

BdOI Monthly Contest Round 2 Solutions:

Problem Number Game

Problem Information

Problem Author: Tahjibul Islam Robi

Problem Preparation: Tahjibul Islam Robi

Editorial: Tahjibul Islam Robi, MD Riasat Islam

Subtasks hinting key observations

Preliminary Observation:

We can divide the task into two main subproblems:

1. Getting the value of $f(a_i)$ from a_i .
2. Choosing the best strictly increasing subsequence which maximizes the sum of $f(a_i)$.

Subtask 1: $n = 1, a_i \leq 10^6$

Here $n = 1$ and $a_i \leq 10^6$. Since there is only one element, we will not have to think about the main second problem (which is choosing the best increasing subsequence).

To get the value of $f(a_i)$, we can use a Sieve to precompute every prime number under 10^6 . Let a prime number be p . If $a_i \pmod p == 0$, then p is the prime base of a_i . Now we can try every q from 1 to 59 and check if $p^q == a_i$. If $p^q == a_i$, the answer is simply q .

Subtask 2: $n \leq 16, q = 1$

Here $q = 1$ for all elements, meaning $f(a_i) = 1$. So we only have to find the longest increasing subsequence (LIS).

Since n is very small, we can try every possible subsequence using bitmasking very easily. We will loop a variable x from 1 to $2^n - 1$. For each x , if the i -th bit of x is set, we take the element a_i and push it to a new array b .

If the constructed array b is strictly increasing, it is a valid subsequence, and the answer for this possible subsequence is simply the size of array b . We just take the maximum possible size of array b over all valid cases.

Subtask 3: $n \leq 16, a_i \leq 10^6$

This subtask is a direct combination of Subtask 1 and Subtask 2.

We will use the bitmasking idea from Subtask 2 to generate all possible subsequences from array a . Let one possible subsequence be b . If b is strictly increasing, we will take the sum of all $f(b_i)$.

Because $a_i \leq 10^6$, we can use the Sieve idea from Subtask 1 to easily get the value of $f(a_i)$. The answer for this subtask is the maximum sum of $f(b_i)$ over all valid increasing subsequences.

Solving the First Main Problem ($a_i \leq 10^{18}$)

Before moving to any further subtasks, let's solve the first main problem completely: how will we get the value of $f(a_i)$ for very large numbers?

Let p be the prime base and q be the exponent such that $a_i = p^q$. First, we have to notice one crucial thing: $(10^6)^3 = 10^{18}$, and $(10^6 + 1)^3 > 10^{18}$. This means that for every prime number $p > 10^6$, its exponent q will be at most 2.

If $p \leq 10^6$, we can use our Sieve to precompute every prime number under 10^6 , and we will store every prime number and all of their valid powers up to 10^{18} in a C++ 'std::map'. The key of the map will be p^q , and the mapped value will be q .

If we can find a_i in our map, we immediately retrieve $f(a_i)$ from the value. If we *can't* find the value in our map, we can be absolutely sure that the prime number $p > 10^6$, meaning q is either 1 or 2. We can now try to find the square root of a_i using binary search (or the built-in 'sqrt' function). Let our computed square root be x . If $\lfloor x \rfloor^2 == a_i$, we can say that $q = 2$.

Otherwise, $q = 1$.

Note: There are other ways to get the value of q using the idea of n th-root! But that is not under the IOI syllabus, so this map-and-square-root approach is our intended solution.

Subtask 4: $n = 1$

Here $n = 1$, but a_i can be up to 10^{18} . We will just print the value of $f(a_0)$ using the map and binary search idea mentioned directly above.

Subtask 5: $n \leq 16$

Here $n \leq 16$ and $a_i \leq 10^{18}$. We will get the value of $f(a_i)$ using the large-number idea mentioned above, and then we will do bitmasking to check all subsequences using the same logic from Subtasks 2 and 3.

Subtask 6: $q = 1$

Here $q = 1$, so the size of the longest increasing subsequence is the final answer.

We can use the classic Binary Search LIS DP or a Segment Tree! Both of them work in $O(n \log n)$. You can find more about solving LIS efficiently here: https://cp-algorithms.com/dynamic_programming/longest_increasing_subsequence.html

Subtask 7: The Full Problem

Here, we will get the value of $f(a_i)$ for all elements using our precomputed map and square root trick. Let $\text{cost}[i] = f(a_i)$.

