Hosted by:

The 2026 ICPC Europe Championship

Solutions

A – Three Circles

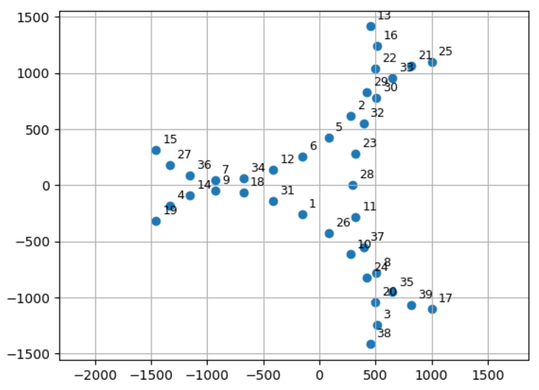
Author: Kevin Pucci

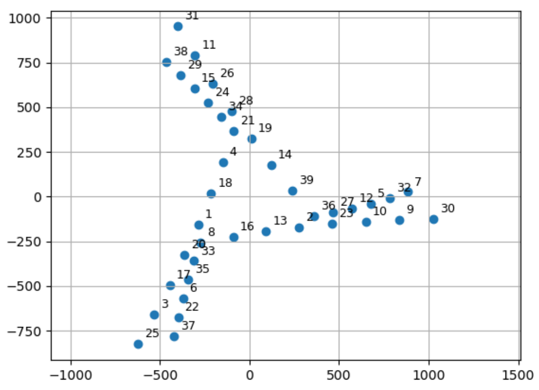
In this problem, we are given n points in the plane and we need to partition them into three disjoint convex sets.

First, consider the case where we divide the points into two sets; call them A and B. This is a classical problem, and the solution is as follows: if we have two non-intersecting convex sets, we can always find a straight line that separates them. Furthermore, by tweaking that line, we can slide it until it touches a point from set A, then rotate about that point until it touches a point from set B. Thus, we can always guarantee a line that passes between a point of one set and a point of the other. We can therefore iterate over all such lines (i.e. all pairs of points) and, for each candidate line, test whether the sets on both sides are convex (by computing their convex hulls).

We first sort the points by x-coordinate (so we can use a linear-time convex hull later) in $O(n \log n)$, and then for each of the $O(n^2)$ candidates we run an $O(n)$ test. Overall, an $O(n^3)$ solution.

For division into three sets (A, B and C), one tempting idea is to find a line that separates A from $B \cup C$, and then recursively solve the problem for B and C. However, this does not work, since there are configurations with no such separating line (see the image on the left below).





Another idea is to find three rays with a common origin (e.g. pick a point, do a polar sweep, and expect the sets to appear as contiguous segments in the order). But this also does not work (see the image on the right).

Therefore, we need a more careful approach, but we can still adapt the idea of a separating line.

First, we strengthen the result for two sets A and B. We can prove that there is a separating line that passes through a point p_A in A and a point p_B in B, such that when moving from p_A to p_B we have A on the left. (This is very similar to how two circles have two internal tangent lines.)

Thus, if we pick three lines $\ell_{A,B}, \ell_{B,C}, \ell_{C,A}$ (where $\ell_{X,Y}$ separates sets X and Y with X on the left), we need to test if

$$convex(A, B, C) + convex(B, C, A) + convex(C, A, B) = n,$$

where $convex(X, Y, Z)$ tests if the set of points on the left of $\ell_{X,Y}$ and on the right of $\ell_{Z,X}$ is a convex polygon, and returns the number of points in this set (otherwise it returns 0). We have $O(n^6)$ candidates for lines, and the test can be performed in $O(n)$, so the total time complexity is $O(n^7)$.

We can optimize this by precomputing all possible values of $convex(X, Y, Z)$. Each value depends on two lines, so we have $O(n^4)$ values, and precomputation can be done in $O(n^5)$. With this, each candidate for three lines can be tested in constant time, so the overall complexity is $O(n^6)$.

Since we iterate over all possible partitions, it's easy to store a solution for each unordered triplet.

