Problem A. Bear Girls

Let's assume that after the k-th date, John decides to choose the current girl and stop dating, regardless of who she is. In this case, it is equally likely that the girl he selects is the most beautiful among all the candidates, the second most beautiful, and so on, up to the k-th most beautiful. If we know the expected beauty of the girl, given that she is the i-th most beautiful among the randomly chosen k girls, denote this by $f(k, i)$, then the expected beauty will be:

$$\text{Expected beauty} = \frac{1}{k}\sum_{i=1}^{k} f(k, i)$$

Let ans[k] be the dynamic programming state representing the expected value of $B - C$ if John starts choosing from the k-th date onwards, having skipped the first $k - 1$ dates.

Now, if John decides to think further and not stop immediately, he evaluates the current girl as the i-th most beautiful among the k girls $f(k, i)$ (though he doesn't know the expected value, because he doesn't know their exact beauty ranking, he only knows how she compares to others he's seen so far). If he does not stop and accept the current girl, he will have to pay the cost c, and move on to the next girl. Hence, the dynamic programming state can be updated as follows:

$$\text{ans}[k] = \frac{1}{k}\sum_{i=1}^{k} \max(\text{ans}[k+1] - \text{cost}, f(k, i))$$

What is hard to calculate is $f(k, r)$ denoting the expected value of the r-th biggest number among k randomly chosen ones. It turns out to be simple to calculate because $f(k, r) = p \cdot f(k+1, r+1) + (1-p) \cdot f(k+1, r)$ for $p = \frac{r}{k+1}$. Don't be misled by the words "simple to calculate". It took me literally weeks to solve this problem and I don't say that it is easy to find the formula above.

Looks like a binomial coefficient, right? And this is how we can prove its correctness. When we have k numbers and the r-th biggest one is marked, we can still add other numbers one by one. The new number will be greater with probability $p = \frac{r}{k+1}$.

Problem B. Bichrome Sky

Consider a way to calculate the area of the intersection of two convex polygons, having n and m sides, respectively.

$\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} \text{area}(\text{under}(p_i, p_{i+1}) \cap \text{under}(q_i, q_{i+1})) \times \text{left}(p_i, p_{i+1}) \times \text{left}(q_i, q_{i+1})$, where $\text{under}(a, b)$ is a trapezoid under the line segment (a, b), and $\text{left}(a, b)$ equals 1 if a is to the left of b, and -1 otherwise.

Let's then consider all ways to choose two sides of two polygons: of the red one and the blue one. And calculate the probability when these two line segments are the sides of the polygons. And multiply it by the signed area of their intersection as above.

The line segment is the side of a convex polygon if all the points lying on the right halfplane are of the other color. So iterate over the fifth point; for each point, calculate the probability that it doesn't make any of the two line segments bad. Multiply over all points; you have the probability. The solution is $O(n^5)$.

Problem C. Zandar's new game

We do step by step, after first from i to j, then from j to k, then from k to l. After each step we end by number of ways so that current index is p, so say it's x_p. Initially $x_p = 1$ for all p.

So we apply "and", then apply "or" then apply "and".

How to apply "and": `x[j] += x[i]` if $F(a_i \wedge a_j) = b_j$, where $\wedge$ is binary "and".

How to apply it faster: go from left to right, maintain $f(l, m, c)$ — number of ways to end by in a_i such that it's lowest 8 bits are l, and the number of bits among the 8 highest bits in a bitwise "and" of $A[i]$ and some arbitrary number x is c, and x's highest 8 bits are m. To calculate x_j, we take for every l, and add $f(l, \text{high}(a_i), B[i] - F(\text{low}(a_i) \wedge l))$. And to update f, just iterate over all 256 different m, and update $f(\text{low}(a_i), m, F(\text{high}(a_i) \wedge m))$ with old x_j.

Similar formulae are for "or" application, just use c instead of b and binary "or" instead of "and".

Problem D. Deterministic Random Exploration

The graph of this structure consists of several components each of which can be represented as a rooted tree with one more edge (denote this edge as *extra*). From another point of view, each component has exactly one cycle.

The task can be separated into four parts. First, we group all components with a cycle of equal length. Then for each group, we will calculate the probability distribution of making $0 \dots k \times m$ steps in a compressed form. Then we will process all components of the group in the decreasing order of their sizes. For each component, we will calculate all ways with *extra* edge and without it independently.

Separating into groups

It's a well-known fact that if the sum of positive integers is n, then there are at most $O(\sqrt{n})$ different values among them. So there will be at most $O(\sqrt{n})$ different groups. Consider the group with cycle length c. Let s be the maximum size of the component in this group. Then we can reduce the probability vector $p_0 \dots p_m$ to the size $s + c$ as making $s + c + k$ steps ($t > 0$) will lead to the same vertex in the graph as making $s + t$ steps. After this step, we don't depend on m anymore. As there are $O(\sqrt{n})$ groups, the total reduction time will be $O(m \times sqrtn)$.

Calculating the distribution for each component

The distribution can be written in polynomial form $P(x) = \sum_{i=0}^{c+s} p_i x^i$. Our goal is to calculate $P^k(x)$ to find the distribution after k steps. This can be done with fast polynomial multiplication and binary exponentation. Note that after each multiplication the size of polynomial must be reduced to $c + s$ as described above or we get the polynomial of a degree $k \times (c + s)$ which is too many. This part will work in $O(n \log n \log k)$ time in total.

Now we can process all components of the group in the order of decreasing size. Before processing the component of size t, the polynomial must be reduced to the degree of $c+t$. After this moment, the solution for each component of size t will be run with already precalculated distributions array of length no more than $2t$.

