GRCPC 2025 Editorial



A. zerorush

Author: Panagiotis Karelis.
Analysis: Anastasis Koliopoulos.

Tags: Greedy.

To begin, we reduce the problem to a decrease-only scenario by considering the absolute values of the sequence. Next, we observe that all operations must be applied to a single element — specifically, the smallest even number. This choice is optimal, since it will reach zero in the fewest steps among all even numbers. Note that an odd number can never become zero, so a sequence consisting entirely of odd numbers admits no solution. Let a denote the selected number, and let k be the number of operations needed for a to reach zero. Then

$$a - 2k = 0 \iff a = 2k \iff k = \frac{a}{2},$$

which is the answer to the problem.

The above approach requires only a single pass through the sequence, yielding a time complexity of $\mathcal{O}(N)$.

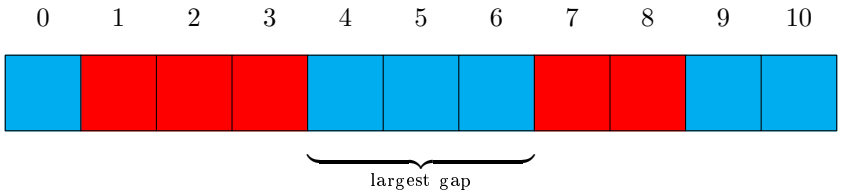
Indicative code

B. foodbreak

Author: Manolis Tsamis.
Analysis: Anastasis Koliopoulos.

Tags: Brute force, Sortings.

Let us first consider a simple, non-optimal but intuitive approach. We define an array $A[0 \dots T]$, where $A[i] =$ red if there is a presentation at timestamp i, and $A[i] =$ blue otherwise. For each presentation $i \in \{1, \dots, N\}$, we set every cell in the segment $[s_i, e_i) = [s_i, e_i - 1]$ to red. After processing all presentations, we find the longest contiguous segment consisting of cells with blue color, which gives us the answer.



This algorithm runs in worst-case time $\mathcal{O}(N \cdot \max_{1 \le i \le n}\{e_i - s_i\} + T) = \mathcal{O}(N \cdot T + T) = \mathcal{O}(N \cdot T)$, and requires $\mathcal{O}(T)$ space, which is insufficient to meet the time and memory limits. Let us now consider a better approach.

We consider the set of intervals $S = \{[s_i, e_i) : i \in [N]\} \cup \{(-\infty, 0), [T, \infty)\}$. For convenience, we will refer to the two additional intervals as (s_0, e_0) and $[s_{N+1}, e_{N+1})$, respectively.

We know that the desired interval must end exactly before the start of some presentation i and begin exactly after the end of some presentation j. That is, it must be of the form $[e_j, s_i) = [e_j, s_i - 1]$.

Observation. For a given interval i, where $1 \le i \le N + 1$, the maximum gap that ends before the start of i is equal to

$$\begin{aligned} \ell_i &= |[e_j, s_i - 1]| \\ &= \max\{s_i - 1 - e_j + 1, 0\} && ([e_j, s_i - 1] \subset \mathbb{N}) \\ &= \max\{s_i - e_j, 0\}, \end{aligned}$$

where $j = \arg\max_{0 \le k \le N}\{e_k : s_k < s_i\}$.

Proof. Suppose that there exists an interval j' such that $s_{j'} < s_i$ and $e_{j'} < e_j$, so that pairing interval i with j' would produce a larger gap than pairing i with j.

If $e_{j'} \ge s_i$, then no gap interval ending at time $s_i - 1$ is formed. Otherwise, if

$e_{j'} < s_i$, we consider two subcases. If $e_j \geq s_i$, this configuration does not yield any gap interval that ends at time $s_i - 1$. If not, e_j lies between $e_{j'}$ and s_i, and thus $[e_{j'}, s_i - 1]$ is not a gap.

