Task 1: Fraud (`fraud`)

Authored and prepared by: Ho Xu Yang, Damian

Subtask 1

Limits: $B_i = 0$

Since $B_i = 0$, S_i is equal to $A_i \times X$.

Hence $S_i > S_j \iff A_i \times X > A_j \times X \iff A_i > A_j$ (since $X > 0$).

As it suffices to ensure $S_i > S_{i+1}$ (by the transitive property of inequalities), the answer is `YES` if and only if $A_i > A_{i+1}$ for all $1 \leq i < N$.

Time Complexity: $O(N)$

Subtask 2

Limits: $N = 2$

We observe that $S_1 > S_2 \iff (A_1 - A_2) \times X + (B_1 - B_2) \times Y > 0$.

The answer is `YES` if and only if at least one of $(A_1 - A_2)$ and $(B_1 - B_2)$ is positive.

Time Complexity: $O(1)$

Subtask 3

Limits: $2 \leq N \leq 10^4$

From our observations in subtasks 1 and 2, if there exists $1 \leq i < N$ such that $A_i \leq A_{i+1}$ and $B_i \leq B_{i+1}$, the answer is `NO`.

Furthermore, if $A_i \geq A_{i+1}$ and $B_i > B_{i+1}$ (or vice versa), then S_i is trivially greater than S_{i+1}.

Hence, let us only consider the remaining case where $A_i > A_{i+1}$ and $B_i < B_{i+1}$ (or vice versa).

We note that $S_i > S_{i+1} \iff (B_i - B_{i+1}) \times Y > (A_{i+1} - A_i) \times X$.

Now let $m = \frac{A_{i+1} - A_i}{B_i - B_{i+1}}$.

Rearranging, we get either $\frac{Y}{X} > m$ or $\frac{Y}{X} < m$, depending on the sign of $(B_i - B_{i+1})$.

The answer is `YES` if and only if the intersection of all the inequalities is non-empty. Equivalently, we check that all pairs of inequalities have a non-empty intersection.

Time Complexity: $O(N^2)$

Subtask 4

Limits: (No further constraints)

Let us now optimise the approach described in subtask 3.

Instead of checking every pair of inequalities, we simply record the largest m value of a "rightwards" inequality m_{right} and the smallest m value of a "leftwards" inequality m_{left}.

Then, the answer is `YES` if and only if $m_{\text{left}} > m_{\text{right}}$.

Time Complexity: $O(N)$

Task 3: Password (password)

Authored and prepared by: Ng Yu Peng

Preliminaries

Note that if we change A_i, P_i to $A_i - c, P_i - c$ taken mod $K + 1$, the answer stays the same. Hence replace K with $K + 1$ and A_i with the remainder of $A_i - P_i + K$ when divided by K, then the problem becomes: given an array A of integers between 0 and $K - 1$, and the given operation, what is the minimum number of operations needed to make everything a multiple of K? We shall work with this formulation for this write-up.

Subtask 1

Limits: $N = 3$

For this subtask, if $A_i = 0$ treat it as $A_i = K$.

Note that if A_2 is not the maximum among the three numbers, we can always solve the problem in $K - \min(A_1, A_2, A_3)$ operations, and clearly this is the minimum possible number of operations necessary. Otherwise if A_2 is the maximum, we have two cases:

Case 1: $K - A_2$ operations involve A_2 In this case the optimal sequence of operations is to pick the range $(1, 3)$ $K - A_2$ times and then pick $(1, 1)$ $A_2 - A_1$ times and $(3, 3)$ $A_2 - A_3$ times, giving a total of $K + A_2 - A_1 - A_3$ operations.

Case 2: More than $K - A_2$ operations involve A_2. Then at least $2K - A_2$ operations are needed to make A_2 a multiple of K. We pick the range $(1, 3)$ $K - A_2$ times and the resulting array can be turned into multiples of K in K moves as we can treat it as $A_2 = 0$, giving a total of $2K - A_2$ operations.

The answer is the minimum of the above cases.

Time Complexity: $O(1)$

Subtask 2

Limits: $A_i \leq A_{i+1}$ for all $1 \leq i \leq N - 1$ and $S_i = 0$ for all $1 \leq i \leq N$

Let i be the smallest index such that $A_i \neq 0$. The problem can be solved in $K - A_i$ moves using the following algorithm: pick the range (i, N) until A_N is divisible by K, then keep picking $(i, N - 1)$ until A_{N-1} is divisible by K, and so on until $A_i, \ldots, A_N$ are all divisible by K. This is also the minimum number of moves needed to make A_i a multiple of K as well, hence it must be the minimum possible number of moves needed to make everything multiples of K. If no such i exists then the answer is 0.