Paths through the extra edge Consider the polynomial $Q(x) = \sum_{i=0}^{t} h_{t-i} x^i$, where h_i is the number of vertices with depth i from the root. The product $P(x) \cdot Q(x)$ contains all the needed information about this type of paths. Indeed, the coefficient near x^{t+d} is $\sum h_i p_{i+d}$. Here d means the number of steps from the root to the final vertex. Considering all coefficients $d > 0$ and adding the respective probability to the answer for respective vertices in the graph, we will allow for all paths through the extra edge.

Paths in the tree

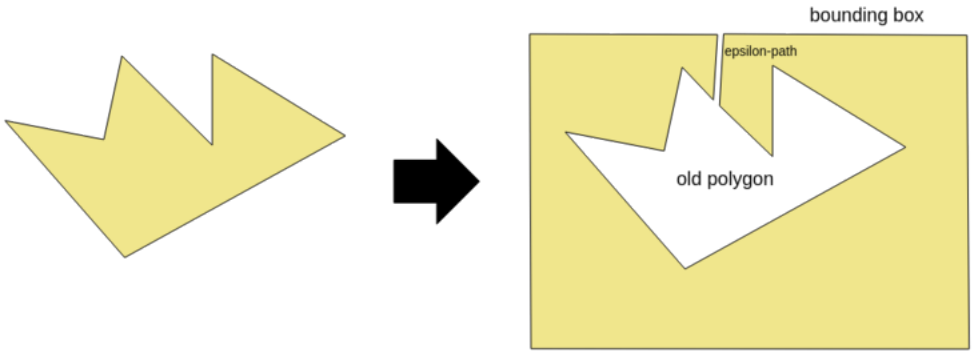
This is the hardest part of the solution. For each vertex u with depth b from the root we have to calculate $\sum p_i d_{i-b}$, where d_i is the number of vertices with depth i in the subtree with root u. Let's write all vertices of the tree in the array in DFS traverse order and separate this array into $O(\sqrt{\frac{n}{\log n}})$ blocks (in practice, the block size is about 4000). For each block i, we will write the same polynomial $Q_i(x)$ as in the previous paragraph but considering only the vertices from this block. Then we will calculate all polynomial products $P(x) \cdot Q_i(x)$. This will take $O(n\sqrt{n \log n})$ time in total. For each vertex u, its subtree will be a consecutive subsequence of the array. Each such subsequence can be split into $O(\sqrt{\frac{n}{\log n}})$ blocks and extra $O(\sqrt{n \log n})$ elements. Each element can be considered in $O(1)$ time and each block also can be considered in $O(1)$ time in the similar way described in the previous paragraph.

Problem E. Tensioned Rope

General idea

If the rope was inside the polygon rather than outside, the task would be much easier. In this formulation, the problem is actually about finding the **shortest** path between vertices. From any starting position, the elastic rope will shrink to the shortest path as the inside of the polygon is a simply connected space.

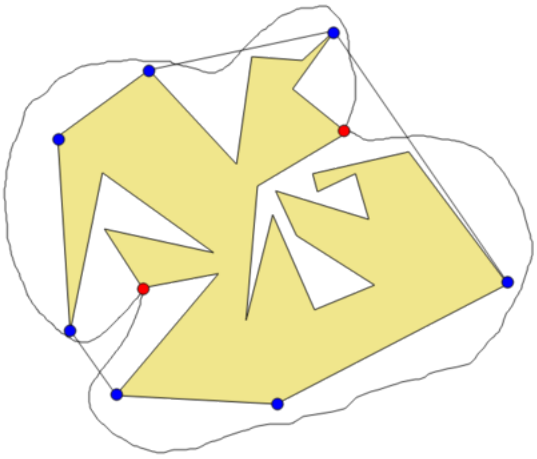
Unfortunately, in our task, the maximum path is not necessarily the shortest path. But the following modification can be applied:



If after this modification the shortest path does not touch the epsilon-path and bounding box, this is the valid rope state. Actually, there are at most two possible states. The first one occurs when the *path root* isn't separated from the *bounding box* by the shortest path between the ends of the rope. In this case, the rope state will be equal to this shortest path. The second one occurs when the *path root* is separated (this case is not always possible).

Let's pick the vertices which belong to the convex hull of the polygon. The polygon border is separated by these vertices. Denote the section between such vertices as a "hole". Now, there are two cases: if the ends of the rope belong to different holes or to the same one.

Different holes

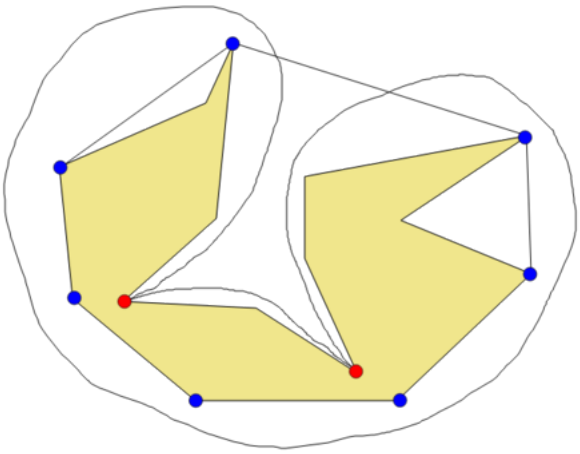


In this case, the rope line will be split into three parts:

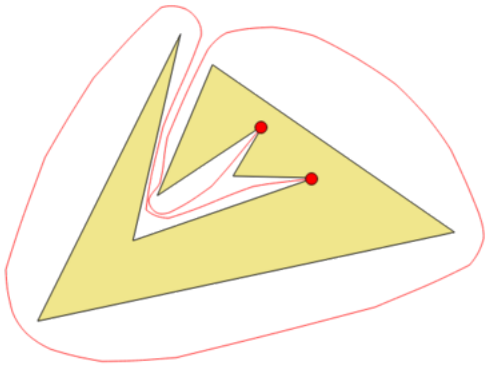
1. The shortest path between the beginning of the rope and one of the ends of the hole.