■

Based on the above observations, we apply the following algorithm. We first sort the intervals by their left endpoint. We then scan them while maintaining the maximum right endpoint seen so far, denoted by $e_{\max}$. For every $i \in \{1, 2, \ldots, N+1\}$, we compute ℓ_i and update $e_{\max}$ accordingly. The final answer is given by $\max_{1 \leq i \leq N+1}\{\ell_i\}$.

The overall time complexity is $\mathcal{O}(N \cdot \log N)$ for sorting the intervals, followed by $\mathcal{O}(N)$ for the linear scan. Hence, the total complexity is dominated by the sorting step and is $\mathcal{O}(N \cdot \log N)$.

Indicative code

C. mansion

Author: Josephina Antoniou.
Analysis: Anastasis Koliopoulos.

Tags: Graphs, SSSP, Dijkstra's algorithm, 0/1 BFS.

We model the problem as follows. Let $G = (V, E)$ be a directed graph with $|V| = N$ and $|E| = M$, where each edge $e \in E$ has weight $w(e) \in \{0, 1\}$. The goal is to find the shortest path from vertex 1 to vertex N, or determine that no such path exists.

Since all edge weights are non-negative, we can apply *Dijkstra's algorithm* to compute the shortest path from vertex 1 to vertex N. Implementing Dijkstra's algorithm using an *std::set* or an *std::priority_queue* yields a time complexity of $\mathcal{O}((N + M) \cdot \log N)$, which is sufficient to pass all the test cases. Using a *Fibonacci heap* we can reduce the time complexity to $\mathcal{O}(N \cdot \log N + M)$, but its complex implementation and its large constant factor makes it impractical for competitive programming contests.

Actually, since all edge weights are either 0 or 1, we can instead use the more efficient *0/1 BFS* algorithm, which avoids the logarithmic factor and runs in $\mathcal{O}(N + M)$ time.

Indicative code

D. packages

Author: Panagiotis Karelis.
Analysis: Anastasis Koliopoulos.

Tags: Binary search, k-th element.

We define the function $c : \{0, 1, \ldots, N\} \to \mathbb{N}$, where $c(k)$ denotes the minimum cost to deliver k packages. We make the following crucial observation.

Observation. The cost function is non-decreasing.

Proof. We'll show that $c(k) \leq c(k+1)$, where $0 \leq k \leq N-1$. We denote by I_{k+1} the set of indices of the packages selected in an optimal solution with $k+1$ packages. For every $i \in I_{k+1}$ we have

$$\begin{aligned} k < k+1 &\implies k \cdot B[i] \leq (k+1) \cdot B[i] && (B[i] \geq 0) \\ &\implies A[i] + k \cdot B[i] \leq A[i] + (k+1) \cdot B[i]. \end{aligned}$$

Writing the above inequality for every i, we get

$$\left.\begin{aligned} A[i_1] + k \cdot B[i_1] &\leq A[i_1] + (k+1) \cdot B[i_1] \\ A[i_2] + k \cdot B[i_2] &\leq A[i_2] + (k+1) \cdot B[i_2] \\ &\vdots \\ A[i_{k+1}] + k \cdot B[i_{k+1}] &\leq A[i_{k+1}] + (k+1) \cdot B[i_{k+1}] \end{aligned}\right\} \xRightarrow{+} \sum_{i \in I_{k+1}} A[i] + k \cdot B[i] \leq c(k+1).$$

Subtracting $A[j] + k \cdot B[j]$ from the left-hand side, for some $j \in I_{k+1}$, we obtain

$$\underbrace{\sum_{i \in I_{k+1} \setminus \{j\}} A[i] + k \cdot B[i]}_{\text{a feasible } k\text{-packages solution}} \geq c(k).$$

Combining the two inequalities above, we get

$$c(k) \leq c(k+1),$$

which completes the proof. ∎

Leveraging the monotonicity of the cost function, we can *binary search the answer*. We choose a value $k \in \{0, 1, \ldots, N\}$ and compute $c(k)$. If $c(k) \leq X$, where X denotes the available budget, we continue searching among values $k' \geq k$; otherwise, we restrict the search to values $k' < k$.