B – Bitwise Beach

Author: Kamil Dębowski

First we compute the multiset of distances between vertical lines. We have two observations:

- Since the sum of numbers in the multiset is n, there are only $O(\sqrt{n})$ distinct values.
- Two equal numbers generate the same set of zones (in a vertical strip), so their areas cancel under bitwise xor (since $x \oplus x = 0$). Thus such a pair can be removed from the multiset.

So we end up with a set of $O(\sqrt{n})$ values: distances that occur an odd number of times in the multiset. We do the same for the multiset of distances between horizontal lines. Now we only have to consider $O(\sqrt{n} \cdot \sqrt{n})$ zones, so xoring them takes $O(n)$ time.

C – Copy-paste

Author: Tomasz Idziaszek

We need to create a string of n letters a by either appending a single letter a to the end of the current string or copy-pasting the current string. To copy-paste, we select the whole string with Ctrl-A, then copy it into the clipboard with Ctrl-C. After that the whole string is still selected, so we must deselect it first, and the only way to do this is to press Ctrl-V. Pressing Ctrl-V again concatenates the contents of the clipboard. Thus by pressing Ctrl-A, Ctrl-C, and then Ctrl-V d times, we replace the current string with d copies of itself.

So when typing the password, we alternate between two operations: either we increase the length by typing individual letters, or we multiply the length by d using a sequence of $2 + d$ operations.

Now consider working backward from the required length n to 0. Any way to type n iteratively decreases n until we reach some composite number and then divide by a factor of this number. In an optimal solution, we never decrease n more than the factor we divide by afterwards: subtracting d and then dividing by d gives the same value as first dividing by d and then subtracting 1.

The solution therefore consists of repeatedly choosing a factor d, subtracting $n \bmod d$ times to make n divisible by d (costing $n \bmod d$ operations), and then dividing by d (costing $2 + d$ operations).

If we always pick $d = 3$, then we use at most 7 operations to divide n by 3, so there exists a solution using at most $7 \log_3(10^{12}) < 177$ steps. It follows that we never need to consider a d larger than 175: a single division will already use more steps. Moreover, for a composite number $d = a \cdot b$ we can replace dividing by d (costing $2 + d$ operations) with dividing by a and then by b (costing $2 + a + 2 + b$ operations). This is ok if $2 + a + b \leq d$, and this only fails for $d = 4, 6$. Thus d should be either 4, 6, or a prime number less than 175. (The bound 175 is very loose; the judges were able to prove a bound of 73, but were not able to find a case where the required factor d is larger than 23.)

Finally, to find the best solution we use Dijkstra's algorithm on a graph where vertices correspond to string lengths, and edges correspond to dividing by values of d. Since all edges go from larger to smaller values, we can avoid recomputation by iterating over vertices in decreasing order and using memoization to store the best costs found so far.

Although it is difficult to give a good upper bound on the number of computation paths to show that the algorithm is fast enough, it is easy to implement it, try the largest value, and observe that it runs efficiently. However, there are more optimizations one could implement to speed up the algorithm, for example a meet-in-the-middle approach that uses dynamic programming to precompute optimal solutions for $n \leq 10^5$, or pruning Dijkstra by using a lower bound of $6 \log_4 n$, which comes from dividing evenly by 4.

D – Deque Sort

Author: Arkadiusz Czarkowski

Notice that each phase moves the subsequence of prefix maxima to the end and reverses the rest of the sequence. As the name of the problem suggests, after some number of phases we obtain a sorted sequence. This happens after at most n phases, since each application extends the correctly placed suffix by at least one element.

The key observation is that if we track the lengths of these subsequences of prefix maxima throughout the process, their total length is $O(n)$, assuming we ignore the suffix that is already in place and never moves. This means we can simulate the process on, for example, a BST that supports split/merge, reverse, and find-next-greater-element operations.

The problem can also be solved with segment trees. Below we describe such a solution, which also yields the time bound.