2. The part of the convex hull between the ends of the holes.
3. The shortest path between the end of the hole and the end of the rope.

Single hole



In this case, the first possible state of the rope is just the shortest path. The second possible state is similarly split into three parts as in the previous paragraph. However, in some cases, this state can be invalid because the rope will have self-touches or self-intersections:



Implementation notes

The implementation consists of five parts:

1. For each diagonal in the polygon, check whether it is outside the polygon.
2. Find the shortest paths from the ends of the rope to each vertex (Dijkstra's algorithm).
3. Build the convex hull of the polygon.
4. Build two possible rope states.
5. Check if these states are valid (i.e., the rope has no self-touches and self-intersections).

Each part, except for the first, can be done in $O(n^2)$ time. The first part can be done either in $O(n^3)$ or $O(n^2 \log n)$ time.

Problem F. Aerobics

The key to this problem lies in realizing that there really is a lot of space on the mat to take advantage of, and a number of different approaches will work.

For the small input, putting the circles along one of the longer edges, and — if it runs out — along the opposite edge will work. The precise analysis of why they fit is somewhat tedious, so we will skip it in favor of describing two solutions that can also deal with the large input.

A randomized solution

The large input is a more interesting case. We will describe two solutions that allow one to solve this problem. The first one will be randomized.

We will order the circles by decreasing radius. We will then take the circles one by one, and for each circle try to place it on the mat at a random point (that is, put the center of the circle at a random point on the mat). We then check whether it collides with any circle we have placed previously (by directly checking all of them). If it does collide, we try a different random point and repeat until we find one that works. When we succeed in placing all the circles, we're done.

Of course, if we manage to place all the circles, we have found a correct solution. The tricky part is why we will always find a good place to put the new circle in a reasonable time. To see this, let's consider the set of "bad points" — the points where we cannot put the center of a new circle because it would cause a collision. If we are now placing circle j with radius r_j, then it will collide with a previously placed circle i if and only if the distance from the new center to the center of circle i is less than $r_i + r_j$. This means that the "bad points" are simply a set of circles with radii $r_i + r_j$.

What is the total area of the set of bad points? Well, since the set is a group of circles, the area is at most the sum of the areas of the circles. (It can be less because the circles can overlap, but it cannot be more). As we are placing the circles ordered by decreasing radius, we know $r_j \leq r_i$, so the area of the i-th "bad" circle is at most $\pi \times (2r_i)^2 = 4\pi r_i^2$. Here is where we will use that the mat is so large — we see that the total bad area is always at most 80 percent of the mat. Therefore, we have at least a 1 in 5 chance of choosing a good center on every attempt. In particular, it is always possible to find a good center, and it will take us only a few tries.

For each attempt, we have to make $O(N)$ simple checks, and we expect to make at most $5N$ attempts, so the expected time complexity of this algorithm is $O(N^2)$ — easily fast enough.

A deterministic solution

As usual, if we are willing to deal with a bit more complexity in the code, we can get rid of randomness in the solution, taking advantage of all the extra space we have in a different way. One sample way to do this follows.

To each circle of radius R we attach a $4R \times 2R$ rectangle as illustrated below:

The top edge of the rectangle passes through the center of the circle. The bottom edge, the left edge, and the right edge are all at a distance of $2R$ from the center of the circle.

We now place circles one at a time, starting from the ones with the larger radius. We always place each circle in the topmost (and if we have a choice, leftmost) point not covered by any of the rectangles we have already drawn.

An argument similar to the one used in the randomized solution proves that if we place a circle like that, it does not collide with any of the previously placed circles. As we place each point in the topmost available point, the centers of the previously placed circles are necessarily above the last one placed, and so if our new circle would collide with one of the previous ones, it would have to be in the rectangle we associated with it.

Now notice that the areas of all the rectangles we place is the sum of $8R^2 = \frac{8}{\pi}$ times the total area of the circles — which is easily less than the area of the mat. This means we always can place the next circle within the mat.

This solution is somewhat more tricky to code than the previous one (because we have to find the topmost free point, instead of just choosing a random one), but still easier than if we tried to actually stack circles (instead of replacing them by rectangles).

Problem G. Persistent priority queue

Use leftist heap for example.

Problem H. Public Transit

Fix the first teleporter at A. Now perform a Depth-First Search (DFS) starting from A. Suppose we are currently visiting node B. Each node N on the path from A to B has a precomputed 'hanging value', which is the length of the longest path starting from N that does not get closer to A or B. Number the nodes on the path from 0 (which is A) to $S-1$ (which is B), where the path contains S nodes. When moving between nodes at positions p_1 and p_2 on the path ($p_2 > p_1$), we can take either a direct route (length $p_2 - p_1$) or an indirect route (length $(S-1) - p_2 + p_1$). Note that the length of the direct route is fixed, while the indirect route increases by 1 minute with each step of the DFS.

In the DFS transition, we add a new node to the path. We will process this node to obtain an integer LIM, meaning that the new node imposes the constraint that we cannot continue the DFS in a direction away from A and B for more than LIM steps.