To determine which packages to deliver for a fixed k, we create the array $C[1 \dots n]$, where

$$C[i] = A[i] + k \cdot B[i]$$

and retrieve the k cheapest packages. There are three ways to obtain the desired packages, described below:

- Sort the array C in $\mathcal{O}(N \cdot \log N)$ time.
- Use the built-in C++ function *nth_element*, which runs in $\mathcal{O}(N)$ time on average.
- Use the *Median of medians algorithm*, which guarantees $\mathcal{O}(N)$ time in the worst case.

The binary search contributes an additional $\mathcal{O}(\log N)$ factor.

Overall, the best achievable time complexity is $\mathcal{O}(N \cdot \log N)$ in the worst case. Nevertheless, all of the above approaches are efficient enough to pass the given constraints. In the provided implementation, we use the second approach.

Indicative code

E. javelins

Author: Panagiotis Karelis.
Analysis: Anastasis Koliopoulos.

Tags: Exchange arguments, DP, State rotation, Space optimization.

We will apply *Dynamic Programming (DP)* to solve this problem.

Before discussing the DP solution, observe that the order in which the targets are hit matters. Therefore, it is not sufficient to decide only which subset of targets to select, but also in which order to hit them.

Let I, where $I \subseteq [N]$ and $|I| \geq 2$ (otherwise it's trivial), denote the set of the selected targets. Suppose we select two targets $i, i' \in I$. When is it optimal to hit target i before target i'?

Let j, j', with $1 \leq j < j' \leq |I|$, denote two time slots corresponding to the order in which the selected targets will be hit.

If we first assign target i to slot j and then target i' to slot j', the energy consumption is given by

$$E = A[i] + (j-1) \cdot B[i] + A[i'] + (j'-1) \cdot B[i'].$$

If we swap the order in which we hit these two targets, the energy consumption is equal to

$$E' = A[i] + (j'-1) \cdot B[i] + A[i'] + (j-1) \cdot B[i'].$$

Solving the inequality below, we get

$$\begin{aligned} E < E' &\iff (j-1) \cdot B[i] + (j'-1) \cdot B[i'] < (j'-1) \cdot B[i] + (j-1) \cdot B[i'] \\ &\iff (j-1) \cdot B[i] - (j'-1) \cdot B[i] < (j-1) \cdot B[i'] - (j'-1) \cdot B[i'] \\ &\iff (j-j') \cdot B[i] < (j-j') \cdot B[i'] \\ &\iff B[i] > B[i']. \end{aligned}$$

This means that we hit target i before i' if and only if $B[i] > B[i']$.

Let's now determine the optimal set of targets using DP, after we sort the targets in decreasing order of $B[i]$. The five stages of dynamic programming are specified as follows.

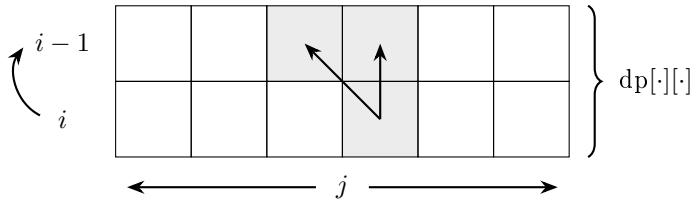
- (Definition of the DP table). We define dp[i][j], where $0 \leq i, j \leq N$, as the minimum energy required to hit j out of the first i targets.
- (Choices). For a target i, we decide whether to skip this target or to include it as the j-th hit target.

- (Recurrence relation). The following recurrence is derived from the above choices.

$$\mathrm{dp}[i][j] = \begin{cases} 0, & i = 0 \text{ and } j = 0, \\ \infty, & i = 0 \text{ and } 1 \le j \le N, \\ \mathrm{dp}[i-1][0], & 1 \le i \le N \text{ and } j = 0, \\ \min \begin{cases} \mathrm{dp}[i-1][j], \\ \mathrm{dp}[i-1][j-1] + A[i] + B[i] \cdot (j-1), \end{cases} & 1 \le i \le N \text{ and } 1 \le j \le N. \end{cases}$$

- (Order of computation). The values are computed in increasing order of i.
- (Original problem solution). The solution is $\max_{0 \le j \le N} \{j : \mathrm{dp}[N][j] \le E\}$.