Time Complexity: $O(N)$

Subtask 3

Limits: $K = 1$

In our solution we will treat this as $K = 2$ as we replaced K with $K + 1$. Notice that if we pick two ranges that overlap, the overlapped portion gets increased by 1 twice, so under modulo 2 there is no change. Hence we can remove the overlapped portion in both ranges and the result would still be the same. By repeating this process for any two overlapping ranges, clearly the sum of lengths of ranges decreases with each removal and hence the process must terminate.

Therefore we may assume that all ranges picked do not overlap. Hence for every range picked the numbers in the range must all be 1. Thus the minimum number of operations needed is the number of blocks of consecutive 1s.

Time Complexity: $O(N)$

Subtask 4

Limits: $N, K \leq 80$

Here we will try to make A_1 to A_i multiples of K and try to extend the solution to make A_{i+1} a multiple of K as well. Notice that any operation done on A_i can be extended to include A_{i+1} as well, and any operation done on A_{i+1} which does not involve A_i does not involve indices less than i as well. This motivates the following dynamic programming state: let $dp(i, j)$ be the minimum number of operations needed to make the first i numbers multiples of K, where j operations affect A_i. Then compute $dp(i, j)$, we first check that $A_i + j$ is a multiple of K, and passing that we take the minimum of:

- $dp(i - 1, l)$ for $l \geq j$, as we can pick j of the l operations on A_{i-1} to extend to A_i.
- $dp(i - 1, l) + j - l$ for $l < j$ as we can extend all l operations on A_{i-1} to A_i and use another $j - l$ operations to hit j operations on A_i.

Then the remaining problem is what the bound on j should be. Notice that if we use operations only on single elements, each element can be made a multiple of K in at most $K - 1$ moves, so the total number of moves needed is at most $N(K - 1)$, thus it is never optimal to have more than $N(K - 1)$ operations on any A_i. Thus the dp state is $O(N^2K)$ and transition is $O(NK)$ on values of j such that $A_i + j$ is a multiple of K (of which there are $O(N^2)$ states) and $O(1)$ on values of j which do not satisfy that (as they are rejected immediately).

Time Complexity: $O(N^3K)$

Subtask 5

Limits: $N \leq 400$

We can reduce the state to $O(N^2)$ by only considering values of j that make $A_i + j$ a multiple of K and ignoring all other values. In other words, let $dp(i, j)$ be the minimum number of operations needed to make the first i elements multiples of K and $Kj + (K - A_i)$ operations affect A_i. Then transition is $O(N)$.

Time Complexity: $O(N^3)$

Subtask 6

Limits: $N \leq 3000$

For this subtask we need to speed up the transition. Clearly $dp(i, j+1) \geq dp(i, j)$. Hence if we pick l such that $Kl+(K-A_{i-1}) \leq Kj+(K-A_i)$ and $K(l+1)+(K-A_{i-1}) > Kj+(K-A_i)$, then

$$dp(i, j) = \min(dp(i, l) + K(j - l) + A_{i-1} - A_i, dp(i, l + 1))$$

Hence the transition is now $O(1)$ as we can find such l in $O(1)$ easily.

Time Complexity: $O(N^2)$

Subtask 7

Limits: No additional constraints

To solve the final subtask, we need to take an entirely different approach to the question. Consider the difference array

$$B_1 = A_1 - 0, B_2 = A_2 - A_1, \ldots, B_N = A_N - A_{N-1}, B_{N+1} = 0 - A_N$$

where all values are taken modulo K, so every element in this array is between 0 and $K - 1$ inclusive.

Notice that an operation chosen for indices $i \leq j$ adds 1 to B_i and reduces B_{j+1} by 1, and every other B_l is unaffected. Thus each operation is equivalent to transforming $B_i \to B_i + 1$ and $B_j \to B_j - 1$ for some $i < j$. Every A_i is a multiple of K if and only if every $B_i = 0$ mod K.

Note that the order of operations is inconsequential. Additionally, $B_1 + B_2 + \cdots + B_{N+1} = 0$ (mod K) by summing it through.

We'll prove a few properties to aid in our solution:

Property 1. In the optimal solution, each element only receives +1 operations or only -1 operations.