To process the node, suppose the node has a hanging value v_2 and position p_2. We want to select a node on the path with value v_1 and position p_1 such that:

- The direct path is infeasible: $p_2 - p_1 + v_1 + v_2 > X$. Rearranging, we want $v_1 - p_1 > X - p_2 - v_2$.
- The limiting factor $\text{LIM} = X - ((S-1) - p_2 + p_1 + v_1 + v_2)$ is minimal. Rearranging, we want to maximize $v_1 + p_1$.

By converting candidate nodes (v_1, p_1) to diagonal coordinates $(v_1 - p_1, v_1 + p_1)$, the problem reduces to a dynamic Range Minimum Query (RMQ) problem, which can be solved using a persistent segment tree. The complexity is $O(N \log^2 N)$ for each update and query, where N is the number of nodes.

Problem I. Desert

If we have a graph of dependencies, with edges uv of two types: v is strictly greater than u, and v is greater or equal to u, then how we assign integers to each vertex. Make a graph with these edges, assign a weight $w(uv) = 1$ for strictly greater, and $w(uv) = 0$ for greater or equal. Make a topological sorting. For each vertex then find the minimum value as: $f_v = \max\left\{1, \min_{uv \in E}\{f_u + w(uv)\}\right\}$. If any f_v is greater than maximum allowed 10^9, then it's impossible. Also if any value already given, assign it to f_v, if f_v calculated is greater than the value already given, then it's impossible as well.

How to make a graph?

To do that we will make a segment tree structure, all edges are of weight 0 directed from nodes to it's children in a segment tree. For each measurement, we create an additional vertex m, from each of the vertices that are lower we add an edge to m of weight 1, and from m we need to add to each subsegment, that were created by removing the lower vertices. To do that we will divide each subsegment into $O(\log n)$ segment tree nodes that cover the subsegment, and make an edge with weight 0 from m to them.

Problem J. Array Transformer

All operations except 2 can be done with simple segment tree. Let's also support the operation "find maximum of a_i for $l \le i \le r$"in this tree. Now to support the operation of second type, let's just bruteforce the segment tree. If the current segment is inside $[l \dots r]$ and $\left\lfloor \frac{m}{d} \right\rfloor - m = \left\lfloor \frac{M}{d} \right\rfloor - M$ (where m is the minimum on this segment and M is the maximum) then let's just add $\left\lfloor \frac{m}{d} \right\rfloor - m$ to all elements on this segment and stop going deeper in the tree. The described algorithm is correct and has time complexity $\mathcal{O}(q \log n \log C)$.

Proof

For fixed $d > 1$ define $f(x) = \left\lfloor \frac{x}{d} \right\rfloor - m$. If $f(M) = f(m)$ $(m \le M)$ then $m = M$ or $m + 1 = M$. That proves the correctness of addition $f(m) - m$ on the segment - there are no numbers on this segment except for m and M.

Each operation of type 2 separates the segment $[l \dots r]$ into several subsegments and does the addition of different constant on each. Let's show that there will be no more than $n \log C$(where $C = 10^9 + 10^4 \times q$) segments at all. Consider the situation when position i was separated from $i + 1$ in some query. After the query, the absolute difference $s = |a_i - a_{i+1}|$ will be at most $\left\lceil \frac{s}{2} \right\rceil$. So there will be no more than $\log C$ steps until $s \le 1$. And if $s = 1$ before and after the operation then positions i and $i + 1$ can actually be merged in one segment.

Problem K. Semi Perfect Powers

A first simplification

As usual in this kind of problem where we have to count some subset of numbers within an interval $[L, R]$, we reduce the number of parameters (L and R) from two to just one. Let $\texttt{precount}(X)$ be the number of semi-perfect powers that lie in $[1, X]$. (For $X = 0$, it returns 0.) Then our answer is just $\texttt{precount}(R) - \texttt{precount}(L - 1)$. Like L and R, X can be at most 8×10^{16}.

Only squares and cubes

Recall that a semi-perfect power is defined in the problem to be a number of the form ab^c where a, b, c are positive integers, $b > 1$, $c > 1$ and $a < b$.

It turns out that we can always assume c is either 2 or 3. That is, all semi-perfect powers can be written with $c = 2$ or $c = 3$. To prove this, we consider the cases when the exponent c is even and odd separately. If $k > 1$, then $ab^{2k} = a(b^k)^2$. So it is a semi-perfect power with the exponent $c = 2$. Again, if $k > 1$, $ab^{2k+1} = (ab) \cdot (b^k)^2$. Since $a < b$, and $k > 1$, we have $ab < b^2 \le b^k$. Thus, semi-perfect powers with the exponent 4 or greater can be written as semi-perfect powers with the exponent 2.

From the above, it follows that in computing $\texttt{precount}(X)$ it is enough to consider numbers that can be written in the form ab^2 or ab^3. (with $a \ge 1$, $b > 1$). Let's call them semi-perfect squares and semi-perfect cubes, respectively.

Counting semi-perfect squares

A useful fact is that every positive integer n can be written uniquely in the form $n = ab^2$, where a is a square-free number and b is positive. A square-free number is a positive integer which is not divisible by a perfect square greater than 1. Equivalently, a square-free number is one whose unique prime factorization contains each prime with multiplicity exactly 1.

To prove this fact, we can work with the unique prime factorization of n. If p^{2k+1} occurs in the prime

factorization, then we must keep p^k in b, and p in a. If otherwise, p^{2k} occurs, then we must keep p^{2k} in b, and p won't occur in a. In any case, we have exactly one choice for a and b. Let us call this representation of n as ab^2 the square-free representation of n.