The final observation is that, in order to compute any DP value in row i, we only need to refer to row $i-1$. Therefore, at any step, instead of storing the entire DP table in memory, we can keep only the current and the previous row.



The time complexity of computing this DP recurrence is $\mathcal{O}(N^2)$, and, after applying the space optimization described above, the space complexity becomes $\mathcal{O}(N)$.

Note. Since the problem is formulated in terms of maximizing the answer, it is more intuitive for the DP recurrence relation to follow the same formulation. In fact, we could formulate a maximization-based recurrence relation as well; however, it would lead to a higher time complexity than the current approach. To avoid this, we applied *state rotation*.

Indicative code

F. audit

Author: Panagiotis Karelis.
Analysis: Anastasis Koliopoulos.

Tags: Offline queries, Sparse Segment Tree.

Let's look at a simpler problem first. The statement is as follows.

Mantain a multiset of non-negative integers, denoted by S, that supports the following operations.

- INSERT(S, a) : Inserts the integer a into S.
- REMOVE(S, a) : Removes exactly one occurrence of a from S. It is guaranteed that a appears at least once in S.
- K-TH(S, k) : Returns the k-th smallest element of S. It is guaranteed that $1 \leq k \leq |S|$.

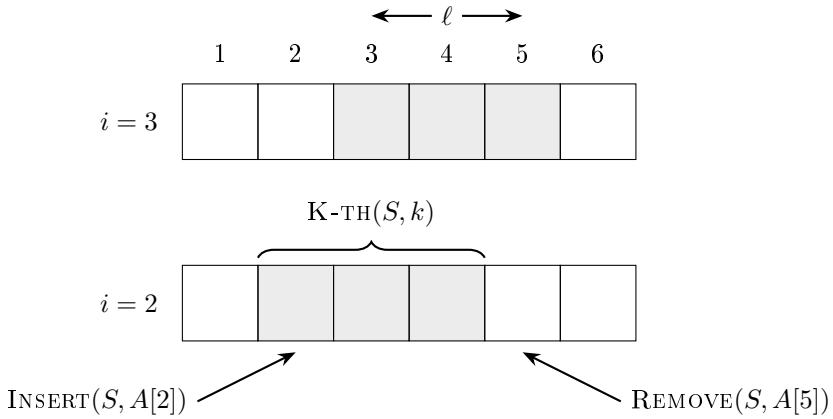
To handle these operations efficiently, we will use a *Sparse Segment Tree*. The structure of the tree and the INSERT operation are illustrated here. For the REMOVE operation, instead of adding 1 to each node along the path to the corresponding value, we simply subtract 1. Finally, for the K-TH operation, we can find the value corresponding to the k-th 1 using the method described here, commonly known as *Walking the Segment Tree*.

All of these operations traverse a path from the root to a leaf in our Segment Tree, performing a constant number of operations at each level. Suppose that the maximum value appearing in the tree is u. Then, the maximum depth of a leaf is $\lceil \log u \rceil$. Therefore, each operation takes $\mathcal{O}(\log u)$ time, giving a total complexity of $\mathcal{O}(Q \cdot \log u)$ for Q operations.

Let's now return to our original problem. For a fixed query length ℓ, we traverse the array from right to left while maintaining the set S. When we reach an index i, we remove $A[i + \ell]$ from S, if $i + \ell \leq N$, and insert $A[i]$ into S. For every query whose range starts at i, we can simply call:

- K-TH$(S, 1)$ to find the minimum,
- K-TH(S, k) to find the k-th smallest element, and
- K-TH(S, ℓ) to find the maximum.

Observe that S always contains the elements of a *sliding window* of size ℓ, as illustrated in the figure below.



We can map each query to its starting index during a preprocessing step. Since there are only two distinct window lengths, we can run the algorithm described above twice, once for each window length.