The main idea is that rather than moving maxima and reversing the rest of the sequence, we remove maxima (by replacing them with -1s) and concatenate them in reversed order at the front of the sequence. In the next step we look for maxima from right to left and concatenate them at the back of the sequence.

Thus the sequence consists of three pieces: left extension, middle, and right extension. Initially the middle is the original permutation, and extensions are empty. In even steps we scan for maxima from left to right and add them to the left extension; in odd steps we scan from right to left and add them to the right extension. We also remember the currently largest element, and if it appears at the end of some extension, we remove it permanently. Thus at the beginning of each step, the current global maximum lies in the middle part.

Note that the left extension is always sorted in descending order, and the right extension in ascending order. Thus in each step we take one maximum from an extension and several maxima from the middle. (In fact we do not even have to move the maximum from the extension, since it is already in the correct place.) Each element can be taken from the middle only once, so the number of maxima we touch is at most n. Hence the length of each extension is at most n.

We build a static segment tree of length n that represents the middle part (for extensions we only need simple vectors). Elements that are not currently part of the sequence are set to -1. We need operations: find the first element greater than x from the left and from the right (we store only the maximum in each internal node of the segment tree), and find the i-th element that is not -1 (we store counts of elements $\neq -1$).

One tricky part is deciding when to stop. If we wait until all elements are removed permanently, we may add an unnecessary phase, since the sequence can already be sorted while some elements remain in the extensions. One way to handle this is to run the algorithm twice: first, compute the number of phases T until all elements are removed; second, answer queries, but on phase $T-1$ additionally test in $O(n)$ whether the sequence is already sorted, and if so, stop one phase earlier.

The total time complexity is $O(n \log n)$.

E – Evaluation

Author: Marcin Mucha

First, some notation: with the usual order of operations, each expression is a sum of *terms*, each term is a product of *factors*, and each factor is a concatenation of *digits*.

To solve the problem we use simple dynamic programming. We iterate through the string from left to right and represent all the expressions we could obtain so far using a state consisting of four values (T, A, B, C):

- T: the total number of expressions,
- A: the sum of all expressions, with each final term excluded,
- B: the sum of all final terms, with each final factor excluded,
- C: the sum of all final terms.

For example, after reading 347522351, the expression $3 \cdot 4 + 75 \cdot 2 + 2 \cdot 3 \cdot 51$ contributes value 1 to T, value $3 \cdot 4 + 75 \cdot 2$ to A, value $2 \cdot 3$ to B, and value $2 \cdot 3 \cdot 51$ to C.

When considering a new digit d, there are three possibilities for the final symbol: addition, multiplication, or concatenation. We now show how to compute the new state.

- T: Since there are three options for the final symbol, the total number of expressions is multiplied by 3, hence $T_{\text{new}} = 3 \cdot T_{\text{old}}$.
- A: All old non-final terms remain non-final in all cases, hence we include three times the old sum of non-final terms. If the final symbol is a $+$, then all final terms become non-final, so we also include the sum of final terms once. Overall, $A_{\text{new}} = 3 \cdot A_{\text{old}} + C_{\text{old}}$.
- B: When the next symbol is a $+$, the final term consists of a single factor, hence the final term without its final factor is just the empty product 1. When the next symbol is a $\cdot$, we append a new factor, so the new final term without its final factor is the old final term. Finally, when the next symbol is concatenation, the final term without its final factor is unchanged. Overall, $B_{\text{new}} = T_{\text{old}} + C_{\text{old}} + B_{\text{old}}$.
- C: When the next symbol is a $+$, the final term consists of a single factor, hence the final term is the digit d. When the next symbol is a $\cdot$, the final term is multiplied by d. Finally, when the next symbol is concatenation, we multiply the final factor by 10 and add d. For the final term, we therefore multiply by 10 and add d times the final term without the final factor. Overall, $C_{\text{new}} = d \cdot T_{\text{old}} + d \cdot C_{\text{old}} + (10 \cdot C_{\text{old}} + d \cdot B_{\text{old}})$, or equivalently $C_{\text{new}} = 10 \cdot C_{\text{old}} + d \cdot B_{\text{new}}$.