Next, we see that if n is a semi-perfect square, then we must have $a < b$ in the square-free representation of $n = ab^2$. That is, the square-free representation always acts as a witness for proving that n is a semi-perfect power. To prove this, suppose that n is a semi-perfect square. Then n can be written as $a'b'^2$ with $a' < b'$. If $n = ab^2$ is the square-free representation of n, we will show that $a < b$, too. Since $a'b'^2 = ab^2$, it is enough to show that $b' \leq b$. For then, $a \leq a' < b' \leq b$, so we will have $a < b$. Let p be any prime dividing n. Then an even power of p divides b'^2 (possibly 0). But the highest even power of p dividing n divides b^2. Hence, if p^k divides b', it must also divide b, for all k. As this happens for all p dividing n, we have $b' \leq b$. Hence, $a < b$.

Using the fact that the square-free representation is unique we can count the semi-perfect squares in $[1, X]$ without having to worry about double-counting. For every choice of square-free a, we have to choose b to be between $(a+1)$ and $\lfloor\sqrt{X/a}\rfloor$. In fact, every such choice of b yields a semi-perfect square. Also, since $a < b$, and $ab^2 \leq X$, we have that $a \leq \sqrt[3]{X}$. We can afford to iterate through all possible choices of a (at most cube root of $8 \times 10^{16} \approx 380,000$) and for each a we have to evaluate a square root.

Counting semi-perfect cubes

We can show analogously that similar facts hold for semi-perfect cubes. Every positive integer can be written uniquely in the form ab^3, where a is cube-free and b is positive. A cube-free number is, as you might have guessed, a positive integer which is not divisible by a perfect cube greater than 1. We call this representation in the form ab^3 (where a is cube-free) a cube-free representation. Additionally, the number ab^3 (where a is cube free) is a semi-perfect cube if and only if $a < b$.

Although it is easy to count semi-perfect cubes as we have done for squares, the problem is that we are counting some numbers twice: those which are both semi-perfect squares and semi-perfect cubes!

One possible approach is to iterate through all the semi-perfect cubes, and check if it is a semi-perfect square too. (We add it to our count if it is not.) However, this approach turns out to be too slow. (There are over 2×10^8 semi-perfect cubes between 1 and 8×10^{16}.)

In our next step, we look more closely at the cube-free representations of numbers which are not semi-perfect squares.

Non-semi-perfect squares

A non-semi-perfect square (or a non-square for short) number n can be written as xy^3 (with x cube-free) and also as ab^2 (with a square-free) and $a \geq b$.

Suppose that a prime p divides both a and b. Then, p^3 divides n. p must also divide y, since x is cube-free. Consider now the number $m = x(y/p)^3$. m has the unique square-free representation $(a/p)(b/p)^2$. And since $a \geq b$, we know that $(a/p) \geq (b/p)$. m is thus a non-semi-perfect square too. Note that p no longer divides both (a/p) and (b/p) in the square-free representation of m. Let us call this operation $(*)$.

We now have a way to generate more non-squares from a non-square. We check that the reverse of this operation $(*)$ also works. Take a prime q, which does not divide a. Then the number $(aq)(bq)^2 = x(yq)^3$ is also a non-semi-perfect square. This is because $(aq)(bq)^2$ is a valid (and thus, the unique) square-free representation, and $a \geq b$ ensures that $aq \geq bq$. We also check that $x(yq)^3$ is the unique cube-free representation.

Taking any non-square number, if we apply the operation $(*)$ repeatedly, we will obtain a non-square S where $(*)$ cannot be applied. Suppose $S = xy^3$ (with x cube-free) and $S = ab^2$ (with a square-free, $a \geq b$). $(*)$ cannot be applied if and only if S has a and b co-prime. Let us call such a number a source. Let us now look at the properties of a source (which is also a non-square number) more closely. In a source S, since a is square-free and also coprime to b, it cannot share any factor with y^3. If it did, then there would be a prime p dividing a for which p^3 divides S, which implies p divides b; contradicting the fact that a and b are coprime. Hence, a divides x. Let $x = a \cdot x_2$. We have that $x_2 \cdot y^3 = b^2$. Note that since a and b are coprime, so are a and x_2, and a and y. We express the condition $a \geq b$ as: $a \geq \sqrt{x_2 \cdot y^3}$, or

$a \cdot x_2 \geq x_2 \cdot y \cdot \sqrt{x_2 \cdot y}$, or $x \geq k^3$, where $k = \sqrt{x_2 \cdot y}$. (Note: k is an integer since b is also one.)

Conversely, if we fix x, we verify which choices of k with $k^3 \leq x$ yield a source S. Working backwards, set $y = \frac{k^2}{(k^2,x)}$, and $a = \frac{x}{(k^2,x)}$. Now, $x \cdot y^3 = x \cdot y \cdot y^2 = a \cdot k^2 \cdot y^2 = a \cdot (k \cdot y)^2$. This is a square-free representation provided that $a = \frac{x}{(k^2,x)}$ is square-free. The inequality $x \leq k^3$ is equivalent to $k \cdot y \leq a$; hence $S = x \cdot y^3$ is indeed a non-square. Additionally, since a and $k \cdot y$ are coprime, S is also a source.