Reading the input array takes $\mathcal{O}(N)$ time, while reading the queries and performing the preprocessing step takes $\mathcal{O}(Q)$ time. Running the algorithm for both window lengths takes $\mathcal{O}\left(N \cdot \log \max_i\{a_i\}\right)$ time, since in this case $u = \max_i\{a_i\}$. Thus, the total time complexity is $\mathcal{O}\left(N \cdot \log \max_i\{a_i\} + Q\right)$.

Note. Our algorithm is *offline*, meaning that we know all the queries in advance, allowing us to preprocess the data and answer the queries in any order. However, if we had to answer the queries *online* and for arbitrary ranges $[\ell, r]$, with the length of the range not fixed, we would need to use more sophisticated data structures, like *Persistent Segment Trees* or *Wavelet Trees*.

Indicative code

G. windowmanager

Author: Konstantinos Tsichlas.
Analysis: Panagiotis Megas.

Tags: 2D Segment Tree.

First, we consider how to mark the topmost window. We need to determine the number of segments required to draw all visible window borders. The important difficulty is that a border of a window is visible only where no window with a larger z-coordinate covers it. Since all z-coordinates are distinct, every point belongs to a unique topmost window. The challenge is determining efficiently which window is on top at every point.

Consider a point (x, y). Among all windows containing it, we are interested only in the one with maximum z. For every window

$$[x_1, x_2] \times [y_1, y_2]$$

we therefore need to perform a rectangle update:

$$[x_1, x_2] \times [y_1, y_2] \leftarrow \max\{\text{currentValue}, z\}.$$

After all updates, a point query should return the z-coordinate of the topmost window containing that point. A natural data structure for this is a *segment tree of segment trees (2D Segment Tree)*. The outer segment tree handles the x-coordinate, while every node contains another segment tree handling the y-coordinate.

Alternatively, we could use a sweep-line algorithm to determine the topmost window at every point.

The next step is to double the grid. Currently, our data structure tracks discrete points, while the problem asks us to find continuous edges. This mismatch leads to overlap issues. As shown in the figure below, for window B, only a single point remains visible in the second row, leaving us with insufficient information to determine if an actual edge segment exists there.

We use

$$x' = 2x, \qquad y' = 2y$$

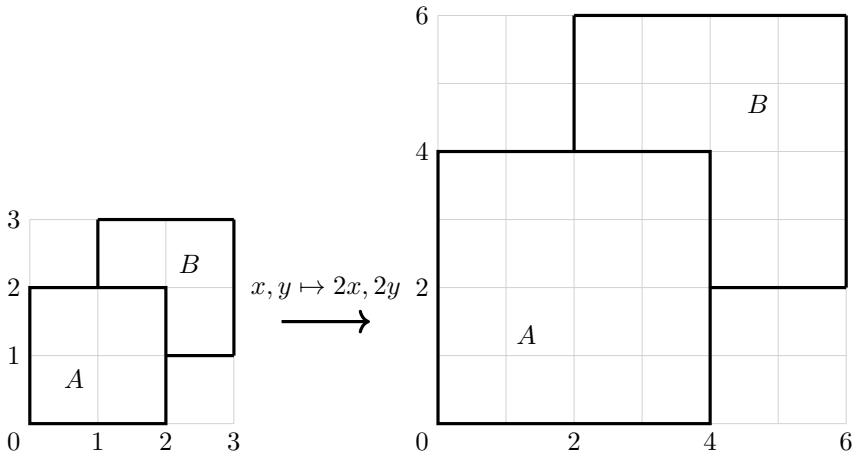
and thus, an original window

$$[x_1, x_2] \times [y_1, y_2]$$

becomes

$$[2x_1, 2x_2] \times [2y_1, 2y_2].$$

This solves the issue because it effectively gives every edge its own buffer cell. It creates spatial separation, ensuring that intersecting horizontal and vertical borders do not overwrite each other at shared corners.



Alternatively, we could avoid doubling the grid by explicitly marking the **edges** contained in each window.