At the end, the sum of all expressions can be found as $A + C$.

F – Fix the Coloring

Author: Ihor Barenblat

If we decide not to paint a nontrivial clique red, the problem reduces to the following: paint some vertices red so that every edge connects vertices of different colors. We can reduce this to 2-SAT: for every vertex v we create a Boolean variable x_v that is true iff v is repainted red.

For every edge u, v whose endpoints currently have the same color (black or white), we add the 2-SAT clause $x_u \vee x_v$, since at least one of these vertices must be repainted red. Moreover, for every edge u, v we add the clause $\neg x_u \vee \neg x_v$, since we cannot repaint both endpoints of an edge red.

Now assume that we decide to paint some clique. Fix a vertex s from this clique (we will iterate over all n candidates for s). Let S be the set containing s and all vertices adjacent to it. Note that the red clique must be a subset of S.

Again, for every vertex v we have a 2-SAT variable x_v that says whether v is repainted red. We add the following clauses:

- for vertex s we add $x_s \vee x_s$ (vertex s must be red),
- for all edges u, v of vertices with the same color (black or white) we add $x_u \vee x_v$,
- for edges u, v where at most one of the vertices belongs to S we add $\neg x_u \vee \neg x_v$,
- for pairs $u, v \in S$ that are not connected by an edge, we add $\neg x_u \vee \neg x_v$ (at most one of them can belong to the red clique).

This 2-SAT instance has a solution iff there is a proper coloring with vertex s in the clique. The size of the instance is $O(n + m + \deg(s)^2)$, where $\deg(s)$ is the degree of vertex s.

Iterating over all candidates for s, the total running time is $O\big(n^2 + nm + \sum_v \deg(v)^2\big) = O(nm)$.

G – Good Permutations

Author: Valerio Stancanelli

In a good permutation, the value can drop by at most 1 between consecutive positions, i.e. $p_{i+1} \geq p_i - 1$. Hence the permutation can be viewed as a concatenation of *descending blocks* where we decrease by exactly 1 until the block ends, for example:

$$3\ 2\ 1 \mid 5\ 4 \mid 6 \mid 9\ 8\ 7.$$

A key observation is that a fixed value determines a whole descending block segment. Assume within a query range that we fix $p[x] = y$ at position x with $x < y$. Then, to avoid a decrease larger than 1, we must have

$$p[x] = y,\ p[x+1] = y-1,\ \ldots,\ p[y] = x.$$

Equivalently, within that block the invariant $i + p[i] = x + y$ holds for all covered positions i.

First, we analyze the fixed elements. Let $p[x_1] = y_1$ and $p[x_2] = y_2$ be two *consecutive* fixed positions with $x_1 < x_2$.

- $y_1 \geq y_2$: then both fixed elements must lie in the *same* descending block. In that case the invariant implies
$$x_1 + y_1 = x_2 + y_2.$$
If this equality fails, the answer is "no".
- $y_1 < y_2$: then the two fixed elements must lie in *different* blocks.

Next, we must ensure that two *different* blocks do not overlap.

Fix a query (l, r). A fixed element at original index x appears in the query at local position $i = x - l + 1$ and has fixed value y. The block containing it covers the range

$$[\min(i, y),\ \max(i, y)].$$

Therefore, for two consecutive fixed elements that must be in distinct blocks ($y_1 < y_2$), the blocks are disjoint if and only if

$$\max(i_1, y_1) < \min(i_2, y_2),$$

where $i_j = x_j - l + 1$. Expanding yields the constraints on l:

$$\begin{cases} i_1 < y_2 \iff l > x_1 - y_2 + 1, \\ y_1 < i_2 \iff l < x_2 - y_1 + 1. \end{cases}$$

The solution consists of two steps:

1. **Preprocessing.** For each fixed element we compute constraints on admissible values of l:

- $l > x_{\text{prev}} - y + 1$ and $l < x - y_{\text{prev}} + 1$, where $(x_{\text{prev}}, y_{\text{prev}})$ is the *last* fixed element of the *previous* block,
- $l > x - y_{\text{next}} + 1$ and $l < x_{\text{next}} - y + 1$, where $(x_{\text{next}}, y_{\text{next}})$ is the *first* fixed element of the *next* block,
- $l > x_{\text{prev}^*}$, where $(x_{\text{prev}^*}, y_{\text{prev}^*})$ is the *last* fixed element of the *previous* block such that $y_{\text{prev}^*} \geq y$, if such an element exists.