Bounding the number of sources

As we saw above, if we fix x, then each source which has x as its cube-free part can be associated with a unique cube k^3 not greater than x. Hence, the number of sources with cube-free part a particular x is at most $\sqrt[3]{x}$. Since we are interested in counting numbers of the form $x \cdot y^3 \leq N$, we have $x \leq \sqrt[4]{N}$. The total number of sources we have to consider is thus at most $\sqrt[4]{N} \cdot (\sqrt[3]{\sqrt[4]{N}}) \sim 430,000$. It seems feasible to iterate over all possible sources and quickly count the numbers which are derived from each source.

non-squares from sources

Suppose n is a number from which the source S is obtained by applications of $(*)$. Let $S = a \cdot b^2$ (with a square-free), then n can be recovered from S by repeatedly performing the reverse operation of $(*)$. In successive applications of the operation $(*)$, that is, 'removing' primes from both a and b, notice that we do not remove the same prime twice. That is, in the reverse operation, we can replace all the removed primes one by one to get back n. If P is the product of these primes, then $n = (aP)(bP)^2$. Since P is a product of primes not dividing a, it is a square-free number which is coprime to a. Indeed, for any square-free number Q coprime to a, the number $(aQ)(bQ)^2$ is a non-square from which source S is obtained. We see that the numbers from which source S is obtained are exactly the numbers $(aQ)(bQ)^2$, where Q is square-free and coprime to a.

Counting non-squares which are semi-perfect cubes

For a fixed cube-free x, we iterate through all sources S of the form $x \cdot y^3 (= a \cdot b^2; a$ square-free). Our sub-problem is now to count all the non-squares derived from S, which are also semi-perfect cubes and not greater than X. As we saw above, all non-squares derived from S are of the form $x(yP)^3 = (aP)(bP)^2$, where P is any square-free number coprime to a. The non-square is also a semi-perfect cube provided that $x < yP$, or $P > \frac{x}{y}$. We must also have $x(yP)^3 \leq N$, or $P \leq \frac{\sqrt[3]{N/x}}{y}$. Hence, our problem now reduces to counting the number of square-free numbers between two bounds which are coprime to a. Define $F(n, a)$ to be the number of square-free numbers between 1 and n, inclusive, which are coprime to a. We can assume that $a, n \leq \sqrt[3]{N} \leq 500,000$.

Inclusion-Exclusion Principle

Computing $F(n, 1)$ is easy. This is just the number of square-free numbers not greater than n. The square-free numbers themselves can be found by a procedure similar to the sieve of Eratosthenes. Starting with all the numbers in the required range, we iterate over perfect squares and scratch off their multiples—whichever numbers remain are exactly the square-free numbers. Let us now try for $a = 2$. $F(n, 2) = F(n, 1) -$ (square-free numbers $\leq n$ which are multiples of 2). We denote by $U(n, d)$ the square-free numbers $\leq n$ which are multiples of d. Then $F(n, 2) = F(n, 1) - U(n, 2)$. In fact, for any prime p, we can see that $F(n, p) = F(n, 1) - U(n, p)$.

For $a = 6$, a little bit of thought reveals that $F(n, 6) = F(n, 1) - U(n, 2) - U(n, 3) + U(n, 6)$. That is, we excluded square-free multiples of 2 and 3; but multiples of both 2 and 3 were excluded twice. Generalizing this for three prime factors, we get $F(n, pqr) = F(n, 1) - U(n, p) - U(n, q) - U(n, r) + U(n, pq) + U(n, qr) + U(n, pr) - U(n, pqr)$. Indeed, we are using the Inclusion-Exclusion principle. If $a = p_1 \cdot p_2 \cdot \cdots \cdot p_k$, by IEP we know that

$$U\left(\bigcap_{i=1}^{k} (\text{sq-free no.s divisible by } p_i)\right) = \sum_{S \subseteq [k]} (-1)^{|S|} (\text{sq.-free no.s divisible by all } p_i \text{ for } i \in S).$$

In general,

$$F(n, p_1 \cdot p_2 \cdot \cdots \cdot p_k) = \sum_{S \subseteq \{1,\ldots,k\}} (-1)^{|S|} U(n, \prod_{i \in S} p_i).$$

Here, we have used that $F(n, 1) = U(n, 1)$.

Time and Space Complexity

In order to compute $F(n, a)$, we need to add 2^k terms, where k is the number of prime factors of a. Recall that a is square-free, and that $a, n \leq M$, where $M = 500,000$. Since the product of the first 7 primes is 510510, greater than 500,000, a can have at most 6 distinct prime factors. Hence, the number of terms is $2^6 = 64$. Supposing each term can be precomputed, this adds up to $64 \cdot$ (no. of sources) operations, that is, at most $64 \cdot 430,000 = 27,520,000$ operations. This fits within the 2 second time limit.

Each term can in fact be pre-computed. In $O(M)$ time, we can compute a boolean table square_free$[1 \ldots M]$. For any square-free integer x, the quantities $U(1, x), U(2, x), \ldots, U(M/x, x)$ (where $M = 500,000$) require $O(M/x)$ space and can be read off from the square_free table in $O(M/x)$. The values themselves can be obtained by summing them. The total space (and also time) we need for the precomputation is therefore bounded by $M/1 + M/2 + \cdots + M/M \sim M \log M$. Computing the list of divisors of each square-free integer in $[1, M]$, as well as their number of distinct prime factors, can also be done in $O(M \log M)$ time using algorithms similar to the sieve of Eratosthenes.

Problem L. Sortish1

Meet in the middle

In cases when the input size is just barely too large for straight brute force, we can try to use a meet-in-the-middle strategy. This needs creativity in defining what the middle means for this problem. There are up to 14 positions that we need to assign numbers to. The 14 numbers may be divided in 7 small numbers and 7 large numbers (n numbers split the $||\lfloor m/2 \rfloor||$ smaller half and the $m - ||\lfloor m/2 \rfloor||$ larger half, m is the number of missing elements in the permutation). The concept of meet in the middle requires us to somehow merge the results of the placements of the smaller half with the results of placements of the larger half. This merging requires us to do simplifications and learn how to divide the problem of calculating the sortedness.

