# Problem Tutorial: "Enchanted Cat"

*Author and problem developer: Anatoly Maksudov*

Note that the number of integers from 1 to $m$ that are divisible by $k$ is equal to $\left\lfloor \frac{m}{k} \right\rfloor$. Therefore, the number of integers from $l$ to $r$ that are divisible by $k$ is equal to $\left\lfloor \frac{r}{k} \right\rfloor - \left\lfloor \frac{l-1}{k} \right\rfloor$.

Thus, the number of suitable positive integers according to the problem statement is equal to $\left\lfloor \frac{10^n - 1}{2^x} \right\rfloor - \left\lfloor \frac{10^{n-1} - 1}{2^x} \right\rfloor$. The number 0 will be suitable when $n = 1$ and any $x$. The asymptotic complexity of such a solution is $\mathcal{O}(1)$.

For solving subtasks 1, 2, and 3, one could separately consider the calculation of the number of numbers divisible by 2, 4, and 8, respectively. In subtasks 4 and 6, one could iterate through suitable numbers with a step of $2^x$. In subtask 5, the first suitable number is $10^{n-1}$, and $10^{n-1} > 0$.

# Problem Tutorial: "SuperSavings"

*Author and problem developer: Vsevolod Lepeshov*

Note that if it is possible to perform $x$ operations, it is also possible to perform $x-1$ operations. Therefore, we can solve the problem using binary search on the answer.

We need to learn how to determine whether it is possible to perform $x$ operations.

Suppose it is possible, and the way to do this involves $c_1$ operations from partner 1, $c_2$ operations from partner 2, ..., $c_n$ operations from partner $n$. Then, for this to be possible, all constraints on the array $c$ are as follows:

- $c_1 + c_2 + \ldots + c_n = x$
- $c_i \geq 0$
- $c_i \leq a_i$
- $x - c_i \leq b_i$

That is, for each $c_i$ there is a constraint: $max(0, x - b_i) \leq c_i \leq a_i$. Therefore, if for some $i$ it turns out that $max(0, x - b_i) > a_i$, it is impossible to perform $x$ operations, as there is no suitable value for $c_i$. Otherwise, we note that $\sum c_i$ can take any value from $\sum max(0, x - b_i)$ to $\sum a_i$. Therefore, if $\sum max(0, x - b_i) \leq x \leq \sum a_i$, it is possible to choose $c_i$ such that $\sum c_i = x$, and thus it is possible to perform $x$ operations. Otherwise, achieving $\sum c_i = x$ is impossible, which means performing $x$ operations is also impossible. This check is performed in $O(n)$, and the asymptotic complexity of the entire solution is $O(n \log A)$, where $A = 2 \cdot 10^9$, the maximum possible answer to the problem.

# Problem Tutorial: "To School Through the Snow"

*Author and developer of the problem: Ivan Belkov*

**Subgroup 1.** $l = 1, dt = 0$

In this subgroup, the amount of heat does not change when passing through any transition. Thus, the problem reduces to the shortest path problem.

In this subgroup, $l = 1$, therefore BFS will find such a shortest path in $O(n + m)$.

**Subgroup 2.** $dt = 0$

As in subgroup 1, the problem reduces to the shortest path problem. However, now the transitions can be of arbitrary length, so in this subgroup, the shortest path problem should be solved using Dijkstra's algorithm. We will obtain a solution in $O(m \cdot log(n))$.

**Subgroup 3.** $dt > 0$

In this subgroup, the amount of heat can only increase. Let's keep track of $dp[t][u]$ - the length of the shortest path to intersection $u$ with a heat level of $t$. Then $dp[t][u]$ is recalculated based on transitions from values with a lower $t$.

We will iterate over possible values of $t$ from 0 to 30, recalculating the lengths of the shortest paths based on transitions from already considered states.

We will obtain a solution in $O(61 \cdot (n + m))$.

**Subgroup 4. The graph is acyclic**

In this subgroup, the graph has no cycles.

We will find a topological sorting of the graph — an order of vertices such that all edges lead from left to right. In this case, it is an order of intersections such that all transitions lead from an intersection with a smaller number to an intersection with a larger number.

Let's keep track of $dp[u][t]$ - the length of the shortest path to vertex $u$ with a heat level of $t$. Then $dp[u][t]$ is recalculated based on transitions in the order of topological sorting of intersections (in this order, all transitions will lead from intersections with already calculated $dp$ values).

We will obtain a solution in $O(61 \cdot (n + m))$.