Proof. Suppose otherwise, then there is an element that has had both +1 and -1 operations done on it, let it be B_j. This means among all the operations, there exist two operations one on i, j where $i < j$ and one on j, k where $j < k$. But these two operations together yield $A_i \to A_i + 1$ and $B_k \to B_k - 1$, so we might as well replace these two operations with one on i, k, giving us a solution with fewer operations, contradiction.

In the optimal solution, let C_i be the total amount added to B_i (so C_i could be negative). Additionally let $S_i = C_1 + C_2 + \cdots + C_i$.

Property 2. $S_{N+1} = 0$.

Proof. As each operation gives one +1 and one -1, the result follows.

Property 3. For each $m = 1, 2, \ldots, N + 1, S_m \geq 0$.

Proof. Every -1 to a B_j corresponds to a +1 to a B_i with $i < j$. Hence the number of -1s contributing to $C_1, C_2, \ldots, C_m$ is at most the number of +1s contributing to $C_1, C_2, \ldots, C_m$, which proves the above result.

In fact, a simple greedy algorithm shows that any array C satisfying property 3 will have a valid sequence of operations yielding those values of C. Specifically, pick the smallest index i with $C_i > 0$, use C_i +1s on index i, and split the C_i -1s to $j > i$ with $C_j < 0$, and after making the changes $C_i = 0$ and changes to the $C_j < 0$ chosen, repeat this process until all elements in C are equal to 0.

Also notice that the number of operations used is $\frac{|C_1|+|C_2|+\cdots+|C_{N+1}|}{2}$. Hence the problem reduces to finding an array C satisfying properties 2 and 3, as well as $B_i + C_i = 0 \bmod K$ for all i, and minimising the above value.

Property 4. $0 \leq |C_i| \leq K - 1$ for all $i = 1, 2, \ldots, N + 1$.

Proof. Suppose that some $C_i \geq K$ (the case where $C_i \leq -K$ is similar). Pick the smallest index j with $C_j < 0$. Replace $C_i \to C_i - K, C_j \to C_j + K$. Then Property 2 is still true.

If $i < j$, Property 3 is still true as $C_1, \ldots, C_{j-1} \geq 0 \geq S_1, \ldots, S_{j-1} \geq 0$ and $S_j, \ldots, S_{N+1}$ are unchanged.

If $i > j$, note that $S_j, S_{j+1}, \ldots, S_{i-1}$ all do not decrease and $S_i, \ldots, S_{N+1}$ are unchanged. so Property 3 is still true.

Additionally $|C_1| + |C_2| + \cdots + |C_{N+1}|$ decreases, so we use fewer operations, contradicting the assumption that our sequence of moves is optimal.

Summarising what we have so far:

Property 1. In the optimal solution, each element only receives +1 operations or only -1 operations.

Property 2. $S_{N+1} = 0$.

Property 3. For each $m = 1, 2, \ldots, N + 1$, $S_m \geq 0$.

Property 4. $0 \leq |C_i| \leq K - 1$ for all $i = 1, 2, \ldots, N + 1$.

Remember that $0 \leq B_i \leq K-1$. Hence from Property 4 we know $C_i = K - B_i$ or $C_i = -B_i$.

Hence this motivates a rough idea for a greedy algorithm: we start with all $C_i = K - B_i$, and we shall greedily choose indices i to do the change $C_i = -B_i$.

Note that originally $C_1 + C_2 + \cdots + C_{N+1} = KN - (B_1 + B_2 + \cdots + B_{N+1})$ which is a multiple of K. Additionally every time we change $K - B_i \to -B_i$ the sum $C_1 + \cdots + C_{N+1}$ decreases by K.

Let $C_1 + \cdots + C_{N+1} = KT$ originally, then since the final configuration has $C_1 + \cdots + C_{N+1} = 0$ we make S changes $K - B_i \to -B_i$.

Also notice that, changing $K - B_i \to -B_i$ basically subtracts K from each of $S_i, S_{i+1}, \ldots S_{N+1}$.

Greedy Algorithm

We sort $(B_i, -i)$, then iteratively, in the sorted order, check if we can change $C_i = K - B_i$ to $C_i = -B_i$ without compromising Property 3. If possible we do the change. This can be checked quite quickly with a range add update, range minimum query segment tree. Then compute the value $\frac{|C_1| + |C_2| + \cdots + |C_{N+1}|}{2}$ at the end to get the answer.

