Problem A. Arrange Jewels In Necklace

Let's note, that a backward column (x_i, y_i, z_i) and a forward column (u_j, v_j, w_j) can have the same central jewel, if $y_i = v_j$. Let's note, that due to all y_i are different, for each forward column there is at most one such backward column. So, let's find all pairs of backward and forward columns, that can have the same central jewel. There are at most m such pairs.

If a pair of columns (x_i, y_i, z_i) and (u_j, v_j, w_j) appear in necklace, then $a_p = x_i$, $c_p = w_j$, $a_{p+1} = u_j$, $c_{j+1} = z_i$, and a central jewel $b_p = y_i = v_j$. We can consider all pairs of (a_p, c_p) in necklace. Considered pair of backward and forward columns can translate (x_i, w_j) into (u_j, z_i). Necklace exists, if and only if, such graph of translations has a cycle. So, the task is to check if the directed graph has a cycle and can be solved in a linear time.

The time complexity is $O(n + m)$ if hash tables are used, or $O((n + m)(\log n + \log m))$ if maps are used.

Problem B. Bean Machine, Reinvented

Enumerate all leafs in dfs order. Now, any vertex corresponds to some segment of leafs. So we need to maintain the probabilities to end up in each leaf and just output some sums on segments. To process the changing query one needs to multiply the values on some segments by certain numbers. All of this queries can be easily done using the segment tree.

Problem C. Captain Thrawn

Let's view the towers as points on a complex plane with the real axis going along the street 0 and the imaginary axis going from south to north through the center of the hexagon. Then if ω is the cube root of 1 different from 1 (i.e. $\omega = -\frac{1}{2} + i\frac{\sqrt{3}}{2}$), then a tower at coordinates (x, y, z) has a complex coordinate $y + x\,\omega$.

Now if a, b, c are complex coordinates of 3 towers, then they form an equilateral triangle iff $\frac{a-b}{b-c} = \omega$. This is the same as $a - b = b\,\omega - c\,\omega$ or $a + c\,\omega + b\,\omega^2 = 0$, since $-1 - \omega = \omega^2$. Note that this does not account for non-zero side length requirement, so this condition still holds for $a = b = c$.

Let's create a polynomial $P(u, v)$ that is initially 0, and for each available tower $a + b\,\omega$ add $u^{a+l}v^{b+l}$ to P. Similarly, $Q(u, v)$ is the same as $P(u, v)$, but for tower coordinates multiplied by ω and $R(u, v)$ for tower coordinates multiplied by ω^2. Now in $a + c\,\omega + b\,\omega^2 = 0$, a corresponds to a term of P, $c\,\omega$ corresponds to a term of Q and $b\,\omega^2$ corresponds to a term of R, and their sum is 0 iff the product of those terms is $u^{3l}v^{3l}$. So we need to find $[u^{3l}v^{3l}]P \cdot Q \cdot R$. To get the answer we need to subtract n for degenerate triangles and divide by 3, since each triangle is counted 3 times for 3 different rotations.

We can find the product of P and Q with FFT or NTT in $\mathcal{O}(l^2 \log l)$ time and then find one coefficient of its product with R in $\mathcal{O}(l^2)$ time for a total of $\mathcal{O}(l^2 \log l)$ time. Note that if we try to calculate all of $P \cdot Q \cdot R$ with FFT then the second multiplication will be 4 times larger than the first, so the solution will run 5 times slower and that was not expected to pass.

Problem D. Divisibility By Factorial

In the case of an odd n, $n!!$ is odd. The only odd factorial is one, meaning in this case $k = 1$. In the case of an even n, $n!! = \prod_{i=1}^{n/2} 2 \cdot i = 2^{(n/2)} \cdot (n/2)!$. If $n/2 + 1$ is not a power of two, then $k = n/2$ (all divisors of the first factor are powers of two). If $n/2 + 1$ is a power of two, it can be shown (for example, by induction) that $2^{(n/2)} \geq n/2 + 1$ (for $n = 2$, the statement is obvious; as we move to $n = n + 2$, the left side doubles while the right side only increases by 1). Therefore, in this case, the factor $(n/2 + 1)$ can always be extracted, and $(n/2 + 2)$ is odd in this case and cannot be a divisor of a power of two, meaning the answer will be $k = n/2 + 1$.

