at the end of the questions if **she currently has \$1**, there are i questions left, and the Sphinx has currently picked the same answer j times in a row previously, where we assume that $j < Q$.

First, we show that $f(i, j)$ is monotonically nondecreasing as j increases for $j \in [1, Q-1]$. Assume for the sake of contradiction that $f(i, a) > f(i, b)$ for $1 \leq a < b \leq Q-1$. Then, when determining $f(i, b)$, we can have Faris bet as if $j = a$, or the Sphinx has already had a of the same answers in a row. Since every possible scenario if there are i questions left and has previously had b of the same answers is a subset of the set of scenarios where there are i questions left and the Sphinx has had a of the answers in a row the same, Faris can have at least $f(i, a)$ money by the end of the i questions. This means that $f(i, a) \leq f(i, b)$ a contradiction.

We establish our base cases. $f(1, j)$ for $j < Q-1$ is equal to 1, because Faris have no guarantee of earning anything. However, $f(1, Q-1) = 2$, because the Sphinx is forced to pick an answer, so Faris should go all in to earn the maximum amount of money. Also, $f(N, 0) = f(N-1, 1)$ because Faris should not bet anything on the first round. This is because there is no guaranteed benefit to anything on the first turn as Faris knows nothing about what the Sphinx will do.

First, we take care of the case of $j = Q-1$. In this case, we have $f(i, Q-1) = 2 \cdot f(i-1, 1)$, since the Sphinx is forced to pick an answer, so Faris can bet all of her money. Now we consider the case of $j < Q-1$. First, we define a *run* as the set of consecutive previous answers that the Sphinx has chosen that are all the same. There are two cases as to what to do.

Case 1: Faris picks the same answer as the Sphinx's run. Let's say Faris bet $0 \leq x \leq 1$ this round. Then if Faris ends up guessing right, she ends up guaranteeing $(1+x) \cdot f(i-1, j+1)$ dollars. If Faris ends up guessing wrong, she end up guaranteeing $(1-x) \cdot f(i-1, 1)$ dollars. Since "guarantee" implies a worst case scenario, we set $f(i, j) = \min((1+x) \cdot f(i-1, j+1), (1-x) \cdot f(i-1, 1))$. However, since $f(i, j)$ is nondecreasing as j increases, we have that the minimum of this is always $(1-x) \cdot f(i-1, 1) \leq (1+x) \cdot f(i-1, j)$. So Faris should always assume she always loses, and thus the maximum she can guarantee in this case is $(1-x) \cdot f(i-1, 1)$. Therefore, she should bet 0, in which case

$$f(i, j) \geq f(i-1, 1)$$

Case 2: Faris picks the answer that's different from the Sphinx's current run. Let's say Faris bets $0 \leq x \leq 1$ this round. Then she has that $f(i, j) = \min((1+x) \cdot f(i-1, 1), (1-x) \cdot f(i-1, j+1))$. Now, we show that to choose an x to maximize this, the two components inside the minimum must be equal. Let $A = f(i-1, 1), B = f(i-1, j+1)$, where we know that $A \leq B$ by what we've shown before. We know $B \neq 0$ since Faris could always just bet nothing each time and guarantee her current \$1. So this is equivalent to finding the x that maximizes $\min(C(1+x), 1-x)$ by letting $C = \frac{A}{B}$ for some $0 \leq C \leq 1$. Graphing $y = \min(C \cdot (1+x), 1-x)$ from $[0, 1]$, it is clear that the graph consists of two lines, first with an increasing slope, then with a decreasing slope, so their maximum is where these two lines intersect. Thus, to maximize the minimum of this, we set these two components in the minimum equal. We get that $(1+x) \cdot f(i-1, 1) = (1-x) \cdot f(i-1, j+1)$. Solving for x yields

$$x = \frac{f(i-1,j+1) - f(i-1,1)}{f(i-1,1) + f(i-1,j+1)}$$

Thus, plugging this back in yields

$$f(i,j) \geq \frac{2 \cdot f(i-1,j+1) \cdot f(i-1,1)}{f(i-1,j+1) + f(i-1,1)}$$

Now we collate theses two cases, and show that it's always optimal to go with Case 2. Note that this is the harmonic mean of $f(i-1,1)$ and $f(i-1,j+1)$, and we know that $f(i-1,1) \leq f(i-1,j+1)$. Thus, this means that

$$f(i-1,1) \leq \frac{2 \cdot f(i-1,j+1) \cdot f(i-1,1)}{f(i-1,j+1) + f(i-1,1)}$$

Thus, we know that it's optimal to pick something different that the Sphinx's current run. Therefore, we have that for $j < Q-1$.

$$f(i,j) = \frac{2 \cdot f(i-1,j+1) \cdot f(i-1,1)}{f(i-1,j+1) + f(i-1,1)}$$

Thus, using this recurrence, we can calculate the value of $f(N,0) = f(N-1,1)$ in $\mathcal{O}(NQ \log MOD)$, which is enough to solve subtask 1.

For subtask 2, we exploit the fact that the recursive formula is based on the harmonic mean. Perhaps the easiest way to see this is letting $g(i,j) = \frac{1}{f(i,j)} \cdot 2^i$. Then our recursion for $j < Q-1$

$$g(i,j) = \begin{cases} g(i-1,1) + g(i-1,j+1) & j < Q-1 \\ g(i-1,1) & j = Q-1 \end{cases}$$

We start with the base case of $g(1,1) = g(1,2) = \cdots = g(1,Q-2) = 2, g(1,Q-1) = 1$. Let's say we let $g(0,1) = 1$ and $g(i,1) = 0$ for $i \leq -1$. Then, we can use induction to derive that

$$g(i,1) = g(i-1,1) + g(i-2,1) + \cdots + g(i-Q+1,1)$$

Thus, we can compute $g(N-1,1)$ in $\mathcal{O}(N)$ using prefix sums. And then we can convert back from $g(N-1,1)$ to $f(N-1,1)$ for an overall $\mathcal{O}(N)$ solution.