Time Complexity: $O(N \log N)$

Proof that the algorithm is optimal

We first show that the algorithm makes T changes, where $C_1 + \cdots + C_{N+1} = KT$ originally before any changes, as this would mean that in the final configuration $C_1 + \cdots + C_{N+1} = 0$ so the algorithm yields a valid array C.

If more than T changes were made, $C_1 + \cdots + C_{N+1} < 0$, which won't happen as the greedy algorithm does not compromise Property 3.

Otherwise if fewer than T changes were made, suppose X changes were made. Then pick the largest index i such that $C_i = K - B_i$.

This means for all $j > i$, $C_j = -B_j$. Changing $K - B_i \to -B_i$ does not contradict Property 3, because for all $l < i$ we still have $S_l \geq 0$ and for all $l \geq i$ we have $S_l \geq S_{N+1} \geq 0$ as $X < T$ and $C_m \leq 0$ for all $m \geq l$, which means that the algorithm has not terminated yet, a contradiction.

Since both cases above are not possible, this means the algorithm makes exactly T changes.

Now consider a sequence of changes different from our greedy algorithm, and we will show that it cannot be optimal.

Notice that changing $K - B_i \to -B_i$ changes $|C_1| + |C_2| + \cdots + |C_{N+1}|$ by $2B_i - K$, so it is better to change C_i with the smaller B_i as we want to minimise that sum.

Now let $X_1, X_2, \ldots, X_T$ be the indices which were changed, sorted in order of changes in the greedy algorithm. So $B_{X_1} \leq B_{X_2} \leq \ldots \leq B_{X_T}$.

Now let $Y_1, Y_2, \ldots, Y_T$ be the indices which were changed, for some S changes which still end with Property 3 being satisfied, such that $B_{Y_1} \leq \ldots B_{Y_T}$. We will show that if $B_{Y_i} \neq B_{X_i}$ for some i then we can modify the sequence of S changes that result in Y to decrease the total number of operations $\frac{|C_1|+|C_2|+\cdots+|C_{N+1}|}{2}$.

Consider the smallest index i such that $B_{Y_i} \neq B_{X_i}$. So $B_{X_1} = B_{Y_1}, B_{X_2} = B_{Y_2}, \ldots, B_{X_{i-1}} = B_{Y_{i-1}}, B_{X_i} \neq B_{Y_i}$.

Note that for any $i < j$, if C_i was changed but not C_j, changing C_j instead of C_i would still satisfy Property 3, as we add K to a larger suffix of the array S than we subtract K from. Hence we may assume that $X_1 = Y_1, \ldots, X_{i-1} = Y_{i-1}$, because for each $B_j = k$ picked we may pick the largest unpicked j with $B_j = k$.

If $B_{Y_i} < B_{X_i}$, the greedy algorithm would have picked Y_i instead of X_i, so we must have $B_{X_i} < B_{Y_i}$.

Hence we assume C_{X_i} was not changed in the sequence of changes giving the array Y. For that sequence of changes, pick the smallest index j that is not among $X_1, X_2, \ldots, X_i$. We change C_{X_i} instead of C_j in the Y sequence. It suffices to show that the resulting array C still satisfies Property 3.

If $X_i > j$, this is clear as we are subtracting K from a smaller suffix of the array S.

Else $X_i < j$ then for all $l < j$, by the minimality of j and the fact that the greedy algorithm does not compromise Property 3, $S_l \geq 0$, and for any $l \geq j$, S_l does not change when we change C_{X_i} instead of C_j.

Thus Property 3 is still satisfied.

If $B_{X_i} < B_{Y_i} \leq B_j$, $B_{X_i} + K - B_j < B_j + K - B_{X_i}$, this swap would decrease $|C_1| + |C_2| + \cdots + |C_{N+1}|$, giving a better solution.

Therefore, if $B_{X_i} \neq B_{Y_i}$ for some i then we have shown that the sequence of changes giving Y is not optimal, completing the proof.

Task 4: Tiles (`tiles`)

Authored and prepared by: Leong Eu-Shaun

Subtask 1

Limits: $1 \leq N, Q \leq 8$

For each query, run a brute force algorithm. One possible solution is to scan the grid from left to right, starting at the top row and ending at the bottom. If a square is black or has already been tiled, skip it. Otherwise, try one of 3 possibilities: leaving it blank, placing a tile across it and the square to its right, or placing a tile across it and the square below it. Once the end of the grid is reached, increment the answer by 1.