Problem E. Expected Symmetry

Firstly, let us reconsider the probabilistic model. First time we look at each element we see its given value, and for each next look up we can see it as 0 or 1 equiprobably. This allows dp_i as the expected number of steps to the stop if the longest prefix seen so far is $s_1 ... s_{2i}$. In this definition dp_0 is the answer to the problem.

Let's define anti-palindrome as binary string s, such that its reverse equals to its inverse (bitwise negation). Now we describe the calculation of the dp. The formula uses that each pair of elements one of which was seen before, makes symmetry point with probability of $\frac{1}{2}$:

$$dp_i = 1 + \frac{1}{n} \cdot [\sum_{k=1}^{i} \frac{dp_i}{2^k} + \sum_{k=i+1}^{2i} \frac{dp_k}{2^k} + \sum_{k=2i+1}^{n} \frac{dp_k}{2^i} \cdot I(s_{2i+1} \dots s_k s_{k+1} \dots s_{2k-2i} \text{ is anti-palindrome})]$$

$$\frac{(n-1)2^i + 1}{2^i} dp_i = n + \sum_{k=i+1}^{2i} \frac{dp_k}{2^k} + \sum_{k=2i+1}^{n} \frac{dp_k}{2^i} \cdot I(s_{2i+1} \dots s_k s_{k+1} \dots s_{2k-2i} \text{ is anti-palindrome})$$

Iterate k from n to 1, and add its impact for $i < k$. Impact in case $i < k \le 2i$ added with basic prefix sum technique. Range for i for impact from second summation should be bounded by the radius of even length anti-palindrome with center in $s_k s_{k+1}$. To count anti-palindrome radii we can implement anti-manacher algorithm.

Another crucial implementation detail is that we should calculate modulo-inverses for numbers $\frac{(n-1)2^i+1}{2^i}$. Since this numbers are known in advance, we can calculate their inverses offline in linear time. The technique is the same as calculating inverses for $1, \dots, n$ with inverse factorials, we just calculate prefix products and inverse prefix products.

Problem F. Fast Value Changes

Note that until all prices are the same, the price of the lowest priced asset increases and the price of the highest one decreases. So the difference between them is decreasing at the speed of 2 until it becomes 0. Therefore the answer is $\frac{\max a_i - \min a_i}{2}$.

Problem G. Gravity Zero Experiments

We have the set of n segments $A_i B_i$.

How to describe the set of gravity directions $\overrightarrow{g}$, such that the ball can go from segment $A_i B_i$ to $A_j B_j$? Let's note, that by definition, this set is the Minkowski subtraction $A_j B_j - A_i B_i$ (and any vector from this set can be multiplied by positive constant, because we care only about the direction). The set of points $A_j B_j - A_i B_i$ is the parallelogram with vertices $A_j - A_i$, $B_j - A_i$, $B_j - B_i$, $A_j - B_i$. The set of gravity directions is enclosed by 4 planes:

- The plane containing $(0,0,0)$ and $A_j - A_i$, $B_j - A_i$. This plane's normal vector is $(A_j - A_i) \times (B_j - A_i)$.
- The plane containing $(0,0,0)$ and $B_j - A_i$, $B_j - B_i$. This plane's normal vector is $(B_j - A_i) \times (B_j - B_i)$.
- The plane containing $(0,0,0)$ and $B_j - B_i$, $A_j - B_i$. This plane's normal vector is $(B_j - B_i) \times (A_j - B_i)$.
- The plane containing $(0,0,0)$ and $A_j - B_i$, $A_j - A_i$. This plane's normal vector is $(A_j - B_i) \times (A_j - A_i)$.

How to check that the gravity vector $\overrightarrow{g}$ belongs to this region: for each of these 4 planes we can check that the sign of the cross product of $\overrightarrow{g}$ and a normal vector is the same as the sign of the cross product of some other parallelogram vertex and this normal vector.

