



Problem Tutorial: “Forgetful Shustrik and the Remote Control”

Brief statement

There is a number A and a fixed number M . In one step, one of the following operations can be performed:

$$A \leftarrow A + M, \quad A \leftarrow A - M, \quad A \leftarrow A \cdot M, \quad A \leftarrow A \bmod M.$$

It is necessary to find the shortest sequence of operations after which the number A' is obtained, satisfying

$$A' \equiv B \pmod{N}.$$

If such a sequence does not exist, output -1 .

Scoring

Points for each subtask are awarded only if all tests for that subtask and necessary subtasks are successfully passed.

Group	Points	Additional constraints	Required groups
1	7	$ A , M, N \leq 8$	–
2	6	$M = 1$	–
3	8	$M = N$	–
4	12	$M \equiv 0 \pmod{N}$	3
5	15	$N \equiv 0 \pmod{M}$	2–3
6	16	$N \cdot M \leq 2 \cdot 10^6$	1–2
7	18	$A = 0$	–
8	18	–	1–7

Main observation

We are interested not in the number A' itself, but only in its remainder modulo N .

If we use only the operations

$$+, -, *,$$

then all transitions modulo N are determined uniquely:

$$x \rightarrow (x + M) \bmod N, \quad x \rightarrow (x - M) \bmod N, \quad x \rightarrow (x \cdot M) \bmod N.$$

So, if we ignore the $\%$ operation, the problem turns into finding the shortest path in a graph with N vertices, where a vertex is a remainder modulo N .

Why the $\%$ operation is highly restrictive

Let us consider the value of the number modulo M .



- The $+$ operation does not change the remainder modulo M .
- The $-$ operation does not change the remainder modulo M .
- The $\%$ operation maps the number into the range $[0, M - 1]$, but its remainder modulo M stays the same.
- The $*$ operation makes the remainder modulo M equal to zero, since

$$A \cdot M \equiv 0 \pmod{M}.$$

From this it follows:

Before the first $*$ operation, the remainder of the number modulo M always equals the initial $A \bmod M$. After the first $*$ operation, the remainder modulo M becomes zero and never changes again.

Now let us look at the $\%$ operation. It does not change the remainder modulo M , but simply replaces the number with its smallest nonnegative representative.

Therefore, the $\%$ operation can be useful only to immediately move into one of two special states:

- into the number $A \bmod M$, if there was no $*$ operation before it;
- into the number 0, if there was already a $*$ operation before it.

Key lemma about the form of an optimal answer

Lemma. In an optimal sequence, the $\%$ operation can appear only in one of three cases:

1. it does not appear at all;
2. it is the first operation;
3. it is preceded by exactly one $*$ operation, and this pair has the form $*\%$.

Proof.

The $+$ and $-$ operations do not change the remainder modulo M , and the $\%$ operation only replaces the number with a representative of the same remainder in the range $[0, M - 1]$.

Therefore, if only $+$ and $-$ operations were performed before $\%$, then this whole prefix does not affect the result of applying $\%$: after it, the number will still be

$$A \bmod M.$$

Hence, such a prefix can be removed, and the path becomes shorter. So, if the $\%$ operation is used before the first $*$, it must be the first operation.

Now consider the case when a $*$ operation has already been performed before $\%$. After the first $*$, the number becomes divisible by M , so its remainder modulo M is zero. Then after $\%$, the number will necessarily be 0.

If there were any other operations between this first $*$ and $\%$, they do not change the remainder modulo M , and therefore after $\%$ the result will still be 0. Hence, all such intermediate operations can be removed. So, if $\%$ is used after the remainder modulo M has become zero, then in an optimal answer it goes immediately after the first $*$, that is, as $*\%$.



Main solution

Now the problem is completely reduced to three possible starts:

1. start from $A \bmod N$ — this is the case when we do not use % at all;
2. start from $(A \bmod M) \bmod N$ — this is the case when the first operation is %;
3. start from 0 — this is the case when the first two operations are *%.

After that, only the operations $+$, $-$, $*$ remain, and they can already be considered entirely modulo N . Therefore, for each of the three starts, we need to find the shortest path to the remainder B in the graph of remainders modulo N .

State graph

Graph vertices:

$$0, 1, 2, \dots, N - 1.$$

From vertex v , there are up to three transitions:

1. $v \rightarrow (v + M) \bmod N$,
2. $v \rightarrow (v - M) \bmod N$,
3. $v \rightarrow (v \cdot M) \bmod N$.

All edges have weight 1, so the shortest path is found by a standard BFS.

Problem Tutorial: “Grand Renaming”

Scoring

Group	Points	Additional constraints	Required groups
1	14	$n = 1$	—
2	17	$\text{sorted}(s_i) = \text{sorted}(t_i)$	—
3	25	s_i and t_i have no common letters	—
4	13	$n \leq 1000; s_i , t_i \leq 10$	—
5	11	$ s_i = 1, t_i = 1$	—
6	11	All strings consist of the letter a	—
7	9	—	1–6

Here, $\text{sorted}(s_i)$ denotes the string s_i sorted in alphabetical order. For example, $\text{sorted}(\text{abacaba}) = \text{aaaabbc}$.



