

BdOI Monthly Contest Round 3 Solutions:

Problem Chicken Farm

Problem Information

Problem Author: Antar Banik

Problem Preparation: Antar Banik

Editorial: Antar Banik

Basic Idea & Overflow Handling

Let $\text{pref}[i] = \sum_{j=1}^i A[j]$ be the prefix sum of the first i food boxes, and let $S_{\text{total}} = \text{pref}[n]$ be the total amount of food across all n boxes. For a given query with k chickens and ratio $B[1], B[2], \dots, B[k]$, let the sum of the ratios be:

$$C = \sum_{j=1}^k B[j]$$

Since the food must be divided in the exact ratio $B[1] : B[2] : \dots : B[k]$, the x -th chicken must receive an exact amount of food equal to:

$$B'[x] = \frac{\text{pref}[n] \times B[x]}{C}$$

Equivalently, if we let $\text{pref}_B[x] = \sum_{j=1}^x B[j]$ be the x -th prefix sum of the ratio array, then the x -th chicken's segment must end at some box index idx_x such that the prefix sum $\text{pref}[idx_x]$ equals:

$$\text{Target}[x] = \frac{\text{pref}[n] \times \text{pref}_B[x]}{C}$$

Avoiding 64-bit Integer Overflow:

In the worst-case scenario, $\text{pref}[n]$ can be up to 10^{14} and $B[x]$ (or C) can be up to 10^{14} . Multiplying $\text{pref}[n] \times \text{pref}_B[x]$ directly can reach 10^{28} , which overflows a standard 64-bit integer (`long long` in C++).

To compute $\frac{a \times b}{c}$ safely without overflow (where $a = \text{pref}[n]$, $b = \text{pref}_B[x]$, and $c = C$):

1. First, reduce the fraction $\frac{a}{c}$ by dividing both a and c by $g_1 = \text{gcd}(a, c)$:

$$a' = \frac{a}{\text{gcd}(a, c)}, \quad c' = \frac{c}{\text{gcd}(a, c)}$$

2. Next, reduce $\frac{b}{c'}$ by dividing both b and c' by $g_2 = \text{gcd}(b, c')$:

$$b' = \frac{b}{\text{gcd}(b, c')}, \quad c'' = \frac{c'}{\text{gcd}(b, c')}$$

3. Since the required food amount must be a valid integer, if $c'' \neq 1$, then $a \times b$ is not divisible by c , so we can immediately output `NO`.
4. Otherwise, $c'' = 1$, and the required target sum is simply $a' \times b'$. Because the true target cannot exceed $\text{pref}[n] \leq 10^{14}$, this product is guaranteed to fit inside a standard 64-bit integer (`long long`).

(Note: Alternatively, one can reduce the ratio array $B[1 \dots k]$ beforehand by dividing all $B[i]$ by $g = \text{gcd}(B[1], B[2], \dots, B[k])$. Then $C' = \sum(B[i]/g)$ must divide $\text{pref}[n]$; if $\text{pref}[n] \bmod C' \neq 0$, the answer is `NO`. Otherwise, the base unit size is $U = \text{pref}[n]/C'$, and the x -th chicken simply needs $U \times (B[x]/g)$ units of food without any overflow.)

Subtask Analysis

Subtask 1: $n \leq 1000, q = 1, k = 2$

We only have a single query with $k = 2$. We can iterate through all possible split points i ($1 \leq i < n$) and check if the prefix sum $\text{pref}[i]$ and suffix sum $\text{pref}[n] - \text{pref}[i]$ satisfy the ratio $B[1] : B[2]$ (i.e., whether $\text{pref}[i]$ equals the expected target sum for the first chicken). If a valid i is found, we print `YES` followed by i and n ; otherwise, we print `NO`.

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

Subtask 2: $n \leq 1000, k = 3, \sum k \leq 1000$

For each query, we need to split the array into $k = 3$ contiguous parts. After computing the required prefix targets $\text{Target}[1]$ and $\text{Target}[2]$, we can simply scan the array from 1 to n in $\mathcal{O}(n)$ time to find the indices $idx_1 < idx_2 < n$ where $\text{pref}[idx_1] = \text{Target}[1]$ and $\text{pref}[idx_2] = \text{Target}[2]$.