Sortedness added by a single number

Initially, there are already up to 1986 numbers in the permutation, and their positions won't change. The sortedness value created by pairs of already-assigned numbers won't change. So we can calculate it at the beginning and then just ignore these pairs when doing the assignment.

Sortedness by each assignment

If we assign the j-th missing number to the i-th unknown position, this new missing number will have interactions with the 1986 other numbers. The sortedness added by these interactions depends solely on i and j. We can precalculate this so that we don't need to do 1986 comparisons every time we assign a number to a position.

Combined sortedness

There is also sortedness contributed by pairs of the numbers we add to the permutation. The nice thing is that we can ignore all the other 1986 numbers when calculating this sortedness. So from now on, we will care about the m positions and their sortedness only.

Which places are small

The first difficulty in splitting the problem in two halves is that any of the m positions can have the smaller or larger missing numbers. So the first step lies in defining which places will hold small numbers and which places will hold large numbers. There are $\binom{m}{m/2}$ combinations possible. Imagine for $m = 6$, we had an assignment 001011, where 0s are the places that go to small numbers and 1s the ones that go to large numbers. We can actually calculate the sortedness contributed by pairs of (small, large) numbers by only knowing this assignment: 001011 (We know that each "large" number is larger than a "small"

number, so the sortedness of sequence 001011 will be the sortedness contributed by pairs of (small, large) numbers). Now we only need to worry about sortedness contributed by pairs of (small, small) and (large, large) numbers.

After the small/large positions are decided, we decide which small position gets which small number and which large position gets which large number. It is very useful to do a meet-in-the-middle process for each assignment of small/large positions, so for example, we run a meet-in the middle for 000111, then 001101 then 011010 and so on.

Permutations

Now the problem is to count the number of permutations for each assignment. Imagine the small/large assignment is fixed. There are $(m/2)!$ permutations of small numbers and $(m-m/2)!$ permutations of large numbers. The meet-in-the-middle works as follows: First simulate the $(m/2)!$ small permutations, and save some information in memory for each of them. Then we simulate the $(m-m/2)!$ large permutations and somehow use the memory saved in the previous step.

So imagine we are trying a permutation of large numbers. Remember that assigning each specific number j to a position i introduces a "sortedness" value with the other $n-m$ numbers that we precalculated in a previous step. So for each of the large numbers we add that precalculated sortedness up, let's name it A. Also important is the "sortedness" introduced by pairs of large numbers. We can calculate this as the sortedness of the large permutation we just created, this will be B. Don't forget the sortedness created by pairs of initially-known numbers: C. And the sortedness created by pairs of large and small numbers (dependent on the assignment in the previous section): D. $A+B+C+D$ is equal to the sortedness value ignoring small numbers. So if from the wanted "sortedness" we subtract $A+B+C+D$, we will find s, the value of the "sortedness" we want to be created by the small permutation.

Imagine that we had a table in memory: small[s] returned the total number of small permutations that contribute a value s to the permutation. Then for the large permutation we just created, we know that there exist small[s] small permutations that can be paired with it. So we just increase the result by small[s] for each large permutation.

Now we know what to do in the small step. We should simulate all $(m/2)!$ small permutations and for each of them, calculate the "sortedness" s it introduces. Remember that each small number j we add to a position i will introduce a "sortedness" value interacting with the $n-m$ initially known numbers. We also need the "sortedness" of the small permutation for the sortedness introduced by (small, small) pairs. For each small permutation, we find s and increment small[s].

Complexity and a needed optimization

For each of the $\binom{m}{m/2} = \frac{m!}{((m/2)!)\cdot((m-m/2)!)}$ small/large assignments we do $m!$ permutations. For each permutation, we use an $O(m)$ loop to calculate the sortedness added by pairs of new and old numbers. But we currently also use an $O(m^2)$ loop to find the sortedness of each permutation. This yields: $O(\frac{m!}{(m/2)!}\cdot m^2)$ complexity. It is too large, even for 6 seconds. Note that the sortedness of each permutation of $m/2$ elements can be precalculated in a previous step. This will enable a solution that runs in less than 2 seconds in topcoder's servers: Note that the total complexity is: $O(N^2+\frac{m!}{(m/2)!}\cdot m+(m\cdot m)\cdot m!)$.

Problem M. Yet Another Board Game

Let us begin by turning the board into a matrix of 1s and 0s. Black cells are represented by 0s and White cells by 1s. The objective is to make all cells into 0s. The two operations toggle some of the cells, the 0s become 1s and the 1s become 0s. We can see it as performing the xor operation between some part of the matrix and the two following shapes:

$$\begin{matrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{matrix}$$

$$\begin{matrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{matrix}$$

A single row changes everything

The problem is all about picking the number of times (0 or 1) we apply one of the two moves on a single cell, and of course the types of cells we use. The order in which the moves are done doesn't really matter. Let us do it from top to bottom.