We now proceed to find the visible borders. For every point (x, y), let v be the top window containing that point, found using our data structure. We check whether the point lies on one of the four borders of v. If

$$x = x_1(v) \quad \text{or} \quad x = x_2(v),$$

we mark the point as belonging to a vertical border. Similarly, if

$$y = y_1(v) \quad \text{or} \quad y = y_2(v),$$

we mark it as belonging to a horizontal border. Thus, after processing the whole grid, we know exactly which points belong to visible horizontal and vertical borders.

Finally, after all visible border points have been marked, we need to convert these marked points into segments. Consider one row. All marked points belonging to horizontal borders form consecutive runs. Every maximal consecutive run corresponds to exactly one horizontal segment. Therefore, we scan every row from left to right and maintain whether we are currently inside a run. Whenever a marked run ends, we increment the answer. We do the same for vertical segments by scanning every column from top to bottom.

The time complexity of the above procedure is calculated as follows. Let

$C = 2001$. There are $\mathcal{O}(N)$ rectangle updates, each taking $\mathcal{O}(\log^2 C)$ time. Naively, querying all C^2 grid points would take $\mathcal{O}(C^2 \cdot \log^2 C)$ time. However, we can resolve the whole grid in $\mathcal{O}(C^2)$ time by walking the segment tree and propagating the maximum values values to its children. Lastly, the two grid scans take $\mathcal{O}(C^2)$ time. Therefore, the total time complexity is $\mathcal{O}(N \cdot \log^2 C + C^2)$.

Indicative code

H. xorpairs

Author: Konstantinos Anemozalis.
Analysis: Anastasis Koliopoulos, Panagiotis Megas.

Tags: Simple math, Digit DP.

We define $D(x) = \langle d_1(x), d_2(x), \ldots, d_{\ell(x)}(x) \rangle$ as the sequence that represents number $x \in \mathbb{N}$ in decimal notation. For instance, if $x = 7385$ then $D(7385) = \langle 5, 8, 3, 7 \rangle$. Notice that in the sequence the digits are listed from right to left. Additionally, $\ell(x)$ denotes the number of digits of x in decimal notation.

Given that

$$g(x) = \bigoplus_{i=1}^{\ell(x)} d_i(x)$$

and

$$f(a,b) = \sum_{x=\min\{a,b\}}^{\max\{a,b\}} g(x),$$

we aim to compute

$$ans = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} f(A[i], A[j]).$$

After sorting A, we obtain

$$\begin{aligned}
ans &= \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \sum_{x=A[i]}^{A[j]} g(x) \\
&= \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \left(\sum_{x=1}^{A[j]} g(x) - \sum_{x=1}^{A[i]-1} g(x) \right) \\
&= \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \sum_{x=1}^{A[j]} g(x) - \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \sum_{x=1}^{A[i]-1} g(x) \\
&= \sum_{j=2}^{N} \sum_{i=1}^{j-1} \sum_{x=1}^{A[j]} g(x) - \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \sum_{x=1}^{A[i]-1} g(x) \\
&= \sum_{j=2}^{N} \sum_{x=1}^{A[j]} g(x) \sum_{i=1}^{j-1} 1 - \sum_{i=1}^{N-1} \sum_{x=1}^{A[i]-1} g(x) \sum_{j=i+1}^{N} 1 \\
&= \sum_{j=2}^{N} \sum_{x=1}^{A[j]} g(x) \cdot (j-1) - \sum_{i=1}^{N-1} \sum_{x=1}^{A[i]-1} g(x) \cdot (N-i).
\end{aligned}$$

The main computational bottleneck in calculating *ans* efficiently, is the evaluation of $\sum_{x=1}^{A[j]} g(x)$ and $\sum_{x=1}^{A[i]-1} g(x)$.

Let us consider calculating that value for a $y \in \mathbb{N}$. What we need to compute is the sum of $g(x)$ for all $x \in \{1, \ldots, y\}$. By listing $g(x)$ for each x, we observe that the numbers can be grouped according to the value of $g(x)$.

Another crucial observation is that $0 \leq g(x) \leq 15 = g_{\max}$. This follows from the fact that the XOR operation is applied only to digits from the set $\{0, \ldots, 9\}$, each of which requires at most 4 bits to be represented. As a direct consequence, there are only 16 groups.

