Problem A. Awesome Numbers

Since the sum of two odd numbers is even, the only way to obtain a prime number as the difference of the squares of two primes is when the second prime number is 2. Thus, awesome numbers are prime numbers of the form $p^2 - 4$. However, since $p^2 - 4 = (p + 2)(p - 2)$, this is only possible when $p = 3$, meaning the corresponding number is 5.

So if there is a 5 between l_i and r_i, we output 1; otherwise, we output 0.

The term "awesome" could also serve as a small hint, based on the the high school experience in some countries.

Problem B. Buildings And Fishermen

Let's construct a graph where for all i, we draw a directed edge from a_i to b_i. Since each number appears a total of 2 times, the degree of each vertex in the graph is 2, and we will obtain a set of cycles. For each cycle, we can solve the problem independently. Let's formulate the problem on a cycle as follows: we need to orient the edges of the cycle in such a way as to maximize the total score of the cycle, where orienting the edge counterclockwise will mean that we are swapping the elements. We find that if two edges exit from vertex v, we need to add c_v; if one edge enters and one exits, we need to add b_v; and if two edges enter, we need to add a_v.

To determine the maximum score for a cycle, we take any pair of adjacent vertices in the cycle and iterate over the direction of the edge between them (let's say this direction is $dir_0 = 0/1$). Suppose we have determined the direction of the edge between the last vertex and the first. We will run the following dynamic programming:

$dp(pos, dirNext)$ – the maximum total contribution that the first pos elements of the cycle give, given that the direction to the next vertex is $dirNext(0/1)$. We will say that the direction is 1 if we orient the edge counterclockwise.

To recalculate, we need to iterate over $dirPrev$ – the direction of the edge from vertex $pos - 1$ to vertex pos. We will make the following update to our dp:

$dp(pos, dirNext) \leq dp(pos-1, dirPrev)+cost(pos, dirPrev, dirNext)$. Here, $cost(pos, dirPrev, dirNext)$ denotes the contribution of the element pos if the direction of the edge from the previous to the current is $dirPrev$, and from the current to the next is $dirNext$. For example, if $dirPrev = 0$ and $dirNext = 1$, the contribution will be a_v. The answer will be in $dp(cnt, dir_0)$, where cnt is the number of vertices in the cycle.

Problem C. City and Parade

Let's first sort the edges by width from largest to smallest, and then we will add them one by one and merge the connected components containing the endpoints of the edges in a disjoint set system. When merging two components, we will add the product of the number of special vertices in these components to the answer. To answer the queries, we need to remember the states of the answer variable at all moments in time that we have processed, and output the answer from the moment in time when all edges with a weight greater than or equal to the specified weight have been added. For this, we can use binary search.

Problem D. Debt Calculation

Let's define sum_i — the debt that we would have accumulated from the 1st hour to i. Notice that $sum_i = sum_{i-1} \cdot k_{i-1} + a_i$. Now let's introduce $prod_i = \prod_{j=1}^{i-1} k_i$. Notice that sum_r — is the debt over the segment from l to r plus the debt over the segment from 1 to $l - 1$, multiplied by $prod_r/prod_l$, thus to answer the queries we need to precompute the arrays sum and $prod$, as well as learn how to find the modular inverse, which is done by raising to the power of $mod - 2$.

Problem E. Extremelly Thin Backpack

Let's consider a simpler restriction on the volumes of the items. Suppose that $3 \cdot v_i > V$ for any i. From this, we can conclude that no matter which three (or more) items we take, they will not fit in the backpack. Thus, only one or two items can fit in the backpack, meaning we only need to consider these two cases.

The first case is quite simple – we need to choose the item with the maximum value (according to the condition, the sizes of all items do not exceed V).

