

Problem A. Stones and Bananas

Author	Alexander Babin
Developer	Peter Losev
Topic	Breadth-First Search

If two cells are on the same piece of paper after the cells with stones have been burned, it means there is a simple path connecting one cell to the other, connecting these two cells and not visiting any of the cells with stones. The statement is also true in the reverse direction, as the cells of the path themselves form a connected set.

Therefore, we will try to find a simple path between cells (x_1, y_1) and (x_2, y_2) that will not pass through cells with stones after moving the stones. In the problem, it was allowed to move no more than k stones, so in the original field, the path from cell (x_1, y_1) to cell (x_2, y_2) cannot pass through more than k stones.

But there is an additional condition: if the found path visits too many cells, it will not be possible to *clear* it, as there will simply be nowhere to put the stones on that path. Specifically, if there are m free cells in the original table, then the sought path cannot pass through more than m cells.

To determine the existence of such a path, we will combine ideas from dynamic programming and breadth-first search. Specifically, we propose to compute $d[t][i][j]$ — the shortest path from cell (x_1, y_1) to cell (i, j) , passing through no more than t stones. To compute the values of this array for a given t , it is sufficient to write a breadth-first search with the following transition:

- When transitioning from the triplet (t, i, j) to a neighboring cell (i', j') , we should set $t' = t$ if (i', j') does not contain a stone and $t' = t + 1$ otherwise.
- In all other details, the algorithm does not differ from a standard breadth-first search.

Next, we need to check that the found path is short enough, and in case of success, reconstruct the path itself, the stones that lie on it, and the free cells that lie outside this path. After that, the answer to the problem can be easily reconstructed. We obtained a solution with a time complexity of $O(n^2k)$.

Problem B. Code Lock

Author	Alexander Zavarin
Developer	Alexander Zavarin
Topic	Dynamic Programming

Each set of segments can be *scanned* from left to right, maintaining k — the number of segments that contain the current element i . During this process, it is also possible to maintain the total number of segments: when moving from i to $(i + 1)$, if the value of k increases by x — this means that the total number of segments has increased by x , as x new segments have opened at position $(i + 1)$; and when it decreases — the total number of segments does not change, as previously counted segments are closing.

We will compute the dynamic values $dp[i][k]$ — the minimum number of segments required to set the correct values $a_1, \dots, a_i$, given that element i is covered by exactly k segments. This array should be computed layer by layer to save memory.

It is easy to show that the answer does not exceed $M = \sum_{i=1}^n (m_i - 1)$, so we are only interested in values of k in the segment $[0, M]$. Transforming $dp[i - 1]$ into $dp[i]$ can be done with three linear steps:

1. Copy the array $dp[i - 1]$ into the array $dp[i]$.
2. In order $k = 1, 2, \dots, M$, perform the update $dp[i][k] \leftarrow \min\{dp[i][k], 1 + dp[i][k - 1]\}$;

3. In order $k = M - 1, M - 2, \dots, 0$, perform the update $\text{dp}[i][k] \leftarrow \min\{\text{dp}[i][k], \text{dp}[i][k + 1]\}$;
4. Filter $\text{dp}[i]$, specifically set $\text{dp}[i][k] \leftarrow \infty$ for all $k \not\equiv a_i \pmod{m_i}$.

The second step handles the possibility of opening new segments, the third handles closing segments, and the fourth imposes the condition that the required number will be entered at position i of the lock. It is easy to show that the first step can be omitted and all calculations can be performed on the same array.

As a result, we obtain a solution with a time complexity of $O(n \cdot \sum m_i)$ and a space complexity of $O(\sum m_i)$.

Problem C. Good Colorings – 7

Author	Babin Alexander Romanovich
Developer	Babin Alexander Romanovich
Topic	Constructives and Spanning Trees

Consider the bipartite graph $G = (R, C, E)$, where the left part consists of the rows of the table $R = \{1, 2, \dots, n\}$ and the right part consists of the columns $C = \{1, 2, \dots, n\}$. An edge from row $i \in R$ to column $j \in C$ will be drawn if the cell (i, j) is white.