Time Complexity: $\mathcal{O}(n \cdot q)$ or $\mathcal{O}(n \sum k)$

Subtask 3: $n \leq 10^5, k = 2, \sum k \leq 10^5$

Since n and q can be up to 10^5 , an $\mathcal{O}(n)$ scan per query will Time Limit Exceed (TLE). However, since $A[i] \geq 1$, the prefix sum array $\text{pref}[1 \dots n]$ is strictly increasing. For each query with $k = 2$, we compute $\text{Target}[1]$ and use **Binary Search** (`std::lower_bound`) on the prefix sum array to check if $\text{Target}[1]$ exists at some index $idx_1 < n$.

Time Complexity: $\mathcal{O}(n + q \log n)$

Subtask 4 & 5 (Full Solution): $n, q \leq 10^5, k \leq n, \sum k \leq 10^5$

We first precompute the strictly increasing prefix sum array $\text{pref}[1 \dots n]$ in $\mathcal{O}(n)$ time.

For each query of size k :

1. Compute the total ratio sum $C = \sum_{j=1}^k B[j]$.
2. For each x from 1 to k , maintain the running ratio prefix sum $\text{pref}_B[x] = \sum_{j=1}^x B[j]$ and calculate the expected prefix sum $\text{Target}[x] = \frac{\text{pref}[n] \times \text{pref}_B[x]}{C}$ using the overflow-safe GCD reduction method described above.
3. If the division is not exact for any x , or if $\text{Target}[x]$ is not found in the pref array using binary search (`std::lower_bound`), then a valid split is impossible and we output **NO**.
4. Since $A[i] > 0$ and $B[x] > 0$, the target prefix sums are strictly increasing ($\text{Target}[1] < \text{Target}[2] < \dots < \text{Target}[k] = \text{pref}[n]$), which automatically guarantees that the found ending indices $idx_1 < idx_2 < \dots < idx_k = n$ are strictly increasing and form non-empty contiguous segments.

If all k targets exist in pref , we print **YES** and the corresponding 1-based indices $idx_1, idx_2, \dots, idx_k$.

Time Complexity: $\mathcal{O}(n + \sum k(\log n + \log(\max B)))$

Space Complexity: $\mathcal{O}(n)$

Dictator Editorial

Fahmid Hasan

Problem statement can be simplified as:

Given a tree of size n and an integer array A of length n . Say, $u_1, u_2, u_3, \dots, u_k$ are all the nodes that are present in the path from u_1 to u_k in order. Then, we define **beauty** of the path from u_1 to u_k as $\sum_{i=1}^k \gcd(A_{u_1}, A_{u_2}, \dots, A_{u_i})$. Calculate the maximum beauty of a path.

Subtask-1 (The Star Graph):

In this subtask, the tree is a star graph centered at node r . Thus, any path can have at most 3 nodes. First, detect the node r i.e. the node which has degree $n - 1$.

Then take a map mp and variable $ans := A_r$. Here, $mp[g]$ will store the maximum value of A_p for $p \neq r$ such that $\gcd(A_p, A_r) = g$.

Now, to create mp , for every node $i \neq r$, let $g = \gcd(A_i, A_r)$:

- First, consider the 2-node paths between i and r : update $ans := \max(ans, \max(A_i, A_r) + g)$.
- update $mp[g] = \max(mp[g], A_i)$
- Finally, if we previously had key g in mp , then update $ans := \max(ans, mp[g] + 2g)$ considering the path $i \rightarrow v$ and $v \rightarrow i$ where u, v, i are pairwise distinct and $\gcd(A_v, A_r) = g$

Now, we need to consider the 3-node paths $u \rightarrow v$ where $\gcd(A_u, A_r) \neq \gcd(A_v, A_r)$. To do this, for every key g in mp , iterate through every key g' in mp and if $g \neq g'$, then update $ans := \max(ans, mp[g] + g + \gcd(g, g'))$

Since g always divides A_r and $A_r \leq 10^4$, the map mp will have very few elements (at most 64). By now, we have considered all possible paths of length 1, 2, and 3, so we can just print ans .

Subtask-2 ($n \leq 10^3$) :

We try every vertex as the root of the tree.