Let us decide what moves to do in the first row. A subset of the cells will be used in moves. The type of move to use in this row is also a variable. We are free to choose any subset and any type of move. The moves we perform will only affect the first and second row, because for a cell $(0, i)$, the affected cells would be: $(0, i-1)$, $(0, i+1)$ and $(1, i)$ (Depending if they exist or not) and $(0, i)$ in case a type 2 move was used. Cell $(-1, i)$ does not exist. The crux of the matter is that after we decide which cells to use and the kind of move, the first two rows will contain some 1s and 0s:

$$\begin{matrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{matrix}$$

Use 101(type 1) centered at cells(0, 2), (0, 5) and (0, 6):

$$\begin{matrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{matrix}$$

$$\begin{matrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{matrix}$$

We should then progress to decide what moves to use for the second row. The key observation is that this is our last opportunity to alter the contents of the first row. A move of any type in cell $(1, i)$ will forcefully toggle the contents of cell $(0, i)$. This means that for every cell $(0, i)$ that is 1, we must make a move using cell $(1, i)$ (So that $(0, i)$ becomes 0) and if $(0, i)$ is 0, we cannot make a move using cell $(1, i)$. In the example above, we are forced to make moves using cells $(1, 0)$, $(1, 1)$, $(1, 4)$, $(1, 6)$ and $(1, 9)$. It appears the only decision we can afford to make is the type of move for this row. Should we use a type 1 move or a type 2 move? In reality, remember that we are making a move at $(1, 6)$ and that previously, we did a move at $(0, 6)$, so the type of move in both cells (they share a column) must be the same. Just like that, we are forced to make type 1 moves at specific cells of row 1.

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{matrix}$$

We should proceed to the third row. This is our last chance to set the contents of the second row to 0. Which means that we need moves at $(2, 0)$, $(2, 2)$, $(2, 4)$, $(2, 5)$ and $(2, 7)$. In addition, the move type for $(0, 2)$ is already known. We can use this logic on each row, using the previous rows to select the moves. Until we decide the moves for the last row, if after these moves, the last row is full of zeros (and since the previous moves were made so that each of the previous rows ended full of zeros too) then we reached the objective in a specific number of moves.

Information from previous rows may also allow us to know the move type for the current row. Although there can be a case in which there is not a specific requirement. Consider this new example:

$$\begin{matrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{matrix}$$

For the first row, we use only a type 2 move at cell $(0, 2)$:

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{matrix}$$

In order to set the first row to 0, we need a move at $(1, 8)$, but this time we can choose any type of moves. It is incidentally best to pick a type 2 move.

There is also a corner case. What if we need to set some cells in a row, but two of those cells have columns that have already-chosen move types that contradict each other, then it is not possible to make any progress in the given situation.

This logic that picks one of 2^n subsets of cells and the type for the moves in the top row and then fills the remaining rows from top to bottom, deciding the movement type if possible and cropping the invalid cases can be implemented using a recursive backtracking. It is possible to implement the recursion in a way that we only need one binary operation to update the rows according to the application of a move to a subset of cells in a row. If such an optimization is used, this recursive approach will actually pass in time. The next section will develop an iterative solution that also demonstrates that the time complexity of the recursive solution is fast enough.

Row and columns

In the previous section, the moves of the top row and their type are decided. Then, for each row, the moves are deduced from the previous row and their type may be decided. At the end, a sequence of moves that reaches the objective can be represented solely by the subset of cells from the first row that participate in moves and the movement types decided for each row.

Turns out it is also possible to represent a valid sequence of moves by the cells chosen for the first row and the types chosen for the columns. Let us assume the types to be used in each column were already chosen and that we choose the initial sets to be used in moves in the top row:

$$\begin{matrix} 2 & 1 & 1 & 1 & 1 & 2 & 1 & 1 & 2 & 1 \end{matrix}$$

(Move types decided for each column)

$$\begin{matrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{matrix}$$

Apply 111 centered at cell (0, 5):

$$\begin{matrix} & & & & 1 & 1 & 1 & & & \\ & & & & 0 & 0 & 0 & & & \\ & & & & 0 & 0 & 0 & & & \end{matrix}$$

$$\begin{matrix} 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \end{matrix}$$

Since the move types for the columns are already chosen, this logic should help us move forward row by row. If we ever encounter a contradiction, we backtrack. This approach lets us solve the problem iteratively. (0,5) was the only move in row 0, we proceed to row 1, and once again, the only way to set the 1s in row 0 to 0s is to use specific row 1 cells in moves: (1,1), (1,3), (1,4), (1,6) and (1,7). This time,

the move types of the columns are already decided. We just need to make sure they are all the same for each of the chosen cells (so that the row only uses one move type). If they are not, the initial combination of moves and column types was invalid. We can once again, use this logic for each of the rows.

Let us say the cells in a row i that are used in moves are: $(i, c_1), (i, c_2), \ldots, (i, c_t)$. Because row i must use moves of only one type, the columns $c_1, c_2, \ldots, c_t$ must be of either type 1 or type 2, but not both. (The exception is when the set of used columns is empty). This is true even for row 0 and limits the number of possible combinations of column types and cells used in row i. We pick one of 2^m subsets of columns, name the subset colType. The cells used in moves in row 0 will also be part of a set, u_0. u_0 must either be a subset of colType or a subset of the complement of colType.

There are $O(3^m)$ ways to pick two subsets A and B of a set X of m elements such that $A \subseteq B$. As a quick proof: It is like for each of the m elements of X, we decide one of three choices: 0, 1 and 2. 0 means that the element does not belong to A and does not belong to B. 1 means that the element belongs to A but does not belong to B and 2 means the element belongs to B (and therefore A). We can use this fact to show that there are $2 \cdot 3^m$ pairs $(\text{colType}, u_0)$. There are 3^m pairs in which $(u_0 \subseteq \text{colType})$ and 3^m other pairs in which $(u_0 \subseteq \sim \text{colType})$ (Where $\sim$ is the complement operator).