Suppose, that the gravity direction $\overrightarrow{g}$ is fixed and we want to update the answer will all cyclic sets with this gravity direction. We can build the directed graph with n vertices with edges from one segment to

another meaning that it is possible to go from one segment to another with given gravity direction $\overrightarrow{g}$. After that, the problem can be solved by calculating the Strongly Connected Components of this graph in $O(n^2)$. Each subset of any component should be updated as cyclic.

How to find all interesting gravity directions $\overrightarrow{g}$ that we should check to find the answer? Let's note, that we have $\frac{n(n-1)}{2}$ parallelograms having $2n(n-1)$ planes separating the interesting directions regions. It is enough to check all pairwise intersections of these planes as an interesting directions. So, we can find all pairwise intersections, their number is $\leq 2n^4$ and check each of them in $O(n^2)$ time. After we found all SCC for all interesting directions, we should mark the answer for them as 1 and propagate the answer 1 to all their subsets. The last part can be done in $O(n2^n)$.

The time complexity is $O(n^6 + n2^n)$.

Implementation details:

- When finding the interesting gravity directions, we do not care about the sign of each direction, because we do not need to check $-\overrightarrow{g}$, if we already checked $\overrightarrow{g}$. So, we can just intersect planes without thinking about the half-planes.
- Due to all interesting planes contain the point $(0, 0, 0)$, the direction of the intersection of two planes with normal vectors $\overrightarrow{n_1}$ and $\overrightarrow{n_2}$ is $\overrightarrow{n_1} \times \overrightarrow{n_2}$. So we just need to check all pairwise inner-products of all $2n(n-1)$ normal vectors.
- The problem can be solved with integers. Let's define $C = 10^6$ as the upper bound of the coordinates (by modulo). Normal vectors will have coordinates $O(C^2)$. So, all interesting gravity directions will have coordinates $O(C^4)$. When we check the gravity direction, we use the cross products between normal vectors and gravity direction, so we will calculate $O(C^6)$ values. So, by using `__int128` we can calculate everything in integers.
- Actually, the implementation is short and simple enough. You need to implement only subtraction, inner and cross product for 3D geometry. After that, you construct all normal vectors, all interesting gravity directions and check each of them.

Problem H. Homework Of Xenia

First colour the graph in 5 colours. This can be done naively (in the same way we prove the 5 colour theorem). Let us now fix some integer r and place all vertices of colour $c \in \{1, 2, 3, 4, 5\}$ on a circumference $x_{2c-1}^2 + x_{2c}^2 = r^2$. Then the distance between any two vertices of different colours will be exactly $\sqrt{2}r$. It remains to find r such that there are at least 1000 integer solutions of the equation $x^2 + y^2 = r^2$. Turns out that $r = 5^3 \cdot 13^2 \cdot 17^2 \cdot 29 \cdot 37 \cdot 41$ suffices for this.

Problem I. Interactive Array Modifications

Firstly, let's ask an array with all zeros. As a result, we will find the position s of the minimal element ($a_s = 0$).

After that, to find all other values, let's make parallel binary searches. For each position i we will have some segment $[l_i, r_i]$, such that $l_i \leq a_i < r_i$. Initially for all $i \neq s$ $r_i = 10^9$ and for all $i < s$ $l_i = 1$, for all $i > s$ $l_i = 0$.

How to reduce lengths of candidate segments? Let's find $m_i = \lfloor \frac{l_i + r_i}{2} \rfloor$. For each i we want to check if $m_i \leq a_i$ or not. Let's ask the following array b:

- $b_i = 10^9 - l_i - m_i + 1$ for all $i < s$
- $b_s = 10^9$
- $b_i = 10^9 - l_i - m_i$ for all $i > s$

As a result, all positions that have $a_i + b_i > a_s + b_s$ have $m_i \le a_i$, all other positions have $m_i > a_i$.

The number of queries to make binary searches is 30 and we had one more initial query, so we solved the problem with 31 queries.