Time complexity: $O(answer)$

Subtask 2

Limits: There will never be any black squares.

The answer to a range query depends only on the length of the range. Use a dynamic programming solution that keeps track of the ”profile” at each column. Suppose a pattern covering n columns has some tiles that jut out into column $n+1$. Let the profile of column $n+1$ be represented by the bitmask b where a tiled square is recorded as a 1 and an untiled square is a 0. For example, if the first and second squares in column $n+1$ are untiled while the third is tiled, then $b = 100_2 = 4$, since the third column corresponds to the hundreds digit in binary.

Let $dp[n][b]$ be the number of ways to tile the first n columns with a profile of b jutting out into column $n+1$. To calculate $dp[n][b]$, iterate over all profiles b' of the previous column. Run a simple brute force to find out how many arrangements of tiles can be placed in column n that produce profile b in column $n+1$ while avoiding profile b' and the black squares in column n.

Time complexity: $O(N)$ with high constant time

Subtask 3

Limits: $1 \leq N, Q \leq 7000$

An inefficient implementation of the previous solution will now fail, because a linear runtime for each query will result in a $O(NQ)$ solution with high constant time. In particular, if for each possible profile of a column, you try all 8 profiles of the previous column and use brute force to count the number of ways tiles can be laid across both columns, your solution is likely to be too slow to pass.

Instead, you can precompute the number of ways to tile 2 columns with profiles b and b'. This can be stored as a series of weights in a recurrence relation expressing $dp[n][b]$ in terms of a sum of $dp[n-1][b']$ for some profiles b'.

By precomputing the weights, most efficient solutions should be able to pass this subtask.

Time complexity: $O(NQ)$

Subtask 4

Limits: $1 \leq N, Q \leq 30000$

We need a data structure to accommodate point updates and range queries. For this, a segment tree will suffice. For each node covering columns s to e, store an 8-by-8 array $dp[a][b]$ which counts the number of ways of tiling the range $[s, e]$ while satisfying constraints imposed by a and b. Bitmask a represents the squares in column s already covered by tiles that jut out from column $s-1$, while bitmask b represents the tiles from column e jutting out into column $e+1$.

Now, we must combine two nodes $[s, m]$ and $[m+1, e]$ with 8-by-8 arrays L and R respectively into a parent node $[s, e]$ with array P. To calculate $P[a][b]$, iterate over all possible bitmasks i. i represents the tiles that jut out from column m in the left node into column $m+1$. For each i, add $L[a][i] \cdot R[i][b]$ to $P[a][b]$. This takes 8^3 operations.

To calculate the answer to a range query, simply determine the array $dp[a][b]$ for the range using the combining operation described above, and output $dp[0][0]$.

Time complexity: $O((N+Q)\log N)$

Task 5: Pond (`Pond`)

Authored and prepared by: Jeffrey Lee

Introduction

For ease of reading, let us say that the linear pond stretches from left to right, with point 1 being the leftmost point of the pond and point N being the rightmost point. We will denote the distance to the right from point 1 of each point i as s_i, i.e. $s_i = \sum_{n=1}^{i-1} D_i$.

Subtask 1

Limits: $N \leq 100$

We introduce a total cost T as the total number of algal strands eaten or remaining in the pond, given the route taken so far. In other words, T is defined as follows:

$$T = \sum_{n=1}^{N} \begin{cases} \text{time } n \text{ was first visited,} & \text{if } n \text{ has been visited} \\ \text{time elapsed since start of swim,} & \text{if } n \text{ has not been visited} \end{cases}$$

Note that T starts equal to 0, strictly increases as we travel around the pond, and ends equal to the total number of algal strands eaten (once we have visited all points). In particular, whenever we swim a distance D while there are u unvisited points, T increases by $D \cdot u$.

We also introduce the $O(N^3)$ states (l, i, r), each representing that we are currently at point i, and have visited the points left of K up to l as well as those right of K up to r. The initial state is hence (K, K, K), while the valid end states are $(1, i, N)$ for all $1 \leq i \leq N$. We can proceed from every state (l, i, r) to the next by moving one point to the left or one point to the right into the two successors $(\min(l, i-1), i-1, r)$ and $(l, i+1, \max(r, i+1))$ respectively.