The final observation is that, since $0 \leq A[i] \leq 10^9$, we have $0 \leq \ell(y) \leq 9$, for any i such that $A[i] = y$.

We can therefore count the number of elements in each group by constructing the corresponding numbers using *Digit DP*. The five stages of dynamic programming are specified as follows.

- (Definition of the DP table). We define $\mathrm{dp}[i][\bar{x}][\delta]$, where $0 \leq i \leq \ell(y)$, $0 \leq \bar{x} \leq g_{\max}$ and $\delta \in \{0, 1\}$, as the number of i-digit numbers with XOR $\bar{x}$ that are less than or equal to y (0 denotes the equal and 1 the smaller numbers).
- (Choices). The choice determines which digit is selected for position i.
- (Recurrence relation). The recurrence relation for $\delta \in \{0, 1\}$ is given below.

$$\mathrm{dp}[i][\bar{x}][0] = \begin{cases} 1, & i = 0 \text{ and } \bar{x} = 0, \\ 0, & i = 0 \text{ and } 1 \leq \bar{x} \leq g_{\max}, \\ \sum_{d=0}^{d_i(y)-1} \mathrm{dp}[i-1][\bar{x} \oplus d][1] + \mathrm{dp}[i-1][\bar{x} \oplus d_i(y)][0], & 1 \leq i \leq \ell(y) \text{ and } 0 \leq \bar{x} \leq g_{\max}. \end{cases}$$

$$\mathrm{dp}[i][\bar{x}][1] = \begin{cases} 1, & i = 0 \text{ and } \bar{x} = 0, \\ 0, & i = 0 \text{ and } 1 \leq \bar{x} \leq g_{\max}, \\ \sum_{d=0}^{9} \mathrm{dp}[i-1][\bar{x} \oplus d][1], & 1 \leq i \leq \ell(y) \text{ and } 0 \leq \bar{x} \leq g_{\max}. \end{cases}$$

- (Order of computation). The values are computed in increasing order of i.

- (Original problem solution). For a specific y, the result is

$$C(y) = \begin{cases} \sum_{\bar{x}=0}^{g_{\max}} \mathrm{dp}[\ell(y)][\bar{x}][0] \cdot \bar{x}, & y \geq 0, \\ 0, & y < 0. \end{cases}$$

By setting $i = j$, we can now write ans as

$$\sum_{i=2}^{N} C(A[i]) \cdot (i-1) - \sum_{i=1}^{N-1} C(A[i]-1) \cdot (N-i).$$

To compute the above quantity, we require $2 \cdot N - 2$ evaluations of the function C, each of which has a cost of at most $\max_i\{\ell(A[i])\} \cdot g_{\max} \cdot 2 \cdot 9$. Including the cost of sorting, we obtain an overall time complexity of $\mathcal{O}(N \cdot \log N + N \cdot \max_i\{\ell(A[i])\} \cdot g_{\max})$.

Indicative code

I. runnerups

Author: Lazaros Karageorgiou.
Analysis: Anastasis Koliopoulos, Panagiotis Megas, Diomedes Varvaresos.

Tags: Segment Tree.

For each $i \in \{1, 2, \ldots, N\}$, we define

$$\ell(i) = \max_{1 \le j < i} \{j : A[j] > A[i]\}, \qquad \ell'(i) = \max_{1 \le j < \ell(i)} \{j : A[j] > A[i]\},$$

$$r(i) = \min_{i < j \le N} \{j : A[j] > A[i]\}, \qquad r'(i) = \min_{r(i) < j \le N} \{j : A[j] > A[i]\}.$$

The edge cases of the above definitions can be handled easily.

We say that an element i *influences* the answer to query $Q = (p_1, p_2, \ldots, p_K)$, if and only if there exist a, b, with $1 \le a < b \le K$, such that $A[i]$ is the second-largest element in the interval $[p_a, p_b]$. Let us now examine the conditions that p_a and p_b must satisfy for this statement to hold.