Existence of an answer

First of all, let us state a necessary condition for an answer to exist: the multiset of letters over all strings s_i must coincide with the multiset of letters over all strings t_i . If this is not the case — the answer is -1 .

The full solution is obtained by combining the solutions for subtasks 2 and 3, so let us examine them.

Subtask 2 (for each i : $\text{sorted}(s_i) = \text{sorted}(t_i)$):

Note that in an optimal solution, there is no need to move letters between different strings. Indeed, if a letter moved from string i to string j , then later an equal letter must return from some k to i (since the multisets are equal). But then, instead of these two operations, we could immediately move a letter from k to j without increasing the total cost. Therefore, the problem splits by indices: for each i , independently, we need to transform s_i into t_i using only operations with $i = j$, each costing 1.

We need to maximize the number of letters that do not participate in any operations. Note that such letters necessarily form a common subsequence of the strings s_i and t_i . Therefore, the letters that do not need to be touched are exactly the longest common subsequence (LCS) of s_i and t_i . The minimum number of operations is $|s_i| - |\text{LCS}|$. The LCS can be found with standard dynamic programming in $O(|s_i| \cdot |t_i|)$.

Subtask 3 (for each i : s_i and t_i have no common letters):

Each letter must change its string at least once, and moving any letter more than once is clearly suboptimal. Therefore, the relative position of a letter inside each string does not matter, and the problem splits into independent subproblems by alphabet character: for a fixed letter c , write down the indices of strings in which c occurs in s , and the indices of strings in which c is needed in t . Their counts must be equal (otherwise the answer is -1). We need to pair them up as (i, j) so as to minimize $\sum (1 + |i - j|)$.

The optimal pairing is obtained by sorting both lists and matching them in sorted order. This can be proved by contradiction: suppose the optimal answer is different. Then there exist two pairs (i_1, j_1) and (i_2, j_2) such that $i_1 < i_2$ and $j_1 > j_2$. In this case, we can swap j_1 and j_2 , and the total cost $(2 + |i_1 - j_1| + |i_2 - j_2|)$ will not increase.

Full solution:

For each string i , find $\text{LCS}(s_i, t_i)$. The letters included in the LCS stay in place. The remaining letters are processed independently by alphabet character: sort the lists of positions and pair them in sorted order, summing $1 + |i - j|$. The resulting asymptotic complexity is $O(n \cdot |s_i| \cdot |t_i| + 26 \cdot n)$.

Problem Tutorial: “Olympiad Schedule”

Scoring

The tests for this problem consist of thirteen groups. Points for each group are awarded only if all tests of the group and all tests necessary for it are passed.



Group	Points	Additional constraints	Necessary groups
1	10	$n, D \leq 20; b_i \leq 20$ for all $i; t = 1$	–
2	7	$n, D \leq 20; b_i \leq 20$ for all i	1
3	11	$n, D \leq 100; b_i \leq 100$ for all i	1, 2
4	9	$n, D \leq 800; b_i \leq 800$ for all i	1, 2, 3
5	5	$n, D \leq 2500; b_i \leq 2500$ for all $i; t = 1$	1
6	4	$n, D \leq 2500; b_i \leq 2500$ for all i	1–5
7	5	$b_i = b_j$ for all i, j	–
8	5	$d_i \neq d_j$ for all i and j , such that $i \neq j$	–
9	5	$d_i = 1$ for all $i; t = 1$	–
10	5	$d_i = d_j$ for all i, j	9
11	8	$D \leq 2 \cdot 10^6$	1–6
12	10	$t = 1$	1, 5, 9
13	16	–	1–12

General idea

Notice that:

1. On each day, it makes sense to keep **only one** olympiad, the one that gives the maximum benefit among all olympiads scheduled on that day. The rest of the olympiads on that day can be “hidden” (moved to another day or kept, but they will not affect the benefit).
2. The problem reduces to choosing **at most** D olympiads and distributing them across different days (one per day), preserving the order and the date constraints.
3. The unchosen olympiads can be placed on any days (without violating the order), but they will not contribute to the answer.

Group 1 (10 points)

$$n, D \leq 20, b_i \leq 20, t = 1$$

Solution: Enumerate all 2^n subsets of olympiads that can contribute benefit. For a fixed subset of size k , we need to check whether these olympiads can be placed on k different days.

Check: We place the chosen olympiads greedily from left to right: each next olympiad is placed on the leftmost day that is valid for it.

Complexity: $O(2^n \cdot n)$.

Group 2 (7 points)

$$n, D \leq 20, b_i \leq 20$$



Solution: We extend the approach from Group 1: after choosing a subset of “useful” olympiads, we need to place **all** olympiads. We iterate over the olympiads and place each next one on the leftmost day that is valid for it. **Complexity:** $O(2^n \cdot n)$.

