Problem A. Square Illumination

Author and developer problem: Nikolay Vedernikov

To illuminate the entire square, we need to illuminate each side. To illuminate the square the length, we need $\lceil \frac{n}{k} \rceil$, and along the width, we need $\lceil \frac{m}{k} \rceil$. In total, we need to place $\lceil \frac{n}{k} \rceil \cdot \lceil \frac{m}{k} \rceil$ lamp.

Problem B. Battleship

Problem author: Artem Vasiliev, developer: Polina Shaidurova

First, let's check that there are no corner touches between occupied ship cells (i.e., cells with the symbol '#'). If there are none, then all the figures on the field are rectangles of size $1 \times a$, which are arranged vertically or horizontally. It remains to check that there are exactly four «Submarines», three «Destroyers», two «Cruisers», and one «Battleship» This can be done, for example, by counting the number of occupied rectangles of size 1×1, 1×2, 1×3, and 1×4 (there should be 20, 10, 4, and 1 of them, respectively).

Problem C. Carpet Showcase

Problem author: Mikhail Ivanov, developer: Rita Sablina

Let's assume that $h \le w$. We will denote the maximum area of carpets that can be laid on a $h \times w$ platform as $D_{h,w}$.

Let's solve the problem for the case $h = 1$. It is advantageous to build a showcase as follows: on the first layer, place a carpet of size $1 \times w$, on the second layer, place carpets of size 1×1 and $1 \times (w-1)$, and so on - place carpets in w layers, cutting off a carpet of size 1×1 on each layer. The total area will be $D_{1,w} = w + (1 + (w-1)) + (1 + (w-2)) + \ldots + 1 = \frac{w^2+3\cdot w-2}{2}$.

Let's solve the problem for the case $h \ne 1$. It is always advantageous to start laying the showcase with a carpet of size $h \times w$. Then it can be easily proved by induction that to maximize the total area of carpets on the first layer, it is advantageous to cut off a strip of size $1 \times w$ from the rectangle, lay it out further as in the case of $h = 1$, and sequentially cut off strips of size 1 from the remaining carpet of size $(h-1) \times w$ until only a carpet of size 1×1 remains.

For $h = w$, the total area will be:

$D_{w,w} = w \times w + D_{1,w} + D_{w,w-1} = w \times w + D_{1,w} + D_{1,w-1} + D_{w-1,w-1} = w \times w + D_{1,w} + 2 \cdot D_{1,w-1} + \ldots 2 \cdot D_{1,1}$.

The final formula for a square is: $D_{w,w} = w^3 + 2 \cdot w^2 - 2 \cdot w$

For $h \ne w$, we will cut off strips of size $1 \times h$ on each layer until only a carpet of size $h \times h$ remains. The total area will be $D_{w,h} = h \times w + D_{1,h} + h \times (w-1) + D_{1,h} + h \times (w-2) + \ldots + D_{1,h} + D_{h,h}$.

The final formula is: $D_{w,h} = \frac{w^2\cdot h + h^2 \cdot w + 4\cdot(w\cdot h) - 2\cdot h - 2\cdot w}{2}$

Problem D. Redrawn graph

Problem author: Egor Gorbachev, developer: Ivan Volkov

Note that for each of the three participating vertices, the operation either increases its degree by 2 (if it was initially not connected to any other vertex in the triplet), decreases it by 2 (if it was connected to both vertices in the triplet), or does not change it at all (in all other cases). In any case, the parity of the degree of each vertex remains the same. So if there exists a vertex which degree is even in one graph and odd in another, the answer is definitely «NO».

Let's show that in the opposite case, the desired sequence of actions always exists. There are different ways to construct such a sequence. For example, it is sufficient to apply the operation to the triplet $(1, u, v)$

for each edge (u, v) that is present in one graph and absent in the other, and such that $u, v \neq 1$. We will prove that this sequence is the answer to the problem.

Suppose that after performing all such operations, there is an edge that is present in our graph but absent in the one we want to obtain (or vice versa). Obviously, such an edge must have the form $(1, x)$ (we have definitely "fixed" all other edges with our sequence). But the degree of vertex x in our graph has the same parity as in the graph we want to obtain. In particular, the degrees of vertex x cannot differ by one, so the edge $(1, x)$ must either be present in both graphs or absent in both — this leads to a contradiction. Therefore, after performing all the specified operations, we will indeed bring the graph to the required form.