With this set of states and total cost T, we can perform Dijkstra's Algorithm on the states to find the minimum T required to form a complete route, making use of the following unidirectional edges:

$$(l, i, r) \longrightarrow \begin{cases} (\min(l, i-1), i-1, r), & T \mathrel{+}= D_{i-1} \cdot (N - r + l - 1) \\ (l, i+1, \max(r, i+1)), & T \mathrel{+}= D_i \cdot (N - r + l - 1) \end{cases}$$

Time complexity: $O(N^3 \log N)$

Subtask 2

Limits: $N \leq 2000$

Notice that if we choose at some state (l, i, r) to proceed left or right to a point (vertex) which has already been visited, it would not be optimal to turn back before a new vertex has been visited in that respective direction at state $(l-1, l-1, r)$ or $(l, r+1, r+1)$ - as turning back would imply visiting a state for a second time with increased T.

As a consequence, we can extract the states of the form (l, l, r) or (l, r, r), in which a new vertex is being visited, and construct direct edges between those states while pruning away all others. The new recurrences would then be:

$$(l, l, r) \longrightarrow \begin{cases} (l-1, l-1, r), & T \mathrel{+}= D_{i-1} \cdot (N - r + l - 1) \\ (l, r+1, r+1), & T \mathrel{+}= (s_{r+1} - s_l) \cdot (N - r + l - 1) \end{cases}$$

$$(l, r, r) \longrightarrow \begin{cases} (l-1, l-1, r), & T \mathrel{+}= (s_r - s_{l-1}) \cdot (N - r + l - 1) \\ (l, r+1, r+1), & T \mathrel{+}= D_r \cdot (N - r + l - 1) \end{cases}$$

As the graph formed by the $2N^2$ remaining states and their edges is now directed and acyclic, the minimum T to reach each state can be computed via dynamic programming to reduce the time complexity per state from $O(\log N)$ to $O(1)$.

Time complexity: $O(N^2)$

Subtask 3

Limits: $K \leq 20$

Since the states (l, l, r) and (l, r, r) in which $r < K$ or $K < l$ cannot be reached from the initial state (K, K, K), we can prune them away as well to leave only the states with $l \leq K \leq r$. There will remain $2K(N - K) = O(KN)$ such states, each evaluated in a time complexity of $O(1)$.

Time complexity: $O(KN)$

Subtask 4

Limits: $D_i = 1$

Replacing the total cost T, let us introduce a cost heuristic C which is equal to the sum of algal strands eaten and current minimum algal strands to be eaten for each point:

$$C = \sum_{n=1}^{N} \begin{cases} \text{time } n \text{ was first visited,} & \text{if } n \text{ has been visited} \\ \text{time elapsed since start of swim} + |s_n - s_i|, & \text{if } n \text{ has not been visited} \end{cases}$$

where i is our current vertex.

Similarly, C starts equal to $\sum_{n=1}^{N} |s_n - s_K|$, is nondecreasing as we travel the pond, and ends equal to the total number of algal strands eaten. The key difference is that swimming a distance D leftwards with u_r unvisited vertices on the right now increases C by $2D \cdot u_r$, while swimming D rightwards with u_l unvisited leftside vertices increases C by $2D \cdot u_l$.

In Subtask 2, we observed that any optimal route has to fit the form

$$K \longrightarrow l_1 \longrightarrow r_1 \longrightarrow l_2 \longrightarrow r_2 \longrightarrow ... \longrightarrow l_x \longrightarrow r_x \longrightarrow 1 \longrightarrow N$$

where $1 < l_x < ... < l_1 \leq K < r_1 < ... < r_x \leq N$.

For this subtask, we seek to prove that the route $K \longrightarrow 1 \longrightarrow N$ is the optimal route which satisfies the above condition.

Firstly, take any one optimal route $K \longrightarrow l_1 \longrightarrow r_1 \longrightarrow l_2 \longrightarrow ... \longrightarrow r_x \longrightarrow 1 \longrightarrow N$. Its cost heuristic C increases as such

$$K \xrightarrow{2(K-l_1)(N-K)} l_1 \xrightarrow{2(r_1-l_1)(l_1-1)} r_1 \xrightarrow{2(r_1-l_2)(N-r_1)} l_2 \longrightarrow ... \longrightarrow r_x \xrightarrow{2(r_x-1)(N-r_x)} 1 \xrightarrow{0} N$$