Groups 3-4 (20 points)

$n, D \leq 800, b_i \leq 800$

Solution: Dynamic programming: $dp[i][d]$ — the maximum benefit for the first i olympiads, if we have considered the first d days (i.e. all olympiads are placed on days $\leq d$, and days $> d$ have not been used yet).

Transition: Consider the i -th olympiad. Iterate over day x from $\max(d_i, d)$ to D , where we will place it:

$$dp[i+1][x] = \max(dp[i+1][x], dp[i][d] + b_i)$$

If we do not count the olympiad in the answer, then we simply move to state $i+1, d$.

Complexity: $O(n \cdot D^2)$.

Groups 5-6 (9 points)

$n, D \leq 2500, b_i \leq 2500$

Solution (for groups 5–6): A more efficient DP with two types of transitions:

State $dp[i][d]$ — the maximum benefit for the first i olympiads, if we have considered the first d days, **and day d is already closed** (i.e. we can no longer place counted olympiads on day d).

Transitions:

1. Olympiad i is placed on day d (and is not counted): $dp[i+1][d] = \max(dp[i+1][d], dp[i][d])$
2. Place the i -th olympiad on day $d+1$: $dp[i+1][d+1] = \max(dp[i+1][d+1], dp[i][d] + b_i)$ (provided $d+1 \geq d_i$)
3. Skip day d (the i -th olympiad is placed later): $dp[i][d+1] = \max(dp[i][d+1], dp[i][d])$

Complexity: $O(n \cdot D)$.

Group 7 (5 points)

All b_i are equal

Solution: Since all benefits are equal, the problem reduces to maximizing the **number of days** that contain at least one olympiad. Greedy: go through the days from 1 to D . For each day, try to find the leftmost olympiad that has not yet been placed and can be put on this day (i.e. $d_i \leq \text{current day}$). If we find one, place it; otherwise the day remains empty. This maximizes the number of occupied days.

It is enough to consider the last n days.

Complexity: $O(n + D)$.

Group 8 (5 points)

All d_i are distinct

Solution: Leave each olympiad on its original date. Then all days are occupied by different olympiads, and the total benefit is $\sum b_i$.

Complexity: $O(n)$.



Group 9 (5 points)

All $d_i = 1$, $t = 1$

Solution: All olympiads are initially on day 1. We can move them to any days ≥ 1 . We need to choose $\min(n, D)$ olympiads (one per day) with maximum total benefit. Take the $\min(n, D)$ olympiads with the largest benefit (if benefits are equal, any ones). Arrange them in order of indices on days $1, 2, \dots$

Complexity: $O(n \log n)$.

Group 10 (5 points)

All d_i are equal (common date d_0)

Solution: Available days: from d_0 to D , in total $D - d_0 + 1$ days. On each such day, at most one olympiad can be placed. Choose $\min(n, D - d_0 + 1)$ olympiads with the largest benefit and arrange them in order of indices.

Complexity: $O(n \log n)$.

Groups 11-13 (16 points)

No additional restrictions

Full solution (greedy from the end):

1. We go through the days from D down to 1.
2. We maintain an **active set** of olympiads that we **plan to count** in the answer, and a data structure for quickly extracting the minimum benefit (for example, a min-heap).
3. For the current day j , we add to the active set all olympiads with $d_i \leq j$ (they can be moved to day j or later).
4. If the active set contains more elements than there are days left (including the current one), remove the olympiad with the **smallest** benefit from it (it will not be included in the answer).
5. After processing all days, the active set will contain the olympiads that give the maximum benefit.

Why it works: At each step, we ensure that for the remaining days (from the end) there is enough room to place all chosen olympiads in order of indices. Removing the least beneficial olympiad is optimal, because if we need to free space, it is better to sacrifice the olympiad with the smallest benefit.

Complexity: $O((n + D) \log n)$.

Note: In this solution, we first kept only the restriction on relative order and ignored the second restriction (the nondecreasing date constraint), and then restored it (started ignoring the *extra* olympiads with minimum benefit). The reverse is also possible — keep only the nondecreasing date constraint. Then the greedy solution (going from left to right and placing the olympiad with maximum benefit) gives us only the set of olympiads, but does not preserve the relative order. We only need to sort the chosen olympiads by index and place them in the last k days.

Problem Tutorial: “Tower of Boxes”

Scoring

In this problem, there are two types of tests.



The first type of test — an array $[1, 1, 2, 2, \dots, N, N]$ is generated and shuffled randomly, resulting in an array $[a_1, a_2, \dots, a_T]$ of size T , where $T = 2N$. A box is added at time t_1 and taken out at time t_2 , if $t_1 < t_2$ and $a[t_1] = a[t_2]$.

The second type of test — it is guaranteed that the boxes are taken out in the same order they are added.

For each value of N (the number of boxes), one test of each type is generated.