In $O(n^2)$ time, we will also compute the following values:

- $r(i)$ the number of red cells in row i ($1 \leq i \leq n$)
- $c(j)$ the number of red cells in column j ($1 \leq j \leq n$)
- d_1 the number of red cells on the main diagonal
- d_2 the number of red cells on the secondary diagonal

Suppose we have painted the white cells from the set U' red, and the others blue, while satisfying the conditions of the problem. Then the cells in U' correspond to a set of edges $U \subseteq E$ in the graph G , for which the following conditions hold:

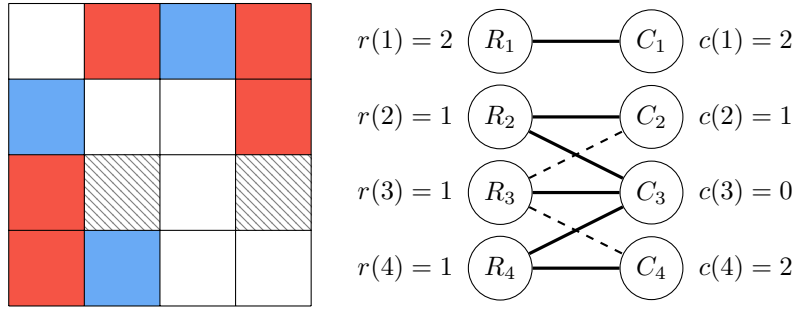
- For any row vertex $i \in R$, the parity of the number of incident edges from U matches the parity of $r(i)$.
- For any column vertex $j \in C$, the parity of the number of incident edges from U matches the parity of $c(j)$.
- The condition corresponding to the parity of the number of red cells on the diagonals after repainting the white cells cannot be formulated in terms of the graph G . We will discuss it later.

Thus, it is sufficient to find any suitable set $U \subseteq E$, and the desired coloring will be found. The conditions on the rows and columns impose conditions on the parity of the vertices in the graph $G_U = (R, C, U)$, from which we can immediately derive the following restriction:

Claim 1. Summing the values $r(i)$ and $c(j)$ in any connected component of the graph G . If the resulting value is odd, then it will not be possible to find a set U .

Proof. By contradiction, suppose a suitable set U exists, then consider C — the component of the graph G where the sum of $r(i)$ and $c(j)$ is odd. Then on one hand, the total degree of the vertices from C in the graph G_U is even, while on the other hand, it must be odd, as it coincides with the sum of $r(i)$ and $c(j)$. This leads to a contradiction.

Identifying connected components and checking this condition for them can be done in $O(n^2)$ time. If the condition is not satisfied for at least one component, then we should output "No" and finish processing the test case.



Example of coloring and the corresponding graph $G = (R, C, E)$,
rigid edges are highlighted in bold, free ones are dashed.

In case the specified condition holds for all components of the graph, we will select any *spanning forest* T in the graph G , and note another fact:

Claim 2. For any set of edges $A \subseteq E \setminus T$, there exists a unique set of edges $B \subseteq T$ such that $U = A \cup B$ satisfies the condition on the parity of the degrees of the vertices.

In other words, after identifying the spanning tree, all edges are divided into two classes — *rigid* (which belong to the spanning tree) and *free* (which do not belong to the spanning tree). First, for each *free* edge, we can freely choose whether it will be included in the set U or not, after which the membership of each *rigid* edge in the set U can be uniquely restored.

In any case, we will have to write code that restores the set B from the set A , so we need to figure out how to do this. For each vertex $v \in R \cup C$, it is easy to compute $s(v) \in \{0, 1\}$ — the parity of the number of edges from A that are incident to it with a fixed B .

Notice that if we choose any path $P \subseteq T$, and then invert the edges on this path in the set A (that is, remove the edges that were there and add those that were not), the degrees will change for exactly two vertices — the endpoints of the path P . Therefore, we can, for example, select a root in each component, hang the spanning tree from it in this component, and then simply invert the paths. With an efficient implementation, this takes $O(n^2)$ time.

Proving uniqueness is also not difficult; it is sufficient to consider two suitable sets A_1 and A_2 and note that in the graph constructed on the edges $A_1 \Delta A_2$ (where Δ denotes the symmetric difference), all degrees are even and there are no cycles, from which it follows that $A_1 \Delta A_2 = \emptyset$.

Thus, we have learned to easily generate sets of edges U , and therefore colorings that satisfy the first two conditions. We just need to figure out how to satisfy the conditions for the diagonals. There are two approaches here.