Problem E. Accounting Chaos

Problem author: Fedor Tsarev, developer: Anastasia Barkina

In this task, the goal was to find the count of numbers that appear in the input data at least n times and then output all such numbers. To achieve this, the proposed approach was as follows:

1. Start by sorting the given array.

2. Then, iterate through all the elements in the array, comparing adjacent elements. When adjacent elements were the same, it was necessary to increment the count of numbers in the current group of identical numbers by 1. When adjacent elements were different, you needed to compare the obtained count of numbers in the current group of identical numbers with n. If the count of repeating numbers was no less than n, you should add the repeating number to the output; otherwise, skip it.

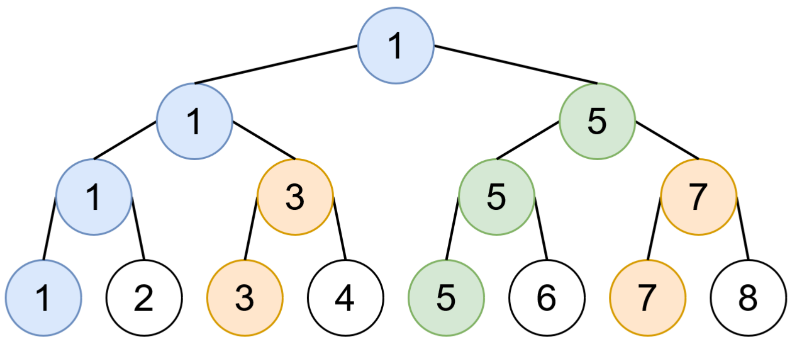
The technical complexity lay in correctly handling the first and last groups of identical numbers to ensure that nothing was lost.

Problem F. Segment Tree

Problem author: Rita Sablina, developer: Grigory Shovkoplyas

In this problem, the original array consists of 2^k elements, therefore the height of the constructed segment tree is k.

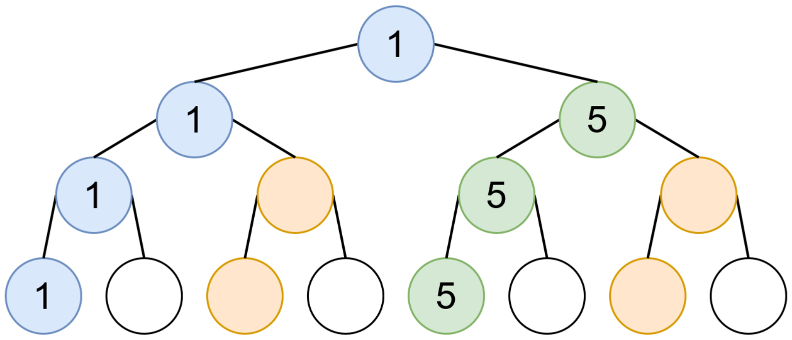
Note that the numbers in the original array are distinct, so if we fill the nodes from bottom to top, each value will correspond to a path from a leaf to some ancestor node.



In the tree containing $2^k - 1$ nodes, there is one path of length k (corresponding to the minimum value of the original array), one path of length $k-1$, two paths of length $k-2$, four paths of length $k-3$, ..., 2^x paths of length $k-x$, ..., 2^{k-1} paths of length 1. We leave the proof of this fact to the reader.

Since all numbers are distinct, by counting the occurrences of each number in the input, we can assign each number a "level"of the tree where the path from bottom to top ends. Then this value must be overwritten by a smaller value.

Next, we can greedily fill the entire segment tree with numbers from the input. For example, we can traverse the tree from top to bottom and fill all the free nodes of the current level with values whose number of occurrences corresponds to that level. For example, let's say for the tree in the previous image we filled the zeroth and first levels, leaving two threes and two sevens, and one occurrence of each of the other values. This means that on the second level, we need to place the values three and seven.



To determine the position to place the value three and the position to place the value seven (it is not always advantageous to place them from left to right in increasing order), we can create an array of pairs of positions and values in the parents on each level, sort it in ascending order of values in the parent, and place the smaller value in the node with the smaller parent value. On each level, it is necessary to check that we indeed have 2^x values with $k - x$ occurrences. If this is not the case on any level, it is impossible to reconstruct the segment tree and therefore the original array.

Finally, it is necessary to check that the final tree is a valid segment tree. The answer can be output in any form, so filling the tree can be done in any way, for example, as shown in the images above, by "pushing"the value to the left child and leaving the right child empty.