However, consider also the alternative route $K \longrightarrow r_1 \longrightarrow l_2 \longrightarrow ... \longrightarrow r_x \longrightarrow 1 \longrightarrow N$, with a cost C' of

$$K \xrightarrow{2(r_1-K)(K-1)} r_1 \xrightarrow{2(r_1-l_2)(N-r_1)} l_2 \longrightarrow ... \longrightarrow r_x \xrightarrow{2(r_x-1)(N-r_x)} 1 \xrightarrow{0} N$$

The movements and cost incurred in both routes are identical past vertex r_1. This leaves only the first two segments of the original optimal route and the first segment of the alternative route as the difference, of which

$$\begin{aligned} C_0 &= 2(K-l_1)(N-K) + 2(r_1-l_1)(l_1-1) \\ &\geq 2(K-l_1)(r_1-K) + 2(r_1-K)(l_1-1) = 2(r_1-K)(K-1) = C_0' \end{aligned}$$

the alternative route has a cost less than or equal to that of the original route, with equality holding when $l_1 = K$.

Thus, we will do no worse to take the alternative route instead of the original optimal route, omitting the first leftwards segment to l_1. Repeating this argument allows us to further delete the new first rightwards movement to r_1 from the alternative route, and then l_2 through r_x until only $K \longrightarrow 1 \longrightarrow N$ remains.

Time complexity: $O(N)$

Subtask 5

Limits: $K \leq N, 2000, \forall i \not\equiv 0 \pmod{100} : D_i \geq D_{i+1}$

We say that a route of the form

$$K \longrightarrow r_1 \longrightarrow l_1 \longrightarrow r_2 \longrightarrow l_2 \longrightarrow ... \longrightarrow r_x \longrightarrow l_x \longrightarrow N \longrightarrow 1$$

with $1 \leq l_x < ... < l_1 < K$ is made out of $x + 1$ rebounds, where a rebound is a motion $i \longrightarrow k \longrightarrow j$ for $i < j \leq K \leq k$. In an optimal route, we would also implicitly have $K \leq r_1 < ... < r_x < N$, for which each rebound would invoke an independent increase in C of $2(s_k - s_j)(j-1) + 2(s_k - s_i)(N-k)$, and specifically the k taken in one rebound would affect the C incurred by the rebound itself but not the C of any other rebounds.

We call this cost $2(s_k - s_j)(j-1) + 2(s_k - s_i)(N-k)$ the ground cost of $i \longrightarrow k \longrightarrow j$, and will from here onwards assume it to be the actual cost of the rebound (as any properties we will be proving about rebounds need only apply to them when they are part of an optimal route).

Suppose for some fixed i and j and any particular k that $i \longrightarrow k+1 \longrightarrow j$ has a lower cost than $i \longrightarrow k \longrightarrow j$, that is,

$$\begin{aligned} & 2(s_k - s_j)(j-1) + 2(s_k - s_i)(N-k) \\ > & \ 2(D_k + s_k - s_j)(j-1) + 2(D_k + s_k - s_i)(N-k-1) \\ = & \ 2D_k(N-k+j-2) - 2(s_k - s_i) + 2(s_k - s_j)(j-1) + 2(s_k - s_i)(N-k) \end{aligned}$$

and equivalently $2D_k(N-k+j-2) - 2(s_k - s_i) < 0$. Then, if $D_k \geq D_{k+1}$ we would get

$$2D_{k+1}(N-(k+1)+s_j-2) - 2(s_{k+1} - s_i) < 2D_{k+1}(N-k+s_j-2) - 2(s_k - s_i) < 0$$

implying in turn that $i \longrightarrow k+2 \longrightarrow j$ must have a lower cost than $i \longrightarrow k+1 \longrightarrow j$. Therefore, an optimal route can be constructed without its rebounds using any k vertex surrounded by $D_k \geq D_{k+1}$, as k can always be replaced by at least one of $k-1$ or $k+1$ without increasing C.

We can solve this subtask by following the dynamic programming solution of Subtask 3 and pruning away all states (l, l, r) and (l, r, r) with $D_r \geq D_{r+1}$.