2. **Querying.** For each query (l, r) we check:
   - All minimum and maximum bounds on l inside the query range are compatible with this l.
   - Every fixed value satisfies $y \leq r - l + 1$ (a good permutation of length k cannot contain values greater than k).

All range checks can be done with simple data structures such as a segment tree or a sparse table. Time complexity is $O((n + q) \log n)$.

H – House

Author: Jakub Onufry Wojtaszczyk

At a high level, the solution is as follows:

- we need to reduce the number of (x, y) pairs we need to consider,
- we need to be able efficiently decide whether a given pair (x, y) is feasible.

First, a remark: n is a red herring, since we use at most the $4k$ longest planks.

Part one. We perform a binary search on the result, i.e., on the product $R = xy$. We assume $x \geq y$.

If it is possible to fit more than $4k$ segments of length $\sqrt{R}$, then we are done. Otherwise, for each of the $4k$ planks we compute how many segments of length $\sqrt{R}$ can be placed in it (we know that this will be the maximum possible number of x's in that plank).

We know that if it is possible to achieve the product R, then it can be done so that some plank is completely filled. (If not, then we can increase x and decrease y, and eventually one plank must fill up.)

So we guess which plank it is, and how many x's we put into it (say α). There are fewer than $8k$ possibilities: for each plank there is the option of putting 0, and there are fewer than $4k$ options overall with a positive count. For each of these possibilities we guess β, i.e., how many y's we put into that plank.

Now we have the equation $\alpha x + \beta R/x = a_i$, which is a quadratic equation, so it has at most two solutions (and if $\alpha = 0$, which occurs in half the cases, it has only one). Therefore, for a given R we have $O(k^2)$ possibilities for the pair (x, y).

Part two. For each plank we have already computed how many x's fit; let the total be L. We know that $L < 4k$. If $L < 2k$, then the answer is clearly "no".

We start greedily: in each plank we insert as many x's as possible, and then as many y's as possible. If at least $(4k - L)$ y's fit, then it is feasible (because we can always replace an x with a y). So assume fewer than that fit.

If a solution exists, it must look like this: in some planks we insert slightly fewer x's (so that the total becomes $2k$), and in their place we insert y's (and we hope that in at least some planks we insert more y's than the number of x's we removed).

At the beginning (when all possible x's and y's are inserted), on the i-th plank we still have $a_i - \lfloor a_i/x \rfloor x - \lfloor \frac{a_i - \lfloor a_i/x \rfloor x}{y} \rfloor y$ units of free space (because we removed as many x's as possible, then as many y's as possible). Let us denote this number by R_i.

For convenience, define the functions:

- $\text{count}(R) = \lfloor \frac{R}{y} \rfloor$ (how many y's fit into distance R),
- $\text{rem}(R) = R - \text{count}(R)y$ (how much space remains after inserting the maximum number of y's).

Thus, $R_i = \mathrm{rem}\big(a_i - \lfloor a_i/x \rfloor x\big)$.

We perform a dynamic programming procedure that goes through the x's one by one (deciding whether each stays or is removed), with the x's belonging to the same plank grouped together.

We define $DP[l][j] = (a, b)$ to mean that after processing the first l x's and keeping j of them, we have gained a additional y's, and on the current plank we still have b units of free space. We compare pairs lexicographically. We set $DP[0][0] = (0, 0)$.

We attempt to compute $DP[l][j]$. If the l-th x is on a new plank, we have two options: keep it or remove it.