First, we know for certain that i must lie between the two. Equivalently, we must have

$$p_a \le i \le p_b.$$

Let us now consider a possible choice for p_a. By the definition of $\ell(i)$ and $\ell'(i)$, it follows that

$$p_a \ge \ell'(i) + 1,$$

since, if the opposite inequality held, then, regardless of the choice of p_b, $A[i]$ would not be the second-largest element in the interval $[p_a, p_b]$. We have that

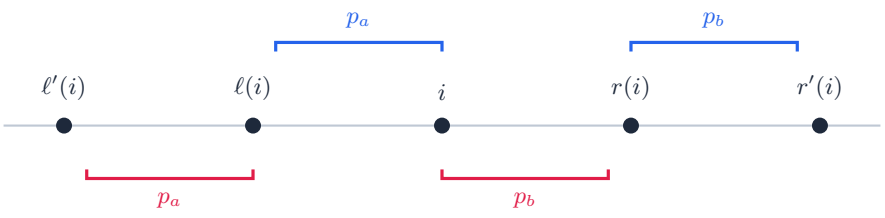
$$\ell'(i) + 1 \le p_a \le i \iff \ell'(i) + 1 \le p_a \le \ell(i) \vee \ell(i) + 1 \le p_a \le i$$

and therefore, if the first case holds for p_a, we must have

$$i \le p_b \le r(i) - 1,$$

whereas if the second case holds, we must have

$$r(i) \le p_b \le r'(i) - 1.$$



Currently, we have to check the influence for N elements. We can reduce that by making the following observation.

Observation. The number of elements that influence the answer is $\mathcal{O}(K)$.

Proof. For an element to contribute to the answer, it must be either the largest or the second-largest element in the interval $[p_i, p_{i+1}]$, for some $i \in \{1, \ldots, K-1\}$. Suppose that there exists a j, where $p_1 \leq j \leq p_K$, that does not satisfy the latter property, but contributes to the answer.

Since $p_i \leq j \leq p_{i+1}$ for some i, every interval $[p_a, p_b]$ with $1 \leq a \leq i$ and $i+1 \leq b \leq K$ contains $[p_i, p_{i+1}]$. Therefore, if at least two elements in $[p_i, p_{i+1}]$ are larger than $A[j]$, the same holds for $[p_a, p_b]$, so $A[j]$ cannot be among its two largest elements.

Thus, by constructing the **set** of elements that are either the largest or the second-largest in each interval $[p_i, p_{i+1}]$, we ensure that this set contains all elements that may contribute to the answer of the query.

We have that

$$\text{numberOfInfluentialElements} \leq \sum_{i=1}^{K-1} 2 = 2(K-1) \leq 2K \in \mathcal{O}(K).$$

■

First, we build a maximum Segment Tree in the array A. We can compute the values $\ell'(i), \ell(i), r(i)$, and $r'(i)$, by walking the Segment Tree, in a total time of $\mathcal{O}(N \cdot \log N)$. For each query, we maintain a set containing the largest and second-largest elements of each interval described above. This can be achieved using the same Segment Tree, with appropriate max queries, in a total time of $\mathcal{O}(K \cdot \log N)$. For each element i belonging to the set, we must find all pairs (a, b) such that $A[i]$ is the second-largest element in $[p_a, p_b]$. Since $p_i < p_{i+1}$, we can use binary search to determine the ranges of possible values of p_a and p_b that satisfy the required conditions, and then multiply their sizes. This requires a total time of $\mathcal{O}(K \cdot \log K)$.

The time complexity of answering all queries is $\mathcal{O}(\sum_i K_i \cdot \log N)$ (if we exclude the preprocessing step), which is acceptable since $\sum_i K_i \leq 5 \cdot 10^5$. Thus, the overall time complexity is $\mathcal{O}((N + \sum_i K_i) \cdot \log N)$.

Indicative code

* We would also like to thank the judges, Apostolos Giannoulidis and Pavel Mavrin, for their contribution to the creation of the problems.

** This editorial may contain mistakes. If you notice any, send an email to anastasiskol@gmail.com.