First approach. (undetermined) We will identify the spanning tree T , then include each free edge with a probability of $\frac{1}{2}$ in the set B , after which we will construct the set A , and then restore the coloring. It is claimed that if a solution exists, it will be obtained with a probability of $\geq \frac{1}{4}$. We repeat this procedure 60 times, and if we do not find a solution even once, we assume that it does not exist. The probability of not finding an answer when it exists is approximately $3 \cdot 10^{-8}$.

Second approach. We will identify the spanning tree T and construct the set A_0 from the empty set $B = \emptyset$. Now, if we add or remove an edge $e = \{u, v\}$ to the set B , obtaining $B' = \{e\} \Delta B$, only the degrees of 2 vertices change. Let the path P connect these two vertices, then the set A changes to $A' = A \Delta P$ with this change in B .

To track the parity of the diagonals, we will associate each edge e (both rigid and free) with a 2-bit mask $x(e)$: the 1st bit of this mask is equal to one if the cell lies on the main diagonal, the 2nd bit is equal to one if the cell lies on the secondary diagonal.

Then the last requirement for the coloring in terms of choosing the set of edges U is formulated as follows:

$$x(U) = x(e_1) \oplus x(e_2) \oplus \dots \oplus x(e_k) = (d_2 \bmod 2) \cdot 2 + (d_1 \bmod 2), \quad U = \{e_1, \dots, e_k\}$$

Here $\oplus$ denotes the bitwise XOR operation. Now, when adding or removing an edge $e' = \{u, v\}$, the value of $x(U)$ changes as follows:

$$x(U') = \bigoplus_{e \in A' \cup B'} x(e) = \left[\bigoplus_{e \in U} x(e) \right] \oplus \underbrace{\left[\bigoplus_{e \in P} x(e) \right]}_{=y(e')} \oplus x(e').$$

The values $y(e)$ for all free edges e can be easily computed in $O(n^2)$ time. But then we arrive at such a formulation: initially, there is some $x(U)$, and we can XOR this value with some $y(e)$, and we need to obtain 0.

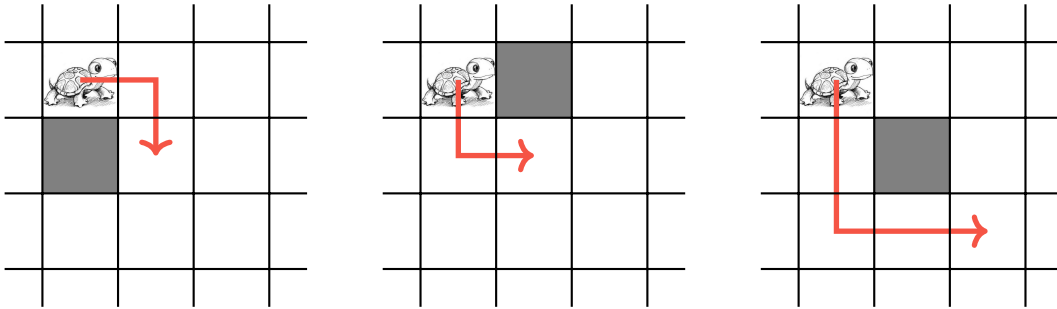
However, we are dealing with two-bit masks, so we can either break the problem into cases or write a brute force solution. Thus, it is possible to find the required set of edges from B in $O(1)$ time.

After that, all sets A , U , and then the desired coloring are restored sequentially.

Problem D. How to Annoy a Turtle

Author	Babin Alexander
Developer	Babin Alexander
Theme	Constructive

First, note that the turtle can always move from cell (x, y) either to cell $(x+1, y+1)$ or to cell $(x+2, y+2)$. There are a total of 3 cases illustrated below:

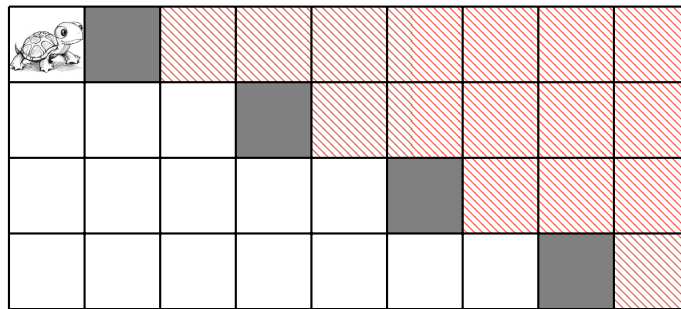


Without loss of generality, assume that $m \geq n$. If $m \leq 2n$, then the turtle can always reach cell $(1, 1)$ to cell (n, m) using a greedy algorithm:

- If the turtle is in cell (x, y) and $y - x = m - n$, then it moves along this diagonal to cell $(x+1, y+1)$, and if it cannot, then to cell $(x+2, y+2)$.
- If $y - x < m - n$, then the turtle tries to move to cell $(x, y+1)$.
- If it fails, it can move to cell $(x+1, y+2)$.

It is easy to see that considering $m \leq 2n$, the turtle will be able to reach cell (n, m) , as in the worst case it moves to the adjacent diagonal, descending one cell down.

On the other hand, in the case $m > 2n$, it is easy to provide an example of how to place obstacles to block the turtle's path. It is sufficient to block cells of the form $(k, 2k)$.



The case where $n > m$ is considered in a completely analogous manner.

Problem E. Fight Club

Author	Alexey Vasilyev
Developer	Alexey Vasilyev
Topic	Stack, prefix maxima

Let the segment $[l, r]$ be *honest* and $a_l < a_r$, then a_r is the nearest greater element to the right of a_l , since all elements $a_{l+1}, \dots, a_{r-1}$ are less than a_l . Similarly, if $a_l > a_r$, the element a_l is the nearest greater element to the left of a_r .

Conversely, suppose that the element a_l has a nearest greater element to the right, which is a_r . Then there are no elements greater than a_l before a_r , and thus the segment $[l, r]$ is *honest*. The same reasoning holds if a_l is the nearest greater element to the left of a_r .

Thus, the number of honest segments is equal to the number of elements for which there are greater elements to the left, plus the number of elements for which there are greater elements to the right.

We will learn to maintain the count of elements for which there are greater elements to the left. For this, we will maintain those elements that are prefix maxima. That is, these are elements for which there are no greater elements to the left. We will maintain their indices in a stack from left to right. When an element a_i becomes the new maximum, it removes all prefix maxima to its right, keeps the previous prefix maxima to its left, and makes itself a prefix maximum. Thus, we can easily maintain this stack with an amortized time complexity of $O(n + q)$.

Therefore, we need to maintain two stacks (prefix maxima and suffix maxima), and the answer to the problem will be $2n - cnt_{pref} - cnt_{suf}$, where cnt_{pref} is the size of the prefix maxima stack, and cnt_{suf} is the size of the suffix maxima stack.

Problem F. Arsen and Toy Soldiers

Author	Ho Dang Zung
Developer	Ho Dang Zung
Topic	Eulerian Graphs

Note that a necessary condition for the balance is that each type of rifle must be issued an even number of times both in the morning and in the evening. This can be easily achieved or checked for impossibility using a greedy algorithm in linear time.

Next, consider a bipartite graph where the left part consists of the types of rifles issued in the morning, and the right part consists of the rifles issued in the evening. The edges will represent the soldiers: thus, a soldier who received rifle u in the morning and rifle v in the evening will be represented by an edge connecting vertex u in the left part and vertex v in the right part.

It can be noted that each component of the constructed graph is Eulerian, as all vertex degrees are even. Therefore, in each component, an Eulerian cycle can be found, and all edges can be oriented according to this cycle. Thus, from each vertex, the same number of edges will exit as enter it.

This fact can also be used for the final stage of solving the problem — distributing soap. We will give soap to those soldiers whose corresponding edges are directed from the right part to the left, and we will not give soap to the others. It is easy to see that the problem's conditions are satisfied.

The presented solution can be easily implemented in $O(n)$ time. The long runtime in practice is due to the fact that this algorithm makes many random memory accesses, which often leads to cache misses.

Problem G. Geometry!

Author	Alexander Ponkartov
Developer	Anton Stepanov
Topic	Angle sorting, segment tree

If we take two polygons A and B , sort their edges by angle, and perform the usual merging procedure of two sorted sequences, we will obtain the polygon $F(A, B)$: it will sequentially contain all edges from A and B with the same lengths and orientations as the originals.