Group	Tests	Values of N	Points	Cumulative Sum	Dependencies
1	1-2	Tests from the statement	0 per test	0	-
2	3-10	100, 200, 330, 420	1 per test	8	-
3	11-18	480, 720, 820, 1080	1 per test	16	1
4	19-26	1240, 1420, 1630, 1870	1 per test	24	1-2
5	27-30	2150, 2470	1 per test	28	1-3
5	31-34	2840, 3260	2 per test	36	1-3
6	35-42	3740, 4300, 4940, 5680	2 per test	52	1-4
7	43-50	6530, 7500, 8620, 9910	2 per test	68	1-5
8	51-58	11390, 13090, 15050, 17300	2 per test	84	1-6
9	59-66	17500, 18000, 18500, 19000	2 per test	100	1-7

One possible solution

One of the simplest solutions is to always place a new item on top of the stack, and when extracting an item, collect all items blocking its extraction into a buffer, extract the item, and then return the items from the buffer to the stack in sorted order — the earlier an item is extracted, the higher in the stack we place it.

Such a solution is not always optimal. For example, from the stack state $[\dots, 2, 1, 3, 4]$ (here the rightmost element is on top of the stack), we move to the state $[\dots, 2, 4, 3]$ when extracting the item with number 1. Then, when extracting the item with number 2, items 3, 4 will be placed into the buffer again. It is clear that if, initially, together with item 1, we also extract one more item — the next item, 2 — the total number of operations will decrease. Namely, from the stack state $[\dots, 2, 1, 3, 4]$ we move to the state $[\dots, 4, 3, 2]$, after which we extract each of the items in one action from the top of the stack. In this case, items 2, 3, 4 will each end up in the buffer once, and the number of actions will be reduced.

Method 1 (12 actions)

Step	Stack	Buffer	Actions
1	$[\dots, 2, 1, 3, 4]$	$\emptyset$	Put 4, 3 into the buffer
2	$[\dots, 2, 1]$	$\{3, 4\}$	Extract 1
3	$[\dots, 2]$	$\{3, 4\}$	Return 4, 3 from the buffer
4	$[\dots, 2, 4, 3]$	$\emptyset$	Put 3, 4 into the buffer
5	$[\dots, 2]$	$\{3, 4\}$	Extract 2
6	$[\dots]$	$\{3, 4\}$	Return 4, 3 from the buffer
7	$[\dots, 4, 3]$	$\emptyset$	Extract 3, 4
8	$[\dots]$	$\emptyset$	



Method 2 (10 actions)

Step	Stack	Buffer	Actions
1	[..., 2, 1, 3, 4]	$\emptyset$	Put 4, 3 into the buffer
2	[..., 2, 1]	{3, 4}	Extract 1
3	[..., 2]	{3, 4}	Put 2 into the buffer
4	[...]	{2, 3, 4}	Return 4, 3, 2 from the buffer
5	[..., 4, 3, 2]	$\emptyset$	Extract 2, 3, 4 one by one
6	[...]	$\emptyset$	

However, let us consider both methods in the case when, after extracting item 1, items 5, 6 arrive to be placed.

Case 2. Method 1 (20 actions)

Step	Stack	Buffer	Actions
1	[..., 2, 1, 3, 4]	$\emptyset$	Put 4, 3 into the buffer
2	[..., 2, 1]	{3, 4}	Extract 1
3	[..., 2]	{3, 4}	Return 4, 3 from the buffer
4	[..., 2, 4, 3]	$\emptyset$	Place 5, 6
5	[..., 2, 4, 3, 5, 6]	$\emptyset$	Put 6, 5, 3, 4 into the buffer
6	[..., 2]	{3, 4, 5, 6}	Extract 2
7	[...]	{3, 4, 5, 6}	Return 6, 5, 4, 3 from the buffer
8	[..., 6, 5, 4, 3]	$\emptyset$	Extract 3, 4, 5, 6 one by one
9	[...]	$\emptyset$	

Case 2. Method 2 (22 actions)

Step	Stack	Buffer	Actions
1	[..., 2, 1, 3, 4]	$\emptyset$	Put 4, 3 into the buffer
2	[..., 2, 1]	{3, 4}	Extract 1
3	[..., 2]	{3, 4}	Put 2 into the buffer
4	[...]	{2, 3, 4}	Return 4, 3, 2 from the buffer
5	[..., 4, 3, 2]	$\emptyset$	Place 5, 6
6	[..., 4, 3, 2, 5, 6]	$\emptyset$	Put 6, 5 into the buffer
7	[..., 4, 3, 2]	{5, 6}	Extract 2
8	[..., 4, 3]	{5, 6}	Put 4, 3 into the buffer
9	[...]	{3, 4, 5, 6}	Return 6, 5, 4, 3 from the buffer
10	[..., 6, 5, 4, 3]	$\emptyset$	Extract 3, 4, 5, 6 one by one

So, our current actions may seem optimal, but taking into account items that will arrive in the future, a different set of actions may turn out to be optimal.

Proposed solutions

Here and below, T denotes the maximum number of elements that can be in the stack simultaneously.