To solve the second case, we will sort all items by volume and iterate through the first selected one. The volume of the first item does not decrease since we have sorted everything. Thus, the size of the second folder will not increase. We will set a pointer to the rightmost second folder that fits with the current first one. When considering the next first folder, the pointer can only move left (it can shift any number of positions), meaning this part of the solution works in linear time. We will also need to precompute the prefix maximum values (we may also need second prefix maximums – this will happen if the first maximum is our first folder).

Returning to the original restriction. Suppose we took k folders. Then the total size of the taken k folders is greater than or equal to the following value:

$\frac{1}{3} \cdot (k \cdot V - 100 \cdot k)$.

It also does not exceed the size of the tablet, and we get the inequality:

$\frac{1}{3} \cdot (k \cdot V - 100 \cdot k) \leq V$.

From this, we have:

$k \leq \frac{3 \cdot V}{V - 100}$.

Let's see in which cases this value is greater than three. Expressing V, we find that $V \leq 400$.

Thus, we have obtained an important result: we need to take more than three folders only for small tablet sizes! Therefore, for $V \leq 400$, we can write standard dynamic programming, while for $V > 400$, we will solve the case with one and two folders as described at the beginning of the analysis. We still need to consider the case $k = 3$.

From the inequality in the restriction, it follows that $v_i \geq \frac{V}{3} - 34$. We will call a folder light if $v_i \leq \frac{V}{3}$, and heavy otherwise. At the same time, the deviation of a heavy taken folder from $\frac{V}{3}$ does not exceed 68 (if we have already taken two light ones, for example). Let's iterate through the deviations of the three taken folders in a loop from -34 to 68, so that their sizes do not exceed V. We will precompute the three most valuable folders for each size value. With this information, we can reconstruct the answer.

Complexity: $O(N \cdot V)$ (if $V \leq 400$) and $N \log N$ otherwise.

Problem F. From One Isotope To Another

At first, this problem looks like maintaining a Kruskal reconstruction tree. Assume that all vertices initially have a parent of $n + 1$. Then we consider the impact of adding an edge on the current Kruskal reconstruction tree.

Let an edge (x,y) be added, then the LCA in the current tree will be z. If x=z or y=z, no operation is required. Otherwise, simulating the Kruskal process, one can notice that it is necessary to merge the chains (x,z) and (y,z) in the order of vertex numbers.

How to merge them? Direct heuristic merging may turn out to be lengthy. The following merging method is proposed: after merging, those that originally belonged to one chain are called continuous segments, and it is obvious that knowing the partition into segments allows for merging with a time complexity of $O(C \log C)$, where C is the number of continuous segments. Finding the segments themselves can also be done in such complexity. We will further prove that the sum C is $O(n \log n)$.

Let the potential function be defined as $w_i = \log(fa_i - i)$, initially $\sum w_i = O(n \log n)$, it is only necessary to prove that each operation reduces the potential by at least $O(C) - O(\log n)$.

Assume that the continuous segments on the two chains are denoted as

$[x_1, y_1], [x_2, y_2], \ldots, [x_C, y_C], [l_1, r_1], \ldots, [l_C, r_C]$.

The initial potential is $\sum \log(x_{i+1} - y_i) + \log(l_{i+1} - r_i)$, the current potential is $\sum \log(l_i - y_i) + \log(x_{i+1} - r_i)$.

According to the well-known inequality $x + y \geq 2\sqrt{xy}$, we have $\log(x + y) \geq (\log x + \log y)/2 + 1$. Thus, on the left $\log(x_{i+1} - y_i) \geq (\log(x_{i+1} - r_i) + \log(r_i - y_i))/2 + 1 > (\log(x_{i+1} - r_i) + \log(l_i - y_i))/2 + 1$. The difference between the left and right parts is $O(C) - O(\log n)$.

The overall complexity is $O(n \log^2 n)$.

Problem G. GymForces Show

Let's say we have two arrays – array x and array y, and beauty is equal to

$\sum_{i=1}^{n} \min(x_1, x_2, ..., x_i) \cdot \max(y_i, y_{i+1}, ..., y_n)$.