A vertex in $F(A, B)$ is counted by the function $G(A, B)$ if the two incident edges come from different polygons. Equivalently, if we look at the circular order of all edges from A and B after merging, $G(A, B)$ is the number of transitions between the colors "A" and "B".

Let the edges of A in this order be divided into s_A continuous segments, and the edges of B into s_B segments. Then each internal adjacency of two edges from A generates a vertex that does not fall into G . There are $k_A - s_A$ such adjacencies, similarly for B . The total number of vertices in $F(A, B)$ is $k_A + k_B$, so

$$G(A, B) = (k_A + k_B) - (k_A - s_A) - (k_B - s_B) = s_A + s_B.$$

Thus, for each pair of polygons, it is only important how many times the edges of one of them are *broken* by the edges of the other.

Now consider all polygons at once. We will collect all their edges into one array, sort it by polar angle, and record for each edge which polygon it belongs to.

Now for any polygon A , we will take its positions in the sorted array and look at the neighboring pairs (p_j, p_{j+1}) (where $p_k = p_0$ for closing the circle). If edges from some polygon B are encountered between these two positions, then it is exactly B that breaks the sequence of A , and the contribution to the answer from the pair (A, B) increases by 1. By repeating this procedure for all polygons, we sum s_A for all A and automatically obtain the sum $G(A, B)$ for all pairs.

It remains to efficiently count the number of **different** polygons that appeared between each pair p_j and p_{j+1} . This is a classical query problem of the type "count distinct in a segment," which can be solved "offline" using a segment tree and sorting queries by their right boundary. The total time complexity is $O(m \log m)$, where m denotes the total number of vertices across all polygons.

Problem H. Tram System

Author	Alexander Babin
Developer	Alexander Babin
Topic	Palindromes, Tables

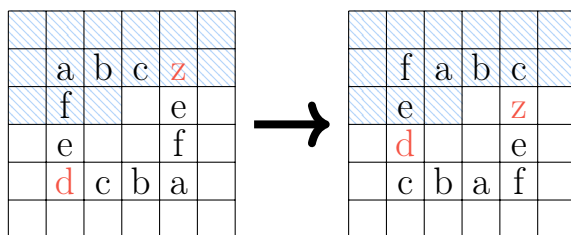
The $n \times n$ table, as per the conditions, is divided into tram tracks, which we will simply call *squares*. It is clear that any final configuration of trams can be obtained by sequentially *rotating* the squares, i.e., applying cyclic shifts clockwise or counterclockwise.

Let's note the *central symmetry* in the representation. Thus, the cell (x, y) will be centrally symmetric to the cell $(n - 1 - x, n - 1 - y)$. It is easy to see that after writing the table *in a row*, the symbols in these cells will fall into symmetric positions.

The value of *mysticism* is precisely the position of the first symbol i , after writing in a row, which does not match its symmetric symbol, i.e., the symbol at position $n^2 - i - 1$.

Notice that with any rotation of the squares, the centrally symmetric symbols in the table will remain centrally symmetric to each other, so it makes sense to pair them based on this principle, separating pairs into *good* and *bad*, where the symbols in the pair match and do not match, respectively.

Now we will write the value 1 in the cells of the table that belong to good pairs, and in the remaining cells, we will write the value 0. The problem has been reduced to maximizing the position of the first 0 after writing the table in a row.



An image to understand what is happening. We found the longest segment **efabc** and moved it so that the bad pair of symbols **z-d** meet as late as possible after writing everything in a row.

Notice that if we mark the first k elements of the table, they will carve out continuous sections on each of the squares. But then we can independently compute the maximum number of consecutive 1s for each square (considering that the square forms a cycle), and then cyclically shift this square so that the first 0 on this square appears as late as possible in the table. In the end, we have a solution with a time complexity of $O(n^2)$.

Important comment. Of course, the described solution assumes some technical details related to correctly processing each of the squares, but the implementation can be significantly simplified if we do not explicitly select the cyclic shift, but instead simply iterate through the elements of the table and stop at the moment when the square of the current cell encounters a greater number of times than the maximum number of consecutive 1s on that square.

Problem I. Rick's Grades

Author	Mikhail Prokhorov
Developer	Anton Nuykin
Topic	Sorting

Replace all occurrences of 0 in the array $b_1, \dots, b_n$ with 6. After this transformation, it is necessary to shuffle the elements of array b in such a way that $a_i \leq b_i$ for each i from 1 to n , and then replace all occurrences of 6 in array b back to 0.