Solution in $O(N\sqrt{T})$ actions

Split the events into blocks of $\sqrt{T}$ events. Before the start of a block, in at most T actions, we can place the smallest $\sqrt{T}$ elements on top of the stack. After that, after the i -th event in the block, we will maintain a state such that at least $\sqrt{T} - i$ smallest elements are among the top $\sqrt{T}$ elements. Maintaining such a state is not difficult — indeed, after the next extraction, the number of smallest elements among the top $\sqrt{T}$ decreases by at most one, and when placing a new element, one smallest element may be pushed deeper into the stack. At the same time, to obtain the next element, it is enough each time to spend $\sqrt{T}$ actions — after all, at least $\sqrt{T} - i > 0$ smallest elements are near the top boundary of the stack at the i -th step in the block. Then once every $\sqrt{T}$ events, such a solution will spend $O(T)$ actions, and for each extraction no more than $\sqrt{T}$ actions. The total number of actions for this solution does not exceed

$$O(N\sqrt{T}) = N\sqrt{T} + \frac{N}{\sqrt{T}}T.$$

Estimate of the number of actions

Consider an example — let N items arrive and leave in queue order, while the queue size is T . Namely, at time $2i$, an element with value $t_i = i + 2T + 1$ appears. Then from time $2T$ onward, items are placed into and extracted from the stack one by one.

Let a_i denote the number of elements removed from the stack at time i . Note that the number of moments when $a_i \geq q$ for any integer $q < T$ is at least $(N - T)/q$. Let us prove this.

Suppose that for q consecutive steps, the value of a_i does not exceed q (we count one step as extracting an item and then placing a new item). Then the element x that must be extracted immediately after these q steps will be at depth at least q in the stack, because in the top q elements of the stack, with each new placement, one element greater than x appears (more precisely, at each step one element from the top q is replaced by a new one, and no element could first replace and then be replaced; therefore all elements were replaced and no elements $\leq x$ remained). Hence, on the next step, to gain access to element x and extract it, at least $q + 1$ actions are needed.

Consider values $q = 2^m$ for all $2^m < T$. Then for each such value q , there will be at least $N/(q + 1)$ moments such that $a_i \geq q$. At the same time, one moment will be counted several times (for different values of q). Then for each value m we assign $(N/2^m - N/2^{m+1})$ moments, starting from $m = \log_2 T$ and ending with $m = 1$, each time excluding moments already used before. We obtain that the total number of actions was on the order of

$$N \log T \approx \sum_{m=1}^{\log T} \left(\frac{N}{2^{m+1}} \right) \cdot 2^m.$$

At the same time, the given asymptotic estimate matches the upper bound on the number of actions — below we present a solution that, for arbitrary extraction times, performs no more than $O(N \log T)$ actions, where T is the maximum number of elements stored in the stack simultaneously (if this parameter is known in advance).

Solution in $O(N \log T)$ actions

We will count the number of events (the arrival of a new element to be placed and the extraction of the next element) that have occurred since the algorithm started. Then after event j , for the largest power of two m such that $2^m \mid j$ (that is, j is divisible by 2^m), we will take $\min(2 \cdot 2^m, |S|)$ elements from the stack and return them in decreasing order of extraction time.

Lemma: Before the action at time j , the next $\min(2^m, |S|)$ items by extraction time (not counting those not yet arrived in the stack) are among the top $2 \cdot 2^m$ elements of the stack.



Proof: We prove by induction on j . Induction base: for $j = 0$, there are zero elements in the stack, and the lemma holds for them.

Inductive step. Note that after processing event $j - 2^m$, the statement holds (since $j - 2^m < j$) by the induction hypothesis. $2^{m+1} \mid j - 2^m$, and therefore the $\min(2 \cdot 2^{m+1}, |S|)$ smallest elements by extraction time are ordered and are at the top of the stack after processing event $j - 2^m$.

Let among the events $[j - 2^m, j)$ there be r extractions and $2^m - r$ placements. Each extraction takes the smallest element from the top $2 \cdot 2^m$, and each placement pushes the latest of the top elements deeper into the stack. Then before processing event j , at least

$$2 \cdot 2^m - 2^m = 2^m$$

smallest elements by extraction time remain among the top $2 \cdot 2^m$ elements, which is exactly the required number $\min(2^m, |S|)$.

Exercise for the reader:

Come up with a solution in $O(N^{4/3})$.

Problem Tutorial: “Cheng Shi and the Collection of Bamboo Inscriptions”

Scoring

Subgroup	Points	Additional constraints			Required subgroups
		q	$\sum s_i $	Type of operations	
1	10	$q \leq 100$	$\sum s_i \leq 1000$	+ and -	–
2	10	$q \leq 1000$	$\sum s_i \leq 10^4$	+ and -	1
3	15	$q \leq 50\,000$	$\sum s_i \leq 10^5$	+ and -	1,2
4	10	$q \leq 5 \cdot 10^5$	$\sum s_i \leq 5 \cdot 10^5, s_i \leq 10$	+	–
5	10	$q \leq 5 \cdot 10^5$	$\sum s_i \leq 5 \cdot 10^5, s_i \leq 10$	+ and -	4
6	20	$q \leq 5 \cdot 10^5$	$\sum s_i \leq 5 \cdot 10^5$	+	4
7	25	–	–	+ and -	1–6

Brief statement

A dynamic multiset of strings is maintained. After each insertion or deletion of a string, we need to find the minimum number of operations required to make all strings equal.