If we keep it, $DP[l][j] = (DP[l-1][j-1].\mathrm{first}, R_i)$. If we remove it,

$$DP[l][j] = \big(DP[l-1][j].\mathrm{first} + \mathrm{count}(x + R_i), \mathrm{rem}(x + R_i)\big)$$

and we take the better of the two options.

If the l-th x is on the same plank as the $(l-1)$-th, the logic is similar:

If we keep it, we take $DP[l-1][j-1]$. If we remove it,

$$DP[l][j] = \big(DP[l-1][j].\mathrm{first} + \mathrm{count}(DP[l-1][j].\mathrm{second} + x), \mathrm{rem}(DP[l-1][j].\mathrm{second} + x)\big).$$

Finally, if $DP[L][k].\mathrm{first} \geq k$, then it is feasible.

The dynamic programming can also start by placing into the plank that is supposed to be full for the chosen pair (x, y).

Overall, the algorithm runs in $O(k^4 \log(\mathrm{result}))$.

I – Palindromes

Author: Jakub Onufry Wojtaszczyk

Core lemmata. We say that string $S[0..|S|)$ has a period $p \in [1..|S|)$ if $S[i] = S[i+p]$ holds for all $i \in [0..|S|-p)$.

Lemma 1. *If S is a palindrome, and $|S| = a$, then a prefix of S of length $b < a$ is a palindrome if and only if S has period $a-b$.*

Proof. We check both directions.

Assume the prefix of length b is palindromic. Take any x such that $x + a - b < a$. We want to prove $S[x + a - b] = S[x]$.

- $x + a - b < a$ implies $x < b$,
- so $S[x]$ lies inside the palindromic prefix,
- hence $S[x] = S[b - x - 1]$,
- and since S is a palindrome,

$$S[b-x-1] = S[a-(b-x-1)-1] = S[x+(a-b)].$$

Conversely, assume S has period $a-b$, and take any $x < b$. We want to show $S[x] = S[b-x-1]$.

- $x < b$ implies $x + a - b < a$,
- so by periodicity $S[x] = S[x + a - b]$,
- and since S is palindromic,

$$S[x+a-b] = S[a-(x+a-b)-1] = S[b-x-1]. \qquad \square$$

Lemma 2 (Periodicity lemma)**.** *If a string S has periods R and T, with $|S| \geq R+T$, then it also has a period $\gcd(R,T)$.*

Proof. Assume without loss of generality that $R > T$. It suffices to prove that S has period $R-T$, after which Euclid's algorithm gives the claim.

Take any x such that

$$0 \leq x \leq x + (R-T) < |S|.$$

We want to show $S[x] = S[x+(R-T)]$.

Since $R+T \leq |S|$, either $x-T$ or $x+R$ lies within $[0..|S|)$. Thus, we can move from x to $x+R-T$ in two valid period steps, which implies equality of the characters at these two positions. $\square$

Lemma 3. *If S is a palindrome with period R, then the string S' of length $|S| + R$, defined by*

$$S'[x] = \begin{cases} S[x], & x < |S|, \\ S[x - R], & x \geq |S|, \end{cases}$$

is also a palindrome.

Proof. We have $|S'| = |S| + R$. For any position x, we want to show

$$S'[x] = S'[|S'| - x - 1].$$

Since $|S| > R$, at least one of the indices x or $|S'| - x - 1$ is smaller than $|S|$; otherwise we would have

$$x + |S| + R - x - 1 \geq 2|S|,$$

which implies $R - 1 \geq |S|$, a contradiction.

Assume without loss of generality that $x < |S|$. Then

$$S'[x] = S[x] = S[|S| - x - 1] = S'[|S| - x - 1],$$

since S is a palindrome.

If $|S| + R - x - 1 < |S|$, then

$$S'[|S| - x - 1] = S'[|S| - x - 1 + R] = S'[|S'| - x - 1]$$

by periodicity. Otherwise, the equality follows directly from the definition of S' for indices $\geq |S|$. In both cases, we get $S'[x] = S'[|S'| - x - 1]$, as desired. □

Solution. Assume the values a_i are listed in a descending order. If 1 is not present in the sequence, we append it explicitly.