For a fixed root r , run a DFS from r . During the DFS, for every node u , maintain the GCD and beauty of the path from r to u . We update the answer with the maximum beauty encountered.

Every path in the tree has two endpoints. If we choose one endpoint as the root, that path becomes a root-to-node path. Therefore, by trying every node as the root, we consider every possible path.

For each of the n choices of the root, the DFS takes $O(n)$ time. Hence, $O(n^2)$ time complexity in total.

Subtask-3 (The Intended Solution) :

Here, we will also try every root. But, instead of running a DFS every time we change the root, we will store data in a way such that we only need to change data for previous root and new root. In general, this idea is called **Reroot DP**.

For the current root, we define $s[u]$ to be a map from GCD g to a multiset. For a fixed g , the multiset contains at most one value contributed by each child ch of u : the maximum beauty among all the paths that start at a node in the subtree of ch , ends at u and has GCD g . Additionally, if $A[u] == g$, we insert g in $S[u][g]$ corresponding to the single vertex path $\{u\}$.

First, we set vertex 0 as the root of the tree and run a bottom-up DFS to calculate s .

When we are at node u , we update $s[u]$. To do that, we take an array mx where every value is initially set to 0. Then, for every child ch , we iterate over every GCD g in $s[ch]$ and update $mx[g1] :=$

$\max(mx[g1], mx.beauty + g1)$ where $g1 = \gcd(g, A[u])$ and $mx.beauty$ is the maximum beauty in $s[ch][g]$. Then, we iterate over i where $mx[i]$ is non-zero and insert those $mx[i]$ in $s[u][i]$ and set $mx[i] := 0$ again. Finally, we also insert $A[u]$ in $s[u][A[u]]$ for the single-node path $\{u\}$.

Now, we need to deal with the rerooting part i.e trying every node as root. We will do this using top-down DFS.

Let's say, right now, node r is the root and s is calculated assuming that. First, we update ans with the maximum beauty in $s[r]$ we have encountered so far. By doing this, we have updated our ans with the maximum beauty among all paths that end at node r .

Then, we need to try a new root. We will transfer our root from r to ch where ch is a child of r . So, we only need to update $s[r]$ and $s[ch]$. First, we need to delete contribution of ch from $s[r]$ as current subtree of ch won't lie inside the subtree of r after transferring the root. We can find the contribution of ch by using the same kind of transition we used during bottom-up DFS. Then, we delete those contributions from $s[r]$ and store pairs of integers $(g, beauty)$ that we deleted in a temporary array called par .

Now, $s[r]$ contains all possible paths reaching r without using ch . To update $s[ch]$, we need to append ch to the end of every such path. Again similar type of transition method used in the bottom-up DFS will work here. We insert the contributions in $s[ch]$ and store pairs of integers $(g, beauty)$ in a temporary array called $child$ that we inserted in $s[ch]$.

Hence, the root was successfully transferred from r to ch . Now, when we need our previous version of the tree, we can easily get that using the arrays par and $child$.

In the end, we can just print ans .

Complexity Calculation

Let

$$D = \max_{i=1}^n \tau(A_i)$$

$$MAX_A = \max_{i=1}^n A_i$$

Then each GCD stored in s_u , $g \mid A_u$. So, $|s_u| \leq \tau(A_u) \leq D$.

Also, for every child ch of u , we process at most D states from $s[u]$. Hence, $O(nD)$ transitions for both DFS.

In each transition,

- We calculate GCD which is of $O(\log MAX_A)$ time complexity
- We do a map query which is of $O(\log D) < O(\log MAX_A)$ time complexity
- We do a multiset query which is of $O(\log n)$ time complexity.

Hence, in total $O(nD(\log n + \log MAX_A))$ time complexity.

Similarly, space complexity is $O(nD)$.

BdOI Monthly Contest Round 3 Solutions: Problem Orders

Problem Information

Problem Author: Nabeul Islam

Problem Preparation: Nabeul Islam

Editorial: Nabeul Islam

Subtasks hinting key observations

Preliminary Observation:

A simplified version of the problem exists for $k = 1$, which can be solved using a min-heap priority queue. However, when $k > 1$ and the constraints on N and Q are up to 200,000, we must dynamically maintain contiguous segments of available pagers. This requires an optimal **Time Complexity** of $\mathcal{O}((N + Q) \log N)$.