Allowed operations on a single string:

- delete the last character;
- append any character to the end.

The string to which we transform all others may be any string, including the empty one.



Observation 1: reduction to a prefix tree

Build a trie over all strings that appear in the input queries. The root corresponds to the empty string, each vertex — to some prefix, and the depth of a vertex equals the length of the corresponding string.

Each string from the multiset is now identified with a vertex of the trie.

If we want to transform a string s into a string t , then:

- we need to delete from s the suffix after their longest common prefix;
- then append the missing suffix of string t .

Therefore, the number of operations is

$$|s| + |t| - 2 \cdot |LCP(s, t)|.$$

But this is exactly the distance between two vertices in the trie. Hence, the cost of transforming string s into string t is equal to the distance between the corresponding vertices in the tree.

So the problem reduces to the following:

There is a tree (a trie), and some vertices contain active tokens with multiplicities. After each insertion/deletion of a token, we need to find a vertex v minimizing the sum of distances to all active tokens.

Why it is enough to consider only trie vertices

The statement allows transforming all strings into any string, even if it is not in the multiset. However, it is enough to consider only trie vertices.

Indeed, if the target string is not a prefix of any string appearing in the input, then the corresponding vertex does not exist in the trie. Then the last character of this string leads to a subtree containing no active strings. If we move the target one character up, the distance to every active string does not increase, and in fact decreases by 1. Therefore, the optimal target is always a vertex of the trie built over all query strings.

Observation 2: the optimal vertex is a tree median

Let a vertex v be chosen in the tree, and let the total cost be

$$F(v) = \sum_{u \text{ active}} \text{dist}(u, v),$$

where strings are counted with multiplicities.

Consider moving from vertex v to its neighbor to . Let x active strings lie in the component containing to after removing edge (v, to) , and let the total number of active strings be M .

Then:

- for these x strings, the distance decreases by 1;
- for the remaining $M - x$ strings, the distance increases by 1.



Therefore,

$$F(to) - F(v) = (M - x) - x = M - 2x.$$

This immediately gives the optimality criterion.

Lemma. A vertex v minimizes the sum of distances to all active strings if and only if, after removing v , every component contains at most half of all active strings.

Proof. If some component contains more than half of the active strings, then moving in that direction decreases the answer, since $M - 2x < 0$. If every component contains at most half, then moving in any direction does not improve the answer.

This is the usual weighted tree median.

How to find the median in a rooted tree

Fix the trie root at the empty string.

Suppose there are M active strings in total. Then the median is a vertex such that:

- no child subtree contains more than $M/2$ active strings;
- and we cannot move upward anymore, otherwise we would enter the “everything else” component containing more than half.

Equivalently: if we go from the root downward, as long as there exists a child whose subtree contains strictly more than half of all active strings, we should go into that child. We stop exactly at the median.

So the median is the deepest vertex whose subtree contains more than half of all active strings.

How to find such a vertex quickly

We will store active strings in trie vertices with multiplicities.

Perform an Euler tour of the trie. Then each subtree corresponds to an interval:

$$subtree(v) \leftrightarrow [tin[v], tout[v]).$$

If we store in a Fenwick tree or segment tree the number of active strings at positions $tin[v]$, then the number of active strings in the subtree of vertex v can be obtained as the sum on the Euler-tour interval.

Now we want to find the median. For this, it is convenient to do two steps.

Step 1. Find, in Euler-tour order, the first active string after which the prefix sum becomes greater than

$$\left\lfloor \frac{M}{2} \right\rfloor.$$

Denote the corresponding vertex by x .

Step 2. The median lies on the path from the root to x . Indeed, the subtree of the median contains more than half of all active strings, so the moment when we “cross half” in the Euler tour must occur inside that subtree.

Now among the ancestors of vertex x , we need to find the deepest vertex v such that

$$cnt(subtree(v)) > \frac{M}{2}.$$



This vertex is the median.

This can be done by binary lifting over ancestors: we can quickly check the size of the subtree of any ancestor, so we find the needed ancestor in $O(\log^2 N)$ or faster.

How to quickly compute the sum of distances to all active strings

After the median is found, we need to compute

$$\sum_{u \text{ active}} \text{dist}(u, \text{med}).$$

For this, we use the standard dynamic on the centroid tree.

What is stored in the centroid tree

Build the centroid tree on the trie.

For each centroid vertex c , we store:

- $\text{cnt}[c]$ — how many active strings are accounted for in the structures of this centroid;
- $\text{sum}[c]$ — the sum of distances from all active strings to c .

In addition, for each centroid x (except the root of the centroid tree), we store information about the part corresponding to its component with respect to its parent in the centroid tree:

- $\text{subcnt}[x]$ — how many active strings lie in this part;
- $\text{subsum}[x]$ — the sum of distances from these strings to the parent of x in the centroid tree.