**Subgroup 5.** $n, m \leq 10^5, -30 \leq dt \leq 30$

We solve the shortest path problem, while we want to calculate $dist[u][t]$ for each intersection and each heat level.

Notice that the set of states is not too large: there are only 61 different values of $t$.

We will construct a graph where vertex $u_t$ will represent such a state: we are at vertex $u$ and our heat level is $t$.

For transition $u\ v\ l\ dt$, we will add an edge of length $l$, leading from $u_t$ to $v_{t+dt}$ for all values of $t$.

In the resulting graph, we need to find the shortest path from vertex $1_0$ to one of the vertices $n_t$ for some $t$. We solve this problem by running Dijkstra's algorithm from vertex $1_0$.

We will obtain a solution in $O(61 \cdot (n + m) \cdot log(n))$.

# Problem Tutorial: "New Year Experiments"

*Author: Ruslan Kapralov, problem developer: Ilnur Valeev*

**Subtask 1.** $n, q \leq 10^3$, $t \leq 5$, **all types of operations**

To solve this subtask, the described process could be implemented. Queries of types 1–3 can be processed in $O(n)$, and to answer operation 4, one needs to access an array cell in $O(1)$. To answer query 5, it is necessary to create a copy of the array and sort it.

**Subtask 2.** $n \cdot q \leq 10^8$, $t \leq 2$, **all types of operations**

To solve this subtask, it was necessary to speed up the previous solution, specifically to learn how to answer query 5 in $O(n)$. For this, an algorithm for finding the $k$-th order statistic in linear time is used.

**Subtask 3.** $q \leq 4 \cdot 10^4$, $t \leq 5$, $\max a_i < 2^{10}$

There are at most $2^{10}$ distinct numbers in the array $a$. We will use this to optimize the previous solution. We will create an array $d$, where $d[x] = x$. Now, operations of types 1–3 will be performed on the array $d$ in $2^{10}$. To answer query 4, we will use the value in the array $a[i]$ as an index for the array $d$, that is, $d[a[i]]$. We can answer query 5 in $2^{10}$.

**Subtask 4. No operation of type 5**

For this subtask, it is necessary to construct a boolean function to answer queries of types 1–4 in $O(1)$.

**Subtask 5.** $n, q \leq 2 \cdot 10^5$, **no operation of type 3**

Let's consider how bitwise operations affect numbers. **AND** and **OR** affect numbers in the same way. For a specific bit $i$:

- **OR** with one or **AND** with zero "collapses"bit $i$.

Such "collapsing"occurs no more than once for each bit. If after the next operation some bit "collapses we recalculate the array from scratch. Thus, the array will be rebuilt no more than $\log(A)$ times. To answer query 5, we create a copy and sort the array only if there have been **AND** and **OR** operations that collapsed at least one bit beforehand.

**Subtask 6.** $n, q \leq 2 \cdot 10^5$, **no operations of types 1 and 2**

To solve this problem, it is convenient to use a data structure — a binary trie. We store the array of numbers $a$ in the trie, built from the most significant bits to the least significant. In this form, the numbers in the tree will be sorted. If we store the count of numbers in the subtree at each node of the trie, we can find the $k$-th largest element in the array in $\log(A)$, where $A$ is the maximum number in the array.

Now let's consider the influence of bitwise operations on the trie.

- **XOR** with zero does not change the values of the bits or the structure of the tree.
- **XOR** with one swaps the children at each node of the trie at depth $i$.

For each depth $i$, it is sufficient to maintain a flag indicating which of the child nodes the zero bit leads to.

**Solution for full points**

**AND** and **OR** affect the trie in the same way. For a specific bit $i$:

- **OR** with one or **AND** with zero "collapses"bit $i$, meaning that at depth $i$, all nodes will have exactly one child node.

Such "collapsing"occurs no more than once for each bit. If after the next operation some bit "collapses we simply rebuild the trie. Thus, the trie is rebuilt no more than $\log(A)$ times.

The final asymptotic complexity is $O(n \log^2(A) + q \log(A))$.