Now we can use Dynamic Programming to solve the subsequence problem. Let $dp[h]$ be the maximum exponent sum of a valid subsequence where the last element included has the exact value h . For each element a_i and its corresponding $\text{cost}[i]$:

1. We can extend any valid subsequence that ends at some value x , where $x < a_i$.
2. The optimal previous subsequence to extend will be the one with the maximum answer found in the range $[0, a_i - 1]$.

3. So the DP transition is: $dp[a_i] = \text{cost}[i] + \max(dp[x])$ for all $x \in [0, a_i - 1]$.

Since $n \leq 10^5$, a naive $O(n^2)$ loop for the maximum query (Step 3) will cause TLE. To optimize this, we will first coordinate-compress the values of array a to map them to a smaller range $[1, n]$. Then, we will use a Segment Tree over these compressed values to perform the range maximum queries and point updates in $O(\log n)$ time. This brings our final time complexity to $O(n \log n)$, comfortably passing the full problem!

BdOI Monthly Contest Round 2 Solutions:

Problem Wifi Password

Problem Information

Problem Author: Tahjibul Islam Robi

Problem Preparation: Tahjibul Islam Robi, Nabeul Islam

Editorial: Tahjibul Islam Robi, MD Riasat Islam, Nabeul Islam

Subtasks hinting key observations

Preliminary Observation:

$\text{MEX}(A)$ is defined as the smallest non-negative integer that is not present in the array A . For an index i , we are only allowed to perform at most one swap for every prefix.

A swap of $A[i]$ and $B[i]$ changes $\text{MEX}(A)$ in two cases:

Case 1: Removing a unique value which is smaller than current MEX

If $A[i] < \text{MEX}(A)$ (which is the current MEX) and $A[i]$ is unique in the prefix, swapping $A[i]$ will result in $A[i]$ (which we are swapping out) becoming the new MEX of A . In this case, $\text{MEX}(A)$ simply becomes $A[i]$. This very same logic also applies in the case of B . If $B[i] < \text{MEX}(B)$ and is unique, swapping $B[i]$ will change the MEX of B to $B[i]$.

Case 2: Adding a value where $B[i] = \text{MEX}(A)$

If $B[i] = \text{MEX}(A)$, then the swap adds the current missing number to A . As a result, the MEX then moves to the next missing number after $\text{MEX}(A)$, unless we remove the value $A[i]$ and $A[i]$ itself is unique and lies strictly

between $\text{MEX}(A)$ and that next missing number. In this case, $\text{MEX}(A) = A[i]$.

These two cases also apply for B . However, if neither condition is true, the $\text{MEX}(A)$ remains as it is (and vice versa for B).

Subtask 1: $n = 10$, $A[i] = B[i]$

Since $A[i] = B[i]$ for every prefix, swapping any pair leaves both arrays unchanged. So $\text{MEX}(A) = \text{MEX}(B)$ for every prefix, and the answer is always 0. This can be printed directly.

Subtask 2: $A[i]$ is even

This will require the full solution which is discussed later in the editorial.

Subtask 3: $n \leq 1000$

For each prefix, we can try every possible swap explicitly. We can maintain a final ‘output’ vector of length n (initialized to infinity) where the index j denotes the value of F for the prefix of length $j + 1$.

We can have at most $n + 1$ scenarios: either we do no swaps at all, or we do a swap on index 0, or index 1, up to index $n - 1$.

First, we naively run a simulation maintaining **curr_mexa** and **curr_mexb** without doing any swaps to tackle the baseline case. Then, for all i ($0 \leq i < n$), we explicitly perform a swap on index i by swapping $A[i]$ and $B[i]$. After the swap, we run a full pass from left to right simulating the MEX values. At each step j of the pass, we keep updating our output array:

$$output[j] = \min(output[j], |\text{curr_mexa} - \text{curr_mexb}|)$$

After this pass is complete, we undo the swap on index i (backtracking) and move on to index $i + 1$ to repeat the same process.