Update

When adding or removing an active string at vertex v , we move up through all centroids on the path from v to the root of the centroid tree.

For each such centroid c :

- update $\text{cnt}[c]$;
- update $\text{sum}[c]$ by $\pm \text{dist}(v, c)$.

And for the centroid child through which we came upward, we also update subcnt and subsum .

The number of such centroids is $O(\log N)$.

Query

For an arbitrary vertex v , the sum of distances to all active strings is computed in the standard way.

We move along the path from v to the root of the centroid decomposition. For each centroid c :

- add
$$\text{sum}[c] + \text{cnt}[c] \cdot \text{dist}(v, c),$$

which corresponds to all active strings accounted for in c ;



- but the part of the strings lying in the same centroid subtree as v has been counted once too many, so it must be subtracted using *subcnt* and *subsum*.

As a result, the sum of distances from v to all active strings is computed correctly in $O(\log N)$.

Problem Tutorial: “Nimesweeper!”

Scoring

Group	Points	Additional Constraints	Required Groups
1	8	$n \leq 8$	–
2	11	$n \leq 18$	1
3	20	$n \leq 45$	1–2
4	19	$n \leq 128$	1–3
5	18	$n \leq 243$	1–4
6	20	Degrees of all vertices ≤ 3	–
7	4	–	1–6

Full solution

Let us immediately present the full solution. The problem reduces to the following formulation: given a tree, all of whose vertices are initially white. We consider a process in which some vertices are sequentially recolored black. After each step, each white vertex stores the number of its black neighbors. We need to determine the maximum number of vertices that can be colored in some order so that after each of these steps the XOR of all values in the white vertices is equal to zero. If this maximum number is k , then the answer to the problem is $k + 1$.

Let us describe how the XOR changes after coloring one vertex. Suppose that at some moment a white vertex v is colored. Let t denote the number of its black neighbors at that moment. Before coloring, the value t was included in the total XOR, since vertex v was white. After coloring, vertex v is no longer counted. For each white neighbor u of vertex v , the value at vertex u increases by one, that is, it changes from a_u to $a_u + 1$. Therefore, the total XOR changes by

$$t \oplus \bigoplus_{u \text{ — white neighbor of } v} (a_u \oplus (a_u + 1)).$$

Therefore, if before the considered step the XOR was equal to zero, then after it remains equal to zero if and only if the amount of XOR change is zero.

Introduce the notation

$$\Delta(x) = x \oplus (x + 1).$$

Then the condition for coloring vertex v correctly takes the form

$$t = \bigoplus_{u \text{ — white neighbor of } v} \Delta(a_u).$$



Let us examine the values of the function $\Delta(x)$. For any x , the number $\Delta(x)$ has the form $2^k - 1$. This follows from the binary representation: when moving from x to $x + 1$, a finite suffix of bits changes, so the XOR produces a sequence of consecutive ones. For example,

$$0 \oplus 1 = 1, \quad 1 \oplus 2 = 3, \quad 2 \oplus 3 = 1, \quad 3 \oplus 4 = 7.$$

Therefore, if for a fixed white vertex the number of black neighbors increases successively from 0 to 1, then from 1 to 2, then from 2 to 3, and so on, the resulting labels form the sequence

$$1, 3, 1, 7, 1, 3, 1, 15, 1, 3, 1, 7, 1, 3, 1, 31, \dots$$

Thus, each time the next neighbor of a fixed vertex is colored, one of the indicated labels appears in the condition for preserving zero XOR.

For a fixed vertex, this sequence is determined solely by the number of already colored neighbors. If a vertex already has k colored neighbors, then the set of used labels coincides with the first k terms of the indicated sequence. Therefore, only the length of the corresponding prefix is essential.

Call a vertex state prefix-like if it corresponds to the first k elements of the sequence

$$1, 3, 1, 7, 1, 3, 1, 15, \dots$$

for some k . In the implementation, such a state is encoded by the number of already used labels of each type. Under the constraint $n \leq 369$, it is enough to consider only the labels

$$1, 3, 7, 15, 31.$$

Let us prove this. Let $c[k]$ denote the minimum number of vertices in a tree needed for the existence of a valid configuration in which some white vertex has exactly k black neighbors. To go from $k - 1$ to k , one more neighbor of that vertex must be colored. In this case the label

$$(k - 1) \oplus k = 2^x - 1$$

appears. To make the total XOR zero again after such a step, there must exist a configuration inside the subtree of the new neighbor that compensates exactly for this label. The minimum number of vertices required for this is at least $c[2^x - 1]$. Therefore,

$$c[k] \geq c[k - 1] + c[2^x - 1].$$

From the base values $c[1] = 1$, $c[2] = 3$, $c[3] = 7$ and this recurrence estimate, it follows that already $c[63] > 369$. Hence, the label 63 and all larger labels cannot appear in any valid configuration under the given constraints. Therefore, it is enough to consider only the labels 1, 3, 7, 15, 31.