We start with the answer 1 that we will be multiplying by powers of 2 (modulo m). The expression $2^t \bmod m$ can be evaluated in $O(\log t) = O(\log N)$ time using exponentiation by squaring.

First, if $N > a_1$, then multiply the answer by 2^{N-a_1} and solve the problem for $N = a_1$. In this case, we have no information about the last $N - a_1$ bits of the string, so each such bit multiplies the number of solutions by 2.

We now consider a more general problem: we are given several prefixes of lengths $a_1 > \cdots > a_k$ known to be palindromes, the whole string has length a_1, and possibly there exists $R < a_1$ such that the whole string has period R (i.e. $W[x] = W[x + R]$ whenever both indices are valid).

We solve the problem recursively by reducing it to a smaller instance, possibly increasing the number of free bits. We apply the following four reductions until $a_1 = 1$.

1. If R exists and $R \geq a_1/2$, remove R and add the information that there is a palindromic prefix of length $a_1 - R$.

2. If a_2 exists and $a_2 > a_1/2$, remove a_2 and instead add the information that the whole string has period $a_1 - a_2$.

If the string already had a period R, then (since reduction 1 is applied first) we have $R < a_1/2$ and also $a_1 - a_2 < a_1/2$, so by Lemma 2 we replace R by $\gcd(R, a_1 - a_2)$. If there was no period, we take $a_1 - a_2$ as the new period.

This reduction is applied repeatedly as long as there exists another palindromic prefix longer than half of the string.

3. If a_2 exists $a_2 < a_1/2$ and there is no period, then the total number of solutions equals the number of solutions for $a_2, a_3, \ldots$ times $2^{\lceil a_1/2 \rceil - a_2}$ (modulo m).

   Any solution for the prefix can be extended to a solution of the whole string by freely choosing the bits up to half of $|W|$ and then reflecting. Conversely, any solution of the whole maps to exactly one solution of the prefix.

4. If a_2 exists $a_2 < a_1/2$ and there exists a period $R < a_1/2$, replace a_1 by $a_1 - kR$, where k is the largest integer such that
$$a_1 - kR \geq \max\{R, a_2\}.$$
   The solution for the reduced instance extends to the original one by Lemma 3.

The reductions never remove element 1 from the sequence. In the end we multiply the result by 2 (modulo m), which corresponds to the free bit at the first position of the string.

Complexity analysis. Each application of reductions 3 or 4 reduces a_1 by a constant fraction. Reduction 3 reduces it by at least a factor of 2. Reduction 4 reduces it to at most 3/4 of its previous value, since otherwise we would have $R \leq a_1/4$, contradicting the maximality of k.

Therefore, reductions 3 and 4 together are applied at most 145 times, because $(4/3)^{145} \geq 10^8$.

Reduction 1 is applied at most once per application of reduction 3 or 4, so also at most 145 times. Reduction 2 decreases the number of palindromic prefixes by 1. Initially there are L such prefixes, and every application of reduction 1 (which adds a prefix) is preceded by reduction 3 or 4 (which removes one), so reduction 2 is applied at most L times.

If the palindromic prefixes are stored in a heap or a set, all operations take $O(\log n)$ time. Exponentiation and computing a gcd take $O(\log N)$ time. Thus, the total runtime is
$$O\big(n(\log n + \log N)\big).$$

J – Jewel Guards

Author: Jakub Radoszewski and friends

The first step is to compute, for each unit of time, the subset of guards that work at this unit. Actually, we can process in chronological order all events that consist of a guard starting and ending work (i.e., the left endpoints and the excluded right endpoints of the intervals). At all time units between two consecutive events, the subset of guards is the same. Thus the night can be decomposed into at most $2c$ intervals of time, where c is the number of intervals in the input, with equal subsets of guards that work. This decomposition can be computed in $O(c \log c)$ time by any fast sorting algorithm or in $O(c \log k)$ time by e.g. Heap Sort. We aggregate this data so that, for each subset of guards M, we store as $t[M]$ the total time during which exactly guards from the set M work. Sets of guards are represented as bitmasks in $[0..2^k)$.