Problem J. JQU

Consider the following dynamic programming, $dp_{i,j,w} = \{a, x\}$, where a is the maximal string one can obtain from $s[i:]$ if we have j m-functions opened on the prefix and did/didn't open a new function right before $s[i]$ and x is the number of ways in which we can obtain a. This dp can easily be calculated in $O(n^3)$. Exercise: optimize this solution to $O(n^2 \log n)$.

Problem K. Know Your Powers

From now on denote $2 \cdot 10^9$ by C.

Solution 1. Note that, knowing the factorization of a number n, one can find $\sigma_1(n)$. Indeed, if $n = p_1^{\alpha_1} \cdots p_k^{\alpha_k}$, then $\sigma_1(n) = \prod_i (1 + p_i + \ldots + p_i^{\alpha_i})$. However, going over all $\binom{n}{2}$ sums and factoring them naively is too slow.

On the other hand, a "naive" factoring algorithm can be described as follows: for all primes p below $\sqrt{C}$, if p divides n, then divide n by p while we can, then update the current value of $\sigma_1(n)$. In the end, if $n > 1$ then n is a prime number greater than $\sqrt{C}$, and we should update the answer accordingly.

Most of the runtime for this algorithm comes from the unsuccessful divisibility checks, so the main idea of the solution is to, for each small prime p, handle all sums together and only do the division when the sum is divisible by p. So the pseudocode would be like this:

```
cur_values = [[a[i] + a[j] for j=0..n] for i=0..n]
cur_answers = [[1 for j=0..n] for i=0..n]
for p in primes below sqrt(C):
    for i in 0..n:
        place i in the group a[i] mod p
    for each nonempty group r_1:
        let r_2 = (p - r_1) mod p
        if r_1 = r_2:
            for (i, j) in group r_1:
                update cur_values[i][j] and answers[i][j]
        else if group r_2 is nonempty too:
            for i in group r_1 and j in group r_2:
                update cur_values[i][j] and answers[i][j]
for i = 0..n:
    for j in i+1..n:
        if cur_values[i][j] > 1:
            cur_answers[i][j] *= cur_values[i][j] + 1
```

The time complexity of this solution is $O(\sqrt{C} + \pi(\sqrt{C}) \cdot n + n^2 \log C)$, where $\pi(n)$ is the number of primes not exceeding n (which is $\Theta(n/\log n)$). Of course, this requires fast implementation (at least without extra logarithmic lookup) of "placing an index into a group" and "going over all nonempty groups from this iteration", which can be achieved by using vectors carefully. Also, finding primes below $\sqrt{C}$ is a well-known task which can be done using Eratosthenes sieve, which, in order, can be implemented in $\sqrt{C}$ or even $\sqrt{C} \log C$ in this problem.

Solution 2. Another way of calculating $\sigma_1(n)$, which doesn't involve factoring, would be simply by going over all numbers d up to $\sqrt{n}$, adding d if it divides n, adding n/d if d divides n and $d^2 < n$. Again, we can do this simultaneously for all numbers.

In other words, sort all numbers. Then, for each d below $\sqrt{C}$ (not necessarily prime), we split all a_i into groups again, based on their remainders modulo d. After that, for all pairs of nonempty groups with

opposite remainders, we can use two pointers to find the number c of pairs of numbers from those groups whose sum is at least d^2, and add $d \cdot c$ to the answer. Then, using a similar two pointers technique, one could find the sum s of all pairs of numbers from those groups whose sum is *strictly greater* than d^2, and add s/d to the answer. The case where the group remainder is opposite to itself is analogous to the former.

The time complexity of this solution is $O(n\sqrt{C})$.

General note. In both solutions it's important to keep track of how many expensive operations the solution does, and a too careless implementation of an otherwise correct solution may violate the time limit. Expensive operations include, for example, divisions, modular operations, and pushes to vectors.

Problem L. Latin Words

Lets say that the letter (the substring) is empty if it does not contain Roman numerals.