Fun fact: The general formula for $g(i,1)$ is actually the formula for the $Q-1$-Fibonacci number. Using matrix exponentiation, you can solve problem in $\mathcal{O}(Q^3 \log N)$. We didn't put this on the test because we didn't feel it added anything nice to the problem.

Time complexity: $\mathcal{O}(NQ \log MOD), \mathcal{O}(N), \mathcal{O}(Q^3 \log N)$

Problem: Gabriel Wu and Timothy Qian
Flavortext: Timothy Qian
Editorial: Timothy Qian
Code: C++, Java, Python

§12 Building Atlantis

Notice that, as long as each pillar is covered by at least one robot, the heights of all of the pillars grow arbitrarily high. Thus, after a long time, the value d_i of any robot is pretty much nothing relative to the height of the pillars. Thus, we can treat this problem as continuous: instead of considering discrete moves, we think of the robots as distributing "growth rate" among their interval. Let $P(i, j)$ be the *proportion* of the time that robot i chooses to grow pillar j as time grows to infinity. Note that $P(i, j) = 0$ whenever $j < l_i$ or $j > r_i$. Now, define the *growth rate* of a pillar j be

$$G(j) := \sum_{i=1}^{N} f(i, j).$$

The asymptotic ratio of the heights of two pillars is just equal to the ratio of their growth rates (think of this as L'Hopital's rule). Now the problem is reduced to finding the growth rate of each pillar. Observe that if there are two pillars j and k that are both in the interval of robot i, and if $G(j) < G(k)$, then $P(i, k)$ must be 0. This is because if k is asymptotically taller than j, then robot i will never choose k over j.

This inspires the following solution. We can binary search for the minimum growth rate, and then do a separate binary search for the maximum growth rate. To compute the minimum growth rate, we ask the confirmation question **"Is there a way for each robot to distribute its growth rate such that $G(j) \geq X$ for all pillars j"**? (In this question we are temporarily ignoring the fact that the behavior of the robots is determined deterministically, instead treating it as if there is a single mastermind that controls the robots' choices.) If the answer to the confirmation question is no, then obviously all pillars can't have at least X growth rate when the robots are following their usual algorithm, meaning that the minimum growth rate must be less than X. Here, "usual algorithm" refers to the robots' normal process of always choosing the minimum height pillar on a move.

What if the answer to the confirmation question is yes? For the sake of contradiction, assume that when the robots are following their usual algorithm, there is a non-empty set of pillars that have less than X growth rate. Let S be this set. Then any robot with an interval containing any pillar in S must be devoting all of their growth rate to pillars in S (by our previous observation). But if as many robots as possible are contributing to the total growth rate of S, and yet still no pillar in S is hitting X growth rate, then the answer to the confirmation question *cannot* be yes. This is a contradiction. Therefore, if the answer to the confirmation questions is yes, then when the robots are following their usual algorithm all pillars have at least X growth rate. Therefore **the minimum growth rate is greater than equal to X iff the answer to the confirmation question is yes.**

So how do we actually answer this confirmation question for the minimum? There is a slick greedy solution. Sort all the robots by left endpoint, then sweep from left to right. At pillar j, first add all robots with $l_i = j$ to a priority queue. Then greedily take growth rate from the robots that will "expire first" (AKA has the left-most right endpoint), until you have reached X growth rate for that pillar. Proceed to the next pillar. If you ever run out of robots in the priority queue, the answer is automatically no. But if you get through all of the pillars, successfully distributing X growth rate to each pillar, the answer is yes. So now we know how to find the minimum growth rate

in $\mathcal{O}((N+M)\log N \log precision)$. The $\log N$ factor comes from the priority queue; the $\log precision$ factor comes from the binary search.

We can find the maximum growth rate in a very similar way. We binary search with the confirmation question question **"Is there a way for each robot to distribute its growth rate such that $G(j) \leq X$ for all pillars j"**. If the answer to this question is no, then the maximum growth rate cannot be less or equal than X; it must be greater than X. What if the answer is yes? Like before, for the sake of contradiction assume that when the robots are following their usual algorithm, there is a non-empty set S of pillars that have greater than X growth rate. Then any robot with an interval containing any pillar in S, that doesn't have all its pillars in S, must be ignoring these pillars completely (because they would never end up as the lowest pillars in their range). But if as few robots as possible are contributing to the total growth rate of S, yet they are all still surpassing X growth rate, then the answer to the confirmation question must have been no. Once again, this is a contradiction. Therefore, **the maximum growth rate is less than or equal to X iff the answer to the confirmation question is yes**.

You'll notice the maximum growth rate binary search has a very similar structure as the binary search for the minimum. Answering the confirmation question is also very similar, except for this time you're *trying* to get rid of as much of your growth rate as possible at each pillar without going over X. If you end up with a non-empty priority queue after pillar M, or if you ever have growth rate that "expires" because you reach its right endpoint without using it all up, then the answer is no. Otherwise, the answer is yes. The time complexity is the same.

At the end, we simply print the minimum growth rate divided by the maximum growth rate. Other ways to answer the confirmation question may also have passed our time limit, including using flow with a segment tree graph that balbit came up with.

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

Problem: Gabriel Wu
Flavortext: Timothy Qian
Editorial: Gabriel Wu
Code: C++, Java, Python