If a , the array of Morty's grades, is sorted in ascending order, it is beneficial to also sort the array b , Rick's grades after replacing 0 with 6, in ascending order. However, the order of elements in a cannot be changed, so we should sort array a while remembering the original positions of each element. A solution in $O(n \log n)$ looks as follows:

```
t = int(input())
for _ in range(t):
    n = int(input())
```

```
a = list(map(int, input().split()))
b = list(map(int, input().split()))

b = sorted([6 if x == 0 else x for x in b])
u = sorted([(a[i], i) for i in range(n)])

ans = True
res = [0] * n
for i in range(n):
    x, p = u[i]
    if x <= b[i]:
        res[p] = 0 if b[i] == 6 else b[i]
    else:
        ans = False
        break
if ans:
    print(" ".join(map(str, res)))
else:
    print(-1)
```

There is also a more efficient solution that works in $O(n)$ time and uses the fact that the numbers involved in the sorting belong to the interval $[0, 6]$. This allows for digit sorting or an efficient implementation of another greedy approach: "Assign each of Morty's grades the smallest, previously unused grade from Rick."

Problem J. Exercise for Dania

Author	Andrey Parkhaev
Developer	Andrey Parkhaev
Topic	Number theory, trie

For brevity, we will use the notation $f(x, y) = M\left(\frac{xy}{\gcd(x, y)^2}\right)$.

We will consider each a_i as an ordered sequence of primes $p_1 \leq p_2 \leq \dots \leq p_k$ from the prime factorization of a_i (i.e., $a_i = p_1 \cdot p_2 \cdot \dots \cdot p_k$). Now let's understand what we are trying to count in terms of these sequences.

In simple terms, $f(a_i, a_j)$ is the smallest prime divisor that appears in the factorizations of a_i and a_j a **different** number of times. On the other hand, if we look at the corresponding prime sequences for a_i and a_j :

$$a_i = q_1 \cdot q_2 \cdot \dots \cdot q_m, q_1 \leq q_2 \leq \dots \leq q_m$$
$$a_j = p_1 \cdot p_2 \cdot \dots \cdot p_k, p_1 \leq p_2 \leq \dots \leq p_k$$

If t is the length of the longest common prefix of the sequences p and q , then $f(a_i, a_j) = \min\{p_{t+1}, q_{t+1}\}$. In the case where t matches the length of one of the sequences, the $(t+1)$ -th element of the other sequence is taken as the minimum. If t matches the lengths of both sequences, then $f(a_i, a_j) = 0$.

For example, if $a_i = 12$ and $a_j = 20$, then their corresponding sequences are 2, 2, 3 and 2, 2, 5. The longest common prefix of such sequences is 2, and the value of the function for a_i and a_j is $\min\{3, 5\} = 3$.

To utilize the observation above, we will create a data structure that can handle the following queries:

- Add a number x to the structure;
- For y , compute the sum of values $f(x, y)$ for all previously added numbers x .

Each added number x will be treated as a string $p_1 \leq p_2 \leq \dots \leq p_k$ (its prime factorization), and we will also maintain a trie on all such strings.

To answer the second query for a given y , we will descend through the corresponding prime sequence for the number y . If we are at some vertex of the trie v and want to descend by the symbol p (the next prime divisor of y), we can immediately process all added x whose path in the trie coincides with the path of y up to vertex v , but then diverges.

Let there be transitions from vertex v by symbols $q_1 < q_2 < \dots < q_l$ to subtrees of sizes $s_1, \dots, s_l$. Then we can compute the sum $f(x, y)$ for all considered x :

$$\sum f(x, y) = p \cdot \sum_{q_i > p} s_i + \sum_{q_i < p} q_i s_i$$

Thus, the sum of $f(x, y)$ for all x that have the given common prefix with y reduces to two segment sum queries. To efficiently answer the sum queries, we should pre-build a trie for **all** values $a_1, \dots, a_n$, sort the outgoing edges from each vertex, and build a Fenwick tree for summing values s_i and $q_i s_i$.

Let $\max\{a_1, \dots\} = C$, then since each number contains at most $\log_2 C$ prime divisors, the resulting solution works in time $O(n \log^2 C)$. It is worth mentioning that other implementations using implicit segment trees or even Cartesian trees were also allowed.