Time complexity: $O\left(K\frac{N}{100}\right)$

Subtask 6

Limits: $K \le 2000$

Let C_i be the minimum cost to reach each vertex i, left of K. This minimum cost can be constructed using a walk consisting entirely of one or more consecutive rebounds, giving rise to the recurrence

$$C_i = \min\left(C_j + 2(j-1)(s_k - s_j) + 2(N-k)(s_k - s_i)\right), \quad \forall i < j \le K \le k$$

Naively, this formula takes the shape of an $O(N^3)$ DP. We can rearrange the right-hand-side of the recurrence to obtain

$$\begin{aligned} &C_j + 2(j-1)(s_k - s_j) + 2(N-k)(s_k - s_i) \\ &= -2(N-k)s_i + \color{red}{2(N-k)s_k} + \color{blue}{C_j - 2(j-1)s_j} + \color{green}{2(j-1)s_k} \end{aligned}$$

with 1 term linear in s_i and 3 terms independent of i. This allows us to apply the convex hull optimization to this DP, with each (j, k) pair represented by one line over s_i. An $O(KN \log N)$ as-is implementation of this convex hull implies iterating over every i from K to 1, while inserting $N - K + 1$ lines after each i is evaluated. The convex hull would have the following general properties:

- Lines have nonpositive gradients which become more negative (decreasing k) as we go from left to right (in the direction of increasing s_i).
- At most 1 line from each k vertex will be present at any point in time, since all lines by a k node have the same gradient.
- Queries to the convex hull will occur with decreasing s_i (from right to left).

Since we cannot afford to directly compute all lines of such a convex hull, we will instead seek to maintain a lazy representation of it based on special properties of its lines. Revisiting the three terms independent of i,

- $\color{red}{2(N-k)s_k}$ is tied solely to its vertex k and does not change as j decreases.
- $\color{blue}{C_j - 2(j-1)s_j}$ is dependent solely on its iteration j and does not vary across all vertices k of its iteration.
- $\color{green}{2(j-1)s_k}$ decreases as the iteration j decreases; with lines from higher k vertices (lines further left in the convex hull) experiencing a strictly greater decrease than lines from lower k vertices.

Of particular note is the third term $2(j-1)s_k$, which affects how the relative positioning between the $N-K+1$ lines changes from iteration to iteration. It is this term which determines how lines become critical (uniquely minimal in the convex hull for at least one $s_i \in \mathbb{R}$) or redundant (not uniquely minimal) among the $N-K+1$ lines over successive iterations of j.

As $2(j-1)s_k$ shifts every vertex k's lines downwards by the same proportion $2s_k$ whenever the iteration j decreases, any three lines $k=a, k=b$, and $k=c$ with $c<b<a$ (line a on the left, line b in between and line c on the right) would have line b be critical w.r.t. a and c for exactly either an upper-bounded range $j \in (-\infty, z)$ or a lower-bounded range $j \in (z, \infty)$.

Therefore, every vertex's lines would be critical in a (possibly empty) range of $j \in (z_1, z_2)$, which we will precompute by finding the upper bounds and lower bounds separately. For upper bounds, we start with a list of upper bounds of every line k w.r.t its neighbours $k-1$ and $k+1$, and repeatedly record and delete the line with the lowest upper bound, replacing the bounds of its neigbours with those w.r.t. their own new neighbours accordingly. Vice versa can be done for lower bounds, thus determining the critical range of all vertices in $O(N \log N)$.

For this subtask, we will visit all j decsending from K to 0, and individually update all i left of j using the convex hull of j's iteration. We maintain a segment tree storing the indices of the critical vertices at j, and for each i binary search down the convex hull to find the optimal line at s_i. There are then $O(K^2)$ such updates, each taking $O(\log N)$ time.

Time complexity: $O(K^2 \log N + N \log N)$

Subtask 7

We further combine the terms $C_j - 2(j-1)s_j$ and $2(j-1)s_k$ to observe that each iteration j interacts with the overall convex hull when added by replacing some or none of the leftmost lines with its own, causing the overall convex hull to gradually become partitioned from right to left into ranges of decreasing optimal j.

The indices of these ranges' j can be stored in a vector, with an iteration j being present at any point if at least one of its lines is critical in the overall convex hull. The vector can be maintained whenever a new j is added by either discarding it if it provides no lines better than those of the latest prior iteration, or inserting it at the end after removing all previous iterations it makes redundant. This involves a total of $O(N)$ insertions and deletions of iterations, each taking $O(\log N)$ time to confirm or reject.

We can then perform the DP by iterating over i from K to 0. At each i, we determine the optimal k line and its j iteration via a pointer walk, taking care to skip over redundant lines and iterations. The pointer walk will hence make a total of $O(N)$ steps of $O(\log N)$ time each.

Time complexity: $O(N \log N)$