Subtask 1: $N = 1$, $Q \leq 200,000$, $k = 1$

Since $N = 1$ and $k = 1$, there is only one pager in the entire restaurant, and it is guaranteed that every group arrival can be fulfilled. Therefore, for every Type 1 query, the answer is always 1.

Subtask 2: $N \leq 5,000$, $Q \leq 5,000$

For this subtask, the constraints are small enough to simulate the process directly in $\mathcal{O}(N \cdot Q)$ time. We can maintain a boolean array `avail[1...N]` initialized to true. For a Type 1 query, we scan from 1 to N to find the

first contiguous block of k available pagers and mark them false. For a Type 2 query, we iterate from l to r and set $\text{avail}[i] = \text{true}$. Alternatively, this subtask can also be solved using the full solution discussed in Subtask 5.

Subtask 3: $N \leq 200,000$, $Q \leq 200,000$, $k = 1$

This is the simplified version of the problem where every arriving customer requests only 1 pager. If there are no Type 2 queries, the i -th customer simply receives pager i . Even with Type 2 queries (where $l = r$), we can maintain all currently available pagers in a min-heap priority queue (`std::priority_queue`). For each Type 1 query, we extract and print the minimum available pager ID, and for each Type 2 query, we push the returned pager IDs back into the priority queue in $\mathcal{O}(\log N)$ time.

Subtask 4: $N \leq 200,000$, $Q \leq 200,000$, $k \leq 3$

Since $k \in \{1, 2, 3\}$, we can solve this subtask by maintaining three ordered sets (`std::set`): S_1, S_2 , and S_3 , where S_k stores all starting indices i such that pagers $i, \dots, i+k-1$ are all currently available. For a Type 1 query with group size k , we simply take the smallest index $l = *S_k.\text{begin}()$. Whenever a pager x changes its availability state, only the starting indices $x-2, x-1$, and x in S_1, S_2 , and S_3 are affected and need to be updated.

Subtask 5 (Full Solution)

For the full solution where $1 \leq k \leq N$, we use a **Segment Tree with Lazy Propagation**. Each leaf i represents the state of pager i (1 if available, 0 if occupied). For each node u covering the range $[L, R]$ of length $\text{len} = R - L + 1$, we maintain three values:

- $\text{pref}[u]$: the maximum number of consecutive available pagers starting from L .
- $\text{suff}[u]$: the maximum number of consecutive available pagers ending at R .
- $\text{best}[u]$: the maximum number of consecutive available pagers anywhere inside $[L, R]$.

When merging the left child lc (of length len_L) and right child rc (of length len_R), we update node u as follows:

- $\text{pref}[u] = (\text{pref}[lc] == \text{len}_L) ? \text{len}_L + \text{pref}[rc] : \text{pref}[lc]$
- $\text{suff}[u] = (\text{suff}[rc] == \text{len}_R) ? \text{len}_R + \text{suff}[lc] : \text{suff}[rc]$
- $\text{best}[u] = \max(\text{best}[lc], \text{best}[rc], \text{suff}[lc] + \text{pref}[rc])$

To answer a **Type 1** (k) query, we walk down the Segment Tree starting from the root covering $[L, R]$ with midpoint M :

1. If $\text{best}[lc] \geq k$, the earliest block of size k lies completely in the left child, so we recurse on lc .
2. Else if $\text{suff}[lc] + \text{pref}[rc] \geq k$, the earliest block crosses the midpoint M . The exact starting pager ID is $l = M - \text{suff}[lc] + 1$.
3. Otherwise, the block lies in the right child, so we recurse on rc .

Once we find l , we output the answer (l if $k = 1$, or l and $l+k-1$ if $k > 1$) and perform a lazy range update on $[l, l+k-1]$ to set all pagers in that segment to 0 (unavailable). For a **Type 2** (l, r) query, we perform a lazy range update on $[l, r]$ to set all pagers in that segment back to 1 (available). Both queries run in $\mathcal{O}(\log N)$ time, giving an overall time complexity of $\mathcal{O}((N+Q) \log N)$.