For a single pass, finding and maintaining the MEX (e.g., using a ‘std::set’) takes $O(n \log n)$ time. Since we do this for at most $n + 1$ scenarios, our overall final time complexity is $O(n^2 \log n)$, which easily passes for $n \leq 1000$.

Subtask 4: $n \leq 5000$

While $O(n^2 \log n)$ is fast enough for $n \leq 1000$, we need to optimize for $n \leq 5000$. Instead of fixing the swap and doing a pass, we fix the prefix and

evaluate all possible swaps dynamically in $O(1)$ time.

Suppose we are at prefix length k . If we consider swapping $A[i]$ and $B[i]$ (where $i < k$), we can determine its exact impact on the current MEX in $O(1)$ time without doing a full pass.

Let $x = A[i]$ and $y = B[i]$. Let the current MEX for the prefix be **mexA** and **mexB**, and let **nextA** and **nextB** be the second missing numbers in the prefix. The new MEX for A if this swap happens, let's call it **newmexA**, will simply be:

$$\text{newmexA} = \begin{cases} x, & \text{if } \text{cntA}[x] = 1 \text{ and } x < \text{mexA}, \\ \text{nextA}, & \text{if } y = \text{mexA} \text{ and } (\text{cntA}[x] > 1 \text{ or } x \leq \text{mexA} \text{ or } x \geq \text{nextA}), \\ x, & \text{if } y = \text{mexA} \text{ and } \text{cntA}[x] = 1 \text{ and } \text{mexA} < x < \text{nextA}, \\ \text{mexA}, & \text{otherwise.} \end{cases}$$

Here, $\text{cntA}[x]$ indicates the frequency of a number in array A . We compute the vice versa for B to find **newmexB**. Because evaluating the swap now takes $O(1)$ time using the conditions above, trying all i from 0 to $k - 1$ for a prefix of length k takes $O(k)$ time. Doing this for every prefix yields an overall time complexity of $O(n^2)$, seamlessly passing this subtask.

Subtask 5 (Full Solution)

To achieve an $O(n \log n)$ solution, we must optimize the $O(1)$ evaluations from Subtask 4. The key observation is that when we add a new element to extend the prefix, only a very few specific indices will actually change their swap-result:

- Indices whose A or B value just became non-unique (i.e., their frequency just increased from 1 to 2).
- Indices affected by a change in mexA , mexB , nextA , or nextB .

Because MEX only increases as the prefix grows, and a value can become non-unique at most once, each index is recomputed very few times overall. We re-evaluate an index i only if one of the trigger events mentioned in the **newmexA** conditions occurs.

To quickly find the minimum answer among all valid choices, we maintain the candidate differences, $|\text{newmexA} - \text{newmexB}|$, in a Segment Tree. We

initialize the leaves of the segment tree with infinity. Then, we sweep through every prefix. For each prefix, we ONLY update the segment tree at the specific candidate indices that are affected by a change in $mexA$, $mexB$, $nextA$, or $nextB$.

After updating the necessary indices for the current prefix, the root node of the segment tree will naturally store the global minimum difference for us. This allows us to query the best possible swap for each prefix in $O(1)$ time from the root, with updates taking $O(\log n)$ time, smoothly passing the full constraints.

BdOIDS Monthly Round 2 Solutions: Candies

Problem Information

Problem Author: Aritro Sarkar

Problem Preparation: Aritro Sarkar, Nabeul Islam

Editorial: Aritro Sarkar, Md. Riasat Islam, Nabeul Islam

Subtasks hinting key observations

Preliminary Observation:

Subtasks 1 and 2 should be solved separately, as Subtask 1 allows only 2 queries but Subtask 2 allows 10. It is easy to get partial scores for subtask 2, but the idea needed to get 100 points comes from the idea of Subtask 1.

Subtask 1: $v = w + 1, q \leq 2$

Let the special candy be in index i .

1st, we can query $\{1, 0, \dots, 0\}$, and we will get a_1 , which is either w or v .

Let $r := a_1$

In our 2nd query, we can query $\{1, 2, \dots, n\}$ to get:

$$s_1 = w + 2w + \dots + (i-1)w + iv + (i+1)w + \dots + nw.$$

Let $s_2 = r + 2r + \dots + nr$.