The overall complexity of the solution using sorting on each level is $O(n \log n)$.

Problem G. Elevator Ride

Author and developer problem: Ekaterina Vedernikova

Looking at the numbers, you can notice that when reflected, only the digit 2 changes to the digit 5 and vice versa. Therefore, to solve the problem, you need to reverse the floor number, replace the digit 2 with the digit 5 and vice versa. It is important to replace the digits simultaneously. Finally, don't forget to remove leading zeros.

Problem H. Yurik and Important Tasks

Problem author: Daniil Oreshnikov, developer: Mikhail Perveev

To begin with, let's notice that it is convenient to process queries from the end, maintaining the index of the task of interest in the current order of their execution.

After all changes in the order of task execution, we are interested in the task that will be executed as the k-th one. Let's learn how to calculate the position of this task in the list before the last change in the order of task execution.

Let's assume that the last change affected the tasks located in the segment $[l, r]$. If $k \notin [l, r]$, then the last operation did not change the position of the task of interest in the list, so it can be ignored.

Otherwise, let's denote the initial position of the task of interest in the list as i. We know that after the last change, this task moved from position i to position k. This means that the priority assigned to this task was the $(k - l + 1)$-th one in ascending order when numbered starting from one. We also know that the tasks in the segment were assigned priorities $p_1, p_2, \ldots, p_{r-l+1}$.

Thus, in order to determine the priority assigned to the task of interest, it is necessary to find the $(k-l+1)$-th element in ascending order on the prefix of the permutation of length $r-l+1$. Let this element be equal to p_j, then $i = l + j - 1$. Now assign this value to the variable k and proceed to process the penultimate operation of changing the order of task execution. After all operations are processed, the variable k will contain the answer, since before the start of the operations, the tasks were ordered in ascending order of their numbers.

Now let's learn how to quickly find the $(k - l + 1)$-st element in ascending order on a given prefix of a permutation.

To begin with, perform the following preprocessing. We will consider all prefixes of the permutation in ascending order of their length. For each prefix, we want to maintain an array of length n, in which the i-th position will contain one if the element i is present in the current prefix of the permutation, and zero otherwise. In order to maintain this array for each prefix, we will store it as a persistent segment tree, which will maintain a separate version for each prefix of the permutation. This preprocessing can be done in $\mathcal{O}(n \log n)$ time, with $\mathcal{O}(n \log n)$ memory required to store the tree.

Now, in order to find the k-th element in ascending order on a prefix of a given length, we will work with the corresponding version of the segment tree.

Perform a binary search for the answer. Let the binary search fix some number x. In order to check whether x is less than the k-th element in ascending order, it is necessary to find the number of elements on this prefix that do not exceed x, and compare this number with k. In order to find the number of elements that do not exceed x, it is necessary to calculate the sum on the segment $[1, x]$ in the required version of the segment tree. Thus, the answer to the query works in $\mathcal{O}(\log^2 n)$ time. It is also possible to replace the binary search for the answer with a descent on the segment tree and achieve a complexity of $\mathcal{O}(\log n)$.

Final complexity: $\mathcal{O}(n \log n + q \log n)$.

Problem I. Roofs

Author and developer problem: Stepan Filippov

We will solve the problem recursively, let's denote $f(l, r)$ as the answer to the problem if we consider only columns from l to r. Then the task requires calculating $f(1, n)$.

Find m — the position of the tallest column in the segment $[l, r]$. It is clear that we need to attach a roof to it, and it is most advantageous to extend it either as far left as possible, covering the segment $[l, m]$, or as far right as possible, covering the segment $[m, r]$. For the uncovered part, we need to solve the problem recursively. In the first case, the total cost is $c_m + f(m + 1, r)$, in the second — $c_m + f(l, m - 1)$. We calculate both values and choose the minimum.

It is not difficult to understand that the number of recursive calls will be no more than n. For example, it can be noted that each column can be the tallest (denoted above as m) for no more than one segment $[l, r]$ among recursive calls.

When using a data structure that allows efficiently finding the maximum element on a segment, for example, a segment tree, the complexity of the solution is $\mathcal{O}(n \log n)$.

Problem J. Slime Escape

Problem author: Margarita Sablina, developer: Nikita Golikov

Let's construct the following graph: the vertices will correspond to all possible positions of the slime on the table, and the edges will represent transformations. Thus, the task is to find the shortest path from the state "2x2 square" in the top-left corner to the same state in the bottom-right corner. This can be done using breadth-first search.