Let $pref_i = \min(x_1, x_2, ..., x_i)$ and $suf_j = \max(x_j, x_{j+1}, ..., x_n)$.

Then beauty is equal to

$\sum_{i=1}^{n} pref_i \cdot suf_i$.

Initially, $x_i = y_i = a_i$ for all i. When a request to remove an element at position j occurs, let's assign $x_i = 10^6 + 1$ and $y_i = 0$. Now beauty is calculated as

$\sum_{i=1}^{n} pref_i \cdot suf_i - \sum_{k} pref_k \cdot suf_k$ for those positions k that have been removed from the array.

Let's unfold all operations and instead of removing elements, we will add them. We will calculate beauty after all operations. Let's see how the arrays x and y change after adding elements. Suppose we add an element at position j with value val. Then the minimums on the prefixes $j, j+1, ..., n$ may change. Each of them may either not change or decrease. Notice also that the sequence of minimums on prefixes $pref$ is non-increasing, meaning it is monotonic. This means we can find the nearest position $r \geq j$ such that $pref_r < val$, and assign the elements $pref_j, pref_{j+1}, ..., pref_{r-1}$ the value val. Similar operations can be done with the array suf. When we add, the maximums on the prefixes $1, 2, ..., j$ may change. They either do not change, or some of them increase. We find the nearest position $l \leq j$ such that $suf_l > val$, and assign the elements $suf_{l+1}, suf_{l+2}, ..., suf_j$ the value val. We make such assignments because the prefix minimums were greater in this range (the suffix maximums were smaller), and after the new element val is added, the minimums/maximums become equal to val on these segments.

To perform assignments quickly, we will use a segment tree with mass operations. We need segment trees for the sum $pref_i$, the sum suf_j, the sum $pref_i \cdot suf_{i+1}$, as well as segment trees for deferred operations for $pref$ and suf. When assigning a value to a segment, we can recalculate $pref/suf$ and $pref_i \cdot sufi$. The complexity of the solution is $O(n \cdot \log(n))$.

Problem H. Hall 1666

With 7 queries of the form `aa`, `bb`, ..., `gg`, one can determine which numbers encode fives (V, L, D) and which encode units (I, X, C, M), since in the first case there are no corresponding numbers, while in the second case they exist. Then, with 3 queries, one can sort the fives using "bubble sort" (by asking queries of the form xy, if it exists, then $x > y$, otherwise $x < y$). And with 6 queries, one can also sort the units using "bubble sort" (by asking queries of the form xyy, since in the case of the query xy, there may be an undesirable side effect from the reverse notation; the interpretation of queries is similar to that of the fives). After that, all positions are guessed, and one can output 1666 (i.e., all Roman numerals one by one in descending order).

There are solutions that use fewer queries. The reader may find these solutions as an exercise.

Problem I. Interplanetary Business

First, note that when shifting the cutting line to the right, the area of the left part increases while the right

part decreases. That is, the ratio of the left part to the right part grows. We will apply binary search for the answer — we will search for the x-coordinate at which to cut. Now we need to learn how to calculate the area of the left part after the cut. For simplicity, we assume that the cut does not pass through any vertex of the polygon; otherwise, it can be moved by 10^{-9} in one direction, which will not significantly change the area. The cut-off part consists of one or more non-intersecting polygons formed by parts of the sides of the original (possibly truncated) polygon and vertical segments of the cutting line. The area of a polygon is calculated as half the absolute value of the sum over all sides of the oriented areas of triangles formed by the origin and the ends of the side. However, since the orientation of all of them is the same, we can first sum over all pieces and then take the absolute value. We will use this: we will separately traverse all the sides on the left, as well as the pieces of the sides that intersect the cut, and then we need to account for the segments of the cutting line itself. For this, we can keep track of which side we are on while going along the sides and remember the entry point from right to left, and when exiting back to the right, we account for this segment, so all pieces of the cutting line will be counted exactly once. At the same time, since we are using oriented areas, the pieces that should not be there will be subtracted. But we can also separately write down the entry and exit points and then walk through and sum the areas.