Note that $s_1 \neq s_2$.

If $s_1 < s_2$, then $v = r, w = r - 1$, and $i = 1$

If $s_1 > s_2$, then

$$s_2 - s_1 = iv - iw = i \cdot (v - w) = i$$

Therefore, $v = r + 1, w = r$, and $i = s_2 - s_1$

Subtask 2: $q \leq 10$

We define a_i as the taste of the i -th candy.

Partial Solutions:

Brute Force: ≈ 6

A brute force would recover a , which will take n queries. Then, we can iterate over a and find w, v, ind . This solution will give approximately 6 points.

Naive Binary Search: ≈ 42

We define

$$f(x) = \sum_{i=1}^x a_i$$

$f(x)$ is a monotonic function. Which means we can binary search on this function. The binary search should be to find the first index x , such that $f(x) \neq x \cdot a_i$, where $i \in \{1, 2, \dots, x\}$. This solution uses $1 + \lfloor \log_2 n \rfloor = 14$ queries. However, there is a corner case for which we need to use 1 more query, which will give us ≈ 42 points.

consider the array $\{7, 5, 5, 5, 5\}$, $v = 7, w = 5$.

for $x = 1$, $f(x) = 7$, but the available set is 1, so $a_1 \cdot x = a_1 \cdot 1 = 7$. so $f(x) = a_i \cdot x$, where $i \in \{1\}$. Which shouldn't be the case and the binary search would then binary search on the right part. For situations like this, it can be proved/shown that after the binary search. The index x we get, is either v , or index $x - 1$ is v . There is nothing in between. We need to use our last query to determine the value of x .

Full Solutions:

Algebraic Solution: 3 queries

Let the special candy be in index i .

1st, we can query $\{1, 0, \dots, 0\}$, and we will get a_1 , which is either w or v .

Let $r := a_1$

In our 2nd query, we can query $\{1, 2, \dots, n\}$ to get:

$$s_1 = w + 2w + \dots + (i-1)w + iv + (i+1)w + \dots + nw.$$

$$\text{Let } s_2 = r + 2r + \dots + nr.$$

Note that $s_1 \neq s_2$.

If $s_1 < s_2$, then $v = r$, and $i = 1$. We get w after we query $\{0, 1, 0, \dots, 0\}$

If $s_1 > s_2$, then

$$w = r$$

We can use $\{1, 1, \dots, 1\}$ as our 3rd query to get

$$s_3 = w + w + \dots + v + \dots + w = v + w \cdot (n - 1)$$

Therefore, $v = s_3 - w \cdot (n - 1)$

$$s_2 - s_1 = iv - iw = i \cdot (v - w)$$

$$i = \frac{s_2 - s_1}{v - w}$$

Divisor Trick + Binary Search: 10 queries

Let the special candy be in index i .

1st, we can query $\{1, 0, \dots, 0\}$, and we will get a_1 , which is either w or v .

Let $r := a_1$

In our 2nd query, we can query $\{1, 2, \dots, n\}$ to get:

$$s_1 = w + 2w + \dots + (i - 1)w + iv + (i + 1)w + \dots + nw.$$

Let $s_2 = r + 2r + \dots + nr$.

Note that $s_1 \neq s_2$.

If $s_1 < s_2$, then $v = r$, and $i = 1$. We get w after we query $\{0, 1, 0, \dots, 0\}$

If $s_1 > s_2$, then

$$w = r$$

$$s_2 - s_1 = iv - iw = i \cdot (v - w)$$

$$n \leq 10^4, w < v \leq 10^3$$

$$\text{Thus, } i \cdot (v - w) \leq 10^7$$

$$g(x) = \max \{ n \in N \mid n \leq x \text{ and } d(n) \geq d(k) \forall k \leq x \}.$$

g is known as the **Highly Composite Number Function**.

$g(10^7) = 8648640$, which has 448 divisors.

Therefore, the number of valid indices i will be at most 448, since $i \mid (s_2 - s_1)$.

Now, we can use the naive binary search shown in the partial solutions section to get i and v in $\lfloor \log_2 \tau(g(10^7)) \rfloor = 8$ queries.

In total, this solution uses $2 + 8 = 10$ queries.