We will represent the current state of the slime as a pair of its top-left cell and the shifts of the other cells relative to the top-left one. There are a total of 13 possible shifts for the other cells of the slime.

We will precalculate all possible "good" states of the slime: we will iterate through all subsets of size 3 from the 13 possible shifts, and for each subset, we will check if it forms a 4-connected shape together with the top-left corner. It turns out that there are only 19 "good" states. Now we will represent each set of shifts as a number from 0 to 18.

We will precalculate all possible transitions. We will represent a transition as a triple of the shift of the top-left corner of the slime, the initial "good" state, and the final "good" state. For each of the 19 states, we will iterate through all cells of the slime and try to replace them with all neighboring cells in terms of 8-connectivity. If we transition to a "good" state, we will say that such a transition is possible. It turns out that there are only 104 possible transitions.

Having precalculated all possible transitions, we are ready to implement breadth-first search. We will check in advance for each top-left corner on the table and "good" state that all cells for that state do not fall into the holes. Now we just need to implement the standard breadth-first search.

Thus, after precalculation, the solution works in $\mathcal{O}(nmT)$ time, where T is the number of possible transitions, which, as described above, is equal to 104. The solution implemented in this way fits comfortably within the time limit. Less efficient solutions (for example, without precalculating all possible states and/or transitions) may not fit within the time limit.

Problem K. Production Waste

Problem author: Fedor Tsarev, developer: Daniil Oreshnikov

Let's note the following fact: if the plants are idle for a certain period of time, then knowing the amount of waste at the beginning of this period, we can quickly calculate the amount of waste on any other day within that period. Specifically, if there are x units of waste at the end of day a, and no new raw material is used during the days from $a+1$ to b, then at the end of day b, the amount of waste will be exactly $x \cdot (1-r)^{b-a}$.

Using this fact, the problem could be solved using one of the following approaches: binary search, two-pointer method, or event sorting. Let's consider the third approach. We will combine the days on which the plants operated and the days of interest to the inspection into a single sequence of events. For the former, we will create events of the form $(d_i, +, w_i)$, and for the latter, we will create events of the form $(q_i, ?, i)$. Then, we sort all events by the day number, and in case of a tie, we prioritize events of type + before events of type ?.

Now, we will process the events one by one in the order they were sorted. For each event, we will keep track of the previous event day d_{prev}, and maintain t as the amount of waste at the end of day d_{prev}. Then,

- To process an event of the form $(d, +, w)$, we recalculate t as $t \cdot (1-r)^{d-d_{\text{prev}}} + w \cdot r$.
- To process an event of the form $(q, ?, i)$, we recalculate t as $t \cdot (1-r)^{d-d_{\text{prev}}}$, and store t as the answer to the i-th query.

After that, we simply update d_{prev} to the day of the current event, and move on to process the next event. We can calculate powers of $1-r$ using the binary exponentiation algorithm, which is based on the fact that $x^{2k} = (x^k)^2$. This allows us to compute the k-th power of a number in $\mathcal{O}(\log k)$ time. The time complexity of this solution is $\mathcal{O}((n+m)\log\max(\max(d), \max(q)))$.

Problem L. Seats in the subway

Problem author: Nikolay Vedernikov, developer: Vladimir Smaglii

To solve this problem, it is necessary to maintain two ordered sets of segments (it is recommended to use the corresponding set implementation in your language). In the first set, the segments are compared based on the maximum distance that can be obtained on them (for the leftmost and rightmost segments, this value is equal to their length, as it is possible to sit on the edge; in other cases, it is $\lfloor \frac{len}{2} \rfloor$), and if the distances are equal, then the right boundaries are compared (for this, you need to use the corresponding comparator or redefine the comparison operator). In the second set, the segments are compared based on the left boundary.

To add a passenger, it is necessary to extract the last element from the first set and divide it into two halves, removing the middle seat, or remove the extreme seat if the segment is the leftmost or rightmost.

When freeing up seat Y, we search for two segments in the second set - the nearest one on the left and the nearest one on the right. Then, three cases are possible:

1. The nearest segments do not border Y. It is necessary to add a segment from Y to Y.
2. Exactly one of the segments borders Y. It is necessary to attach Y to this segment and update its boundaries.
3. Both segments border Y. It is necessary to merge these two segments into one large segment.

All updates to the segments are made simultaneously in both sets to maintain data consistency and compatibility.