Consider a query for the value of a substring of length 2. If the answer is 0, then the substring is empty. If the answer is a two-digit Roman numeral, then we know both digits (the first digit must be greater than the second: in the inverted representation, the maximum is a one-digit number from the second digit: for `IV`, the maximum is `V`). If the answer is a one-digit Roman numeral a, we ask for the first digit. If it is 0, the first letter is empty and the second is a; if it is a digit less than a, then the second letter is a. If it is a, then the second letter is empty (otherwise, we would have a two-digit number in the answer).

Thus, we first make 12 queries: $(0, 1)$, $(2, 3)$, ..., $(22, 23)$. If among them there are 7 non-zero answers, then we learn the position of each digit in 7 queries. In total, that makes 19 queries.

If there are fewer non-zero answers, we will learn about all digits except those in the block $(24, 25)$ after spending 6 queries. We have used 18 queries. If we have learned all 7 digits, the problem is solved. If we are missing one or two digits, we query the substring $(24, 24)$, after which the remaining digits can be determined uniquely. In total, it is still the same 19 queries.

When building the answer, put the Roman numeral letters onto the places we found, and then fill the gaps with the remaining 19 letters in alphabetical order.

Problem M. Mobile Game Development Lab

If a string can be obtained in the wayr specified in the problem, then there are 3 possible cases:

- a does not end with a nine. Then the string contains all the digits of the number a, except for the last one, twice, as well as the last digit and the digit that is one greater — once each. The length of the string is even.
- a consists only of nines. Then the string contains one one and an equal number of zeros and nines. The length of the string is odd.
- a ends with q nines, and there is at least one digit in a that is not equal to 9. Then $a+1$ ends with q zeros, and the non-zero digit that precedes the i nines is replaced by the digit that is one greater. The length of the string is even.

Let the number of digits i in the string s be denoted as $freq(i)$. We will count all $freq(i)$ while reading the input data and will work only with them afterwards (since the order of the digits can be arbitrary and does not carry information).

If $|s|$ is odd, we check that $freq(0) = freq(9) = (|s| - 1)/2$, $freq[1] = 1$. If this is true, the answer is 1. If false, the answer is 0.

In the case of even $|s|$, we first determine the maximum q, that is, we find $\min(freq(0), freq(9))$. We subtract q from $freq(0)$ and $freq(9)$, after which we find ourselves in the situation of the first case. We check that all $freq(i)$ are even, except for two $freq(j)$ and $freq(j+1)$, which are odd. If this statement is false, the answer is 0. If it is true, then we have found the last digit of the original number, j. We

decrease $freq(j)$ and $freq(j+1)$ by one, resulting in all $freq(i)$ being even. We divide $freq(i)$ by two and obtain the problem of "how many integers without leading zeros can be formed from the given set of digits". If $freq(i) = 0$ for all $i \neq 0$ and $freq(0) \neq 0$, then the answer is 0 (we cant put the leading zeroes).

Let the answer to this problem be denoted as $f(freq)$, where $freq$ is the array of all $freq[i]$.

Then if $freq(0) = 0$, $f(freq)$ equals the number of distinct permutations of these digits, that is,

$$\frac{(\sum_{i=0}^{9} freq(i))!}{\prod_{i=0}^{9} freq(i)!}$$

If $freq(0) > 0$, then from the previous number we need to subtract the number of permutations where the first digit is 0, that is,

$$\frac{(freq(0) - 1 + \sum_{i=1}^{9} freq(i))!}{\prod_{i=1}^{9} freq(i)! \cdot (freq(0) - 1)}$$

After calculating $f(freq)$, in the case where $q \leq 1$, we have obtained the answer.

However, if $q \geq 2$, then we have additional options: each pair of zeros and nines can be removed from the end to the "common" part, that is, for example, 1429914300 can be represented as (14)299, (14)300, as well as (1490)2, (1490)3. Here, the rearranged blocks of digits are indicated in parentheses.

Thus, in this case, in a loop from i from 1 to $q/2$, we increase the values of $freq(0)$ and $freq(1)$ by one and then add the value of f from the modified $freq$ to the answer.

For quick computation of factorials and division by factorials, it is recommended to precompute a table of factorials and a table of numbers that are the inverses of factorials given modulo.