The problem is solved by tree DP. The root is chosen arbitrarily. For each vertex v , we consider its subtree. Relative to the parent, there are two cases: vertex v is colored before the parent or after it. The case when vertex v is not colored in the considered good prefix is included in the second case.

The state *before_parent*[l] means that vertex v is colored before its parent, and at the moment of its coloring the increase in the number of black neighbors of the parent creates label l . In other words, label l is passed upward from the subtree of vertex v . The value of this state is the maximum number of colored vertices in the subtree of v achievable under the stated condition.

The state *after_parent*[l] means that the parent is colored before vertex v . In this case, either vertex v will be colored later and then will pass label l upward at the moment of its coloring, or vertex v will not



be colored in the considered good prefix. The special case $l = 0$ corresponds to no contribution upward. The value of this state is also the maximum number of colored vertices in the subtree of v .

First consider the case when vertex v remains white. Then for each of its children there are two valid options. In the first option, the child is also not colored by that moment; then the corresponding child subtree must be in state *after_parent*[0]. In the second option, the child is colored before vertex v ; then it uses some label l from the prefix sequence of vertex v , and its subtree must be in state *before_parent*[l]. After processing the children one by one, some prefix state of vertex v is formed, that is, it is determined which first labels of the sequence $1, 3, 1, 7, \dots$ have already been realized by its previously colored children. For each such state, we store the maximum number of colored vertices in the already processed part of the subtree.

Now consider the case when vertex v itself is colored. Then its children are split into two groups. The first group consists of children colored before vertex v ; they form some prefix state, denoted by s . The second group consists of children that are still white at the moment vertex v is colored; they will either be colored later or never colored at all.

Suppose the prefix state s corresponds to the set of children of vertex v that are already colored by the moment v is colored. Then the number of such children is uniquely determined by state s . Denote it by $total(s)$. Since at that moment the parent and the other children of vertex v are still uncolored, the number of black neighbors of vertex v at the moment of its coloring is exactly $total(s)$.

Now consider the remaining children of vertex v , that is, those that are still white at the moment of its coloring. Each such child contributes to the correctness condition for coloring v through the corresponding label. Let the XOR of all these contributions be denoted by x .

Then the correctness condition for coloring vertex v is formulated as follows. If the parent of vertex v is absent at that moment or does not contribute additionally, then the total contribution from the still-white children must match the number of already colored children, that is, the equality

$$x = total(s)$$

must hold.

If vertex v is colored before its parent and label l is passed upward, then this label also participates in the balance, and the condition becomes

$$x = l \oplus total(s).$$

Therefore, for an already colored vertex v , it is enough to store two parameters: the prefix state s , which determines the children colored before vertex v , and the value x , equal to the XOR of the contributions of the children that remain white at the moment of its coloring. When processing the next child, two cases are possible. If this child is colored before vertex v , then its subtree must be in state *before_parent*[l], and label l is added to the prefix state. If this child is still white by the moment vertex v is colored, then its subtree must be in state *after_parent*[l], and label l is XORed into the value x . After all children are processed, one of the equalities above must be checked; if it holds, then coloring vertex v is valid.

Now let us describe the transitions.

To compute *after_parent*[0], we consider all valid prefix states of the children in the case when vertex v remains white. Among them, we take the maximum number of colored vertices.

To compute *after_parent*[l] for $l \neq 0$, the following condition is used. The label l must later be passed from vertex v to its parent, so it must belong to a valid prefix of vertex v . This label is reserved for the connection with the parent, and the remaining labels of the prefix are formed by the children. After that, two cases are considered: vertex v remains white, or vertex v is colored; in the second case, the balance condition between already colored and still-white children is additionally checked.



To compute $before\_parent[l]$, vertex v must be colored before its parent. If the children colored before it form a prefix state s , then vertex v already has $total(s)$ black neighbors below, and the contribution l corresponds to the parent. Therefore, the XOR of the contributions from the children that are not colored by the moment vertex v is colored must be equal to

$$l \oplus total(s).$$

Among all states s for which this condition is satisfied, the maximum is chosen.

The correctness of the DP is proved by induction on the subtree. For a leaf, the statement is obvious, since it has no children and all states are computed directly. Assume that for all children of vertex v , the values of the states $before\_parent$ and $after\_parent$ are computed correctly. Consider a valid configuration of a good prefix inside the subtree of vertex v . For each child relative to vertex v , exactly one of three options is possible: the child is colored before vertex v , after it, or not colored at all. Therefore, the behavior of each child subtree is completely determined by one of the states $before\_parent[l]$ or $after\_parent[l]$. When combining the children at vertex v , only two characteristics are essential: the prefix state formed by the children colored before vertex v , and the XOR of the labels from the children that remain white at the moment of its coloring. No additional information about the internal structure of the child subtrees is needed to verify correctness. Therefore, the DP transitions enumerate all valid configurations without loss. Since in each state we take the maximum number of colored vertices, we obtain the correct optimal values of the states for vertex v . By induction, the statement holds for all vertices of the tree, including the root.