Problem J. Jackpot

Let us say we have somehow divided all the cards into three groups, and min_i, max_i are the minimum and maximum values in the i-th group, respectively.

The following conditions must be satisfied:

- $max_1 < min_2 + min_3$
- $max_2 < min_1 + min_3$
- $max_3 < min_1 + min_2$

Let's sort the cards in non-decreasing order of their values. We can say that the first card will go into the first group. Let $i > 1$ be the position of the leftmost card that will go into the second group, and $j > i$ be the position of the leftmost card that will go into the third group. By fixing such indices, we avoid double counting some options (for example, for the test case 111, we do not count the different groupings 123 and 321).

Notice that $min_1 + min_2 \le min_1 + min_3 \le min_2 + min_3$, since we have sorted the array. Let p be the leftmost position such that $a_p \ge min_1 + min_2$. Similarly, let q be the leftmost position such that $a_q \ge min_1 + min_3$. Let r be the leftmost position such that $a_r \ge min_2 + min_3$.

Then from position $j + 1$ to position $p - 1$, we can add cards to any group. From position p to position $q - 1$, we can only add to two groups. And from position q to position $r - 1$, we can only add to one group.

We need to multiply the answer by the corresponding precomputed powers of two and three. Also, from position $i + 1$ to $j - 1$, we have two options for adding cards to the group. If $r < n$, then the current answer is equal to 0.

The complexity is $O(n^2)$.

Problem K. King Primes

Since the sum of two odd numbers is even, the only way to obtain a prime number as the sum of the squares of two primes is when the second prime number is 2. Thus, King Primes are the prime numbers of the form $p^2 + 4$.

Therefore, we generate all prime numbers between 1 and 10^6, store them in a vector called primes, and for each prime number p, we check if $p^2 + 4$ is prime (since $[\sqrt{p^2 + 4}] \ge p$ for all $p \ge 3$, we can check this during the generation process by checking divisibility against all prime numbers less than p). If it is prime, we add it to the vector.

We can also precompute all King Primes (there will be just under 7000 of them), and then explicitly add them to the code. This solution also fits within the standard code size limits (64K) for ejudge.

After that, to answer the queries, we use binary search on the resulting vector. Note that when using the built-in functions `upper_bound` and `lower_bound`, for the upper boundary we should use the first, and for the lower boundary — the second, so that when subtracting, the situation where the boundaries themselves are the King Primes is correctly accounted for.

Problem L. Loose Connection

For solving the problem, many different encoding algorithms could be devised; here is the jury's algorithm: the first eight bytes of transmission are the original message, and the ninth byte consists of the first bits of each of these bytes. For example, `0F1AFF915733DE05` will turn into `0F1AFF915733DE0532`. Then, for decryption, we check how many bytes correspond to the last one; if 4 or more, it means the last one was untouched and everything can be restored from it. If it is no more than 3, it means the last one was inverted, and it needs to be reverted back, and the rest restored as well. This is true because if the last one was untouched, its bits will match those of all untouched bytes, while the bits of touched bytes will not match. Since there are a total of 8 bytes and inversions can be no more than 4, at least 4 must match. If the last one was touched, only the touched ones will match, but there can be no more than 3 of them.

Problem M. Modifying The Queue

The optimal strategy looks like this: remove the rightmost number that can be shot at, after which all numbers behind it become available, meaning that they can also be removed – we will do this from right to left. It turns out that we will remove a suffix from the last number that can be removed to the end. We will repeat this algorithm. In the end, it can be noticed that we will remove everything from the first available number to the end of the array, meaning that only the initial block of numbers cannot be removed. The complexity is $O(n)$.