# Problem Tutorial: "Planet Parade"

*Author and developer of the problem: Vsevolod Lepeshov*

To begin with, let's solve the problem for $\alpha = -1$. For this, we need to notice a property of the operation mod: if $a_i < a_{i+1}$, then $a_i \bmod a_{i+1} = a_i$. Therefore, if in the permutation there is an index $a_{p_i} < a_{p_{i+1}}$, then the entire sum $a_{p_1} \bmod a_{p_2} + \ldots + a_{p_{n-1}} \bmod a_{p_n}$ will be $\geq a_{p_i}$, and hence also $\geq \min(a_1, a_2, \ldots, a_n)$. Thus, the only permutations for which it is possible to satisfy the condition for $\alpha = -1$ are those in which $a_{p_1} \geq a_{p_2} \geq \ldots \geq a_{p_n}$. It is clear that the unique array $\{a_{p_1}, a_{p_2}, \ldots, a_{p_n}\}$ with this property is the sorted array $a$ in non-decreasing order. The number of permutations that yield such an array can be calculated using the formula $c_1! \cdot c_2! \cdot \ldots \cdot c_k!$, where $c_i$ is the number of times the $i$-th smallest unique number appears in the array $a$. Next, we will count the number of suitable unique arrays-permutations of the array $a$, and then multiply the answer by this coefficient.

Thus, for the group $\alpha = -1$, it is sufficient to check the sorted array $a$ in non-decreasing order for compliance with the condition.

Next, we move on to the group $\alpha = 0$. Similarly, we will consider the sorted array in non-decreasing order if it meets the condition. Now we need to consider all arrays such that there is at least one index $a_i < a_{i+1}$, and at this index, the sum of the moduli will equal $a_i$, which is certainly not less than $\min(a_1, a_2, \ldots, a_n)$. Thus, the only option when such an array will meet the condition for $\alpha = 0$ is if $a_i = \min(a_1, a_2, \ldots, a_n)$, and $a_1 \bmod a_2 = 0, \ldots, a_{i-1} \bmod a_i = 0, a_{i+1} \bmod a_{i+2} = 0, \ldots, a_{n-1} \bmod a_n = 0$.

In fact, we need to count the number of ways to split the elements of the array into two groups, so that within the groups, each number divides every other, and the minimum of the array is contained in the first group.

Let the array contain $k$ distinct elements: $b_1 > b_2 > \ldots > b_k$ in quantities $c_1, c_2, \ldots, c_k$ ($c_i$ copies of the number $b_i$).

Then we can solve the problem using dynamic programming. We will sequentially add $b_1, b_2, ..$ to the groups, initially assuming the groups are empty. The states will be: $dp[i][j]$ — the number of ways if the last element of the first group is $b_i$, and the last element of the second group is $b_j$. The recount is simple; we need to iterate over the previous element and check the divisibility condition. The answer will be $dp[1][1] + \ldots + dp[1][k]$, since the minimum must be in the first group. This dynamic programming approach naively works in $O(k^3)$, but if written a bit more carefully, it can work in $O(k^2)$. This will score some points, but for $n = 3 \cdot 10^5$, it is too slow. However, we can notice an interesting fact: if there is at least one suitable partition, the number of distinct numbers in the entire array does not exceed $2 \cdot \log_2(10^9)$, since if in each group all numbers divide all others, each new distinct number in the group is at least half the size of the previous one, meaning there are at most $\log_2(10^9)$ of them. Thus, if the number of distinct numbers $> 60$, we can immediately conclude that the answer is zero and not compute the dynamics, and for $k \leq 60$, even a solution with a complexity of $O(k^3)$ will be more than fast enough. This is the solution for $\alpha = 0$.

For $\alpha = 1$, three new cases are added in which the sum can equal $\min(a_1, a_2, \ldots, a_n) + 1$.

- In both groups, all also divide each other, and the last element of the first group equals $\min(a_1, a_2, \ldots, a_n) + 1$
- In both groups, all also divide each other, except for one place in one of the groups where $a_i \bmod a_{i+1} = 1$ exactly, and the last element of the first group equals $\min(a_1, a_2, \ldots, a_n) + 1$
- If $\min(a_1, a_2, \ldots, a_n) = 1$, then it is possible to partition into three groups, with the last element of the first two being 1, and all dividing each other.

It is not difficult to understand that there are no other cases when the sum equals $\min(a_1, a_2, \ldots, a_n) + 1$. Each of the three cases needs to be counted separately. This is done using similar dynamic programming. For the first case, we simply need to compute the answer differently, while the dynamic programming is completely analogous to the case $\alpha = 0$. For the second case, we need to add a flag to the DP states — whether $a_i \bmod a_{i+1} = 1$ has been used. For the third case, we need to add a DP state for the last index of the third group. A naive implementation will work in $O(k^4)$, which is fast enough, since if $k > 3 \cdot \log_2(10^9)$ we can still conclude that the answer is 0 without computing the dynamics.