Let us say that a subset (bitmask) of guards is *moderately safe* if, during at least m time units of the night, at least one guard from the bitmask watches the jewels. We will first solve an easier task and compute all moderately safe bitmasks. A bitmask M is moderately safe if

$$\sum_{M' \,:\, M' \cap M \neq \emptyset} t[M'] \geq m,$$

i.e., if

$$\sum_{M' \,:\, M' \subseteq \overline{M}} t[M'] \leq n - m$$

where $\overline{M}$ is the complement of the set represented by M (and n is the sum of all values in the array t). Thus the problem of computing moderately safe bitmasks reduces to computing a sum over subsets (SOS) on t, a problem well known to be solvable in $O(2^k k)$ time.

Now we will compute all safe bitmasks, that is, bitmasks such that during at least m time units of the night, at least two guards from the bitmask watch the jewels. Now the condition to be checked is

$$\sum_{M' \,:\, |M' \cap M| \geq 2} t[M'] \geq m.$$

Let $t'[M] = \sum_{M' \subseteq M} t[M']$ be the SOS array of t. By going to the complement and using the inclusion-exclusion principle, we obtain an equivalent condition:

$$\left(\sum_{x \in M} t'[\overline{M} \cup \{x\}] \right) - (|M| - 1) t'[\overline{M}] \leq n - m.$$

It is checked for a bitmask M in $O(k)$ time. Overall, we obtain $O(c \log c + 2^k k)$ time complexity.

K – Multiset Variance

Author: Marek Sokołowski

If we have a multiset and extend it k times (for example, from $\{2, 3\}$ to $\{2, 3, 2, 3, 2, 3\}$, which is the extension with $k = 3$), its mean and variance do not change. Therefore replacing any input multiset by its k-extension does not affect the result (we can take each remaining multiset k times in the solution). Hence we may assume that all input multisets have equal length.

Given multisets $X_1, \ldots, X_n$, consider a linear combination $X = c_1X_1 + \ldots + c_nX_n$ with non-negative integer coefficients $c_1, \ldots, c_n$. Since the multisets have the same length, the mean of X is a linear combination of the means of X_i.

For the moment, think about random variables rather than multisets, and let the coefficients c_i be real non-negative values that sum to 1.

Recall that the variance can be written as $\mathrm{Var}[X] = \mathrm{E}[X^2] - \mathrm{E}[X]^2$. Thus if we fix $\mathrm{E}[X]$, we need to minimize or maximize $\mathrm{E}[X^2]$.

For each X_i we plot a point $P_i = (\mathrm{E}[X_i], \mathrm{E}[X_i^2])$ in the plane. Then the linear combination X corresponds to the point $P = c_1P_1 + \ldots + c_nP_n$. Therefore P lies inside the convex hull of $P_1, \ldots, P_n$. So to optimize for $\mathrm{Var}[X]$ means to find the best point in the convex hull.

Fixing $\mathrm{E}[X]$ and considering all points in this hull with that x-coordinate, the minimum and maximum y-coordinates are achieved on the boundary of the hull. Thus we only need to consider linear combinations of pairs of points that are connected by an edge of the hull.

To find the best linear combination $c_iX_i + c_jX_j$ of two random variables X_i, X_j corresponding to an edge, set $c_i = c$, $c_j = 1 - c$, and optimize over $c \in [0, 1]$. We obtain a quadratic in c; the coefficient of c^2 is negative, so the maximum is at the vertex (or at an endpoint of $[0, 1]$ if the vertex is outside the range). The minimum is at an endpoint, so for the minimum we need only one random variable.

From the formula we get a rational $c = r/q$. Thus we can return from random variables to multisets and take these extended multisets r and $q - r$ times, respectively. To get the answer, we multiply these values by the number of times we extended the multisets.

Time complexity is $O(n \log n)$ for computing the convex hull and iterating over its edges.