Problem K. Squirrel and Steps

Author	Alexander Babin
Developer	Alexander MacNeil
Topic	Dynamic Programming

The problem can be solved using dynamic programming techniques. We will compute the values $\text{dp}[i][a][b]$ — the number of routes for the cat that satisfy the following conditions:

- All paws are on steps i , $i - 1$, or $i - 2$.
- There are a paws on step i , b paws on step $(i - 1)$, and $4 - a - b$ paws on step $(i - 2)$.
- It is not fixed which specific paws are on steps $i, i + 1, i + 2$, but pairs of routes that lead to at least one step having different sets of paws (for example, in one route the left front paw is on step i , and in another the right back paw is) are considered different.

It is clear that it makes sense to consider states where $1 \leq i \leq \max n$, $1 \leq a \leq 4$, and $0 \leq b \leq 3$ and $a + b \leq 4$, resulting in a total of $10 \cdot \max n$ dynamic states.

For convenience, the family of routes considered in one DP state is denoted by the quadruple (i, a, b, c) ($c = 4 - a - b$). To recalculate such dynamics, we need to consider 3 types of transitions:

- If $c = 0$, then the transition $(i, a, b, 0) \rightarrow (i + 1, 1, a - 1, b)$ occurs with a multiplicity of a .
- If $c > 0$, then the transition $(i, a, b, c) \rightarrow (i, a, b + 1, c - 1)$ occurs with a multiplicity of c .
- If $b > 0$, then the transition $(i, a, b, c) \rightarrow (i, a + 1, b - 1, c)$ occurs with a multiplicity of b .

After computing the DP, we can answer each query in $O(1)$ using the values $\text{dp}[n][4][0]$. Of course, all calculations should be performed modulo $10^9 + 7$.

Problem L. Vector Magic

Author	Babin Alexander
Developer	Grekoa Daria
Topic	Sorting

Note that the expression $\sqrt{b_1^2 + \dots + b_n^2}$ can take exactly $n + 1$ different values.

Indeed, if among the numbers $b_1, \dots, b_n$ exactly k numbers take the values ± 3 , and the remaining $n - k$ take the values ± 1 , then $\sqrt{b_1^2 + \dots + b_n^2} = \sqrt{n + 8k}$.

For a fixed k , it is easy to maximize the value of $a_1b_1 + \dots + a_nb_n$. It is clear that for each i ($1 \leq i \leq n$), the sign of b_i should match the sign of a_i (it is beneficial for each term to be non-negative). Thus:

$$a_1b_1 + \dots + a_nb_n = |a_1| \cdot |b_1| + |a_2| \cdot |b_2| + \dots + |a_n| \cdot |b_n|, \quad |b_i| \in \{1, 3\}$$

It remains to note that it is beneficial to choose $b_i = 3$ at those k positions where $|a_i|$ is maximal.

Thus, the solution reduces to the following:

- Use sorting to find $p_1, \dots, p_n$ — a permutation of indices such that $|a[p_1]| \geq \dots \geq |a[p_n]|$
- Compute all prefix and suffix sums in the array $|a[p_1]|, \dots, |a[p_n]|$.
- Iterate over the value of k and compute q_k for each k ($0 \leq k \leq n$):

$$q_k = \frac{3 \cdot (|a[p_1]| + \dots + |a[p_k]|) + (|a[p_{k+1}]| + \dots + |a[p_n]|)}{\sqrt{n + 8k}}$$

- Among $q_0, \dots, q_n$, select the maximum q_k and reconstruct the answer.

The precision of the type `double` is sufficient for the jury tests, so it can be used in the final stage of calculations when dividing by the radical $\sqrt{1 + 8k}$. However, the problem can be solved with integers using a 64-bit integer type, employing the following trick:

$$a, b, c, d > 0: \frac{a}{\sqrt{c}} < \frac{b}{\sqrt{d}} \iff a^2d < b^2c$$

All computed values using this trick do not exceed:

$$(\max |a_i| \cdot n)^2 \cdot 9n = 9n^3 (\max |a_i|)^2 \leq 9 \cdot 10^{12} \cdot 10^6 < 9 \cdot 10^{18} < 2^{63}$$