The 2024 ICPC Latin America Championship

Solution Sketches

Problem A – Almost Aligned

Author: Daniel Bossle, Brasil

Ternary search for the value t that results in the minimum bounding box area might be the most intuitive approach, but it is not correct. We are minimizing the product of two convex curves, which is not necessarily convex.

The area is:

$$(\max(X + V_X \cdot t) - \min(X + V_X \cdot t)) \cdot (\max(Y + V_Y \cdot t) - \min(Y + V_Y \cdot t))$$

These two dimensions follow a piecewise linear curve, which we can obtain with the Convex Hull Trick, and then analyze piece by piece.

Notice that the optimum value is always "in one extreme" (that is, it always occurs either at $t = 0$ or at a "changepoint" of one of the piecewise linear functions).

Total time is $O(N \log N)$.

Problem B – Beating the Record

Author: Agustín Santiago Gutiérrez, Argentina

Solution description

The expected solution is exhaustive search over the space of possible "global strategies", and thus the very small limit on N. A lot of the difficulty in this problem should be the correct modeling of what a "global strategy" for the speedrunning process is. It is fairly easy to assume false hypothesis, like all the following:

1. You will only ever restart the game when the probability to finish this run in under T seconds drops to zero.
2. We can do naive dynamic programming to solve the problem, and the state for that just needs to keep track of the current actual time in game (an integer) and in which of the N hard sections we currently are.
3. We can choose a fixed strategy for each of the N levels (including the different combinations of restarting the game depending on results) and then brute force all combinations.

These approaches are false in general. A counterexample to 1) is:

```
1 1000 1
50 900 900 1 500 800
```

Here, a single full playtrough of the game is guaranteed to beat the world record of $T = 1000$. The first strategy takes 901 seconds, and the second one takes 798 on average. However, if we use the second strategy and also take advantage of instantly restarting the game whenever we fail the risky strategy, we get an expected 99 fails which only take 1 second each, for an expected time of 600 seconds. Note crucially that the previous example depends on the value $S = 1$ being "small enough" that restarts are worth it, and in fact for $S = 4$ or larger it is better not to restart. These kind of effects is what breaks 2): such a DP approach is very tempting, because the state actually contains "enough information to make the optimal decision" (as it is quite intuitive that nothing other than current level and elapsed time is relevant). The problem is how to correctly compute that optimal decision, without looking at the "global strategy" (that is, without having solved the problem first).

For DP to work we need an order of iteration (i.e. no dependency cycles), and to use only values that have already been computed. This example shows that the decision of what to do at a certain game state is not oblivious to the input values and subproblem results of the previous levels. And of course, the decision also depends on later levels. Because of this, the author could not find any recursive formula for such simple state. However, there is a way to solve the problem very efficiently by fixing this issue, which we will explain in a later section.

The failure of assumption 3) points directly at what a "global strategy" is and is not. A counterexample to 3) is

```
2 1000 900
50 10 20 50 10 20
10 89 100 5 79 100
```

For the first level, there is actually just a fifty-fifty split between taking 10 or 20 seconds. However it takes 900 seconds of game to get to that point, so restarts are very expensive and are not a good idea while there is still even a 5 percent chance at the record.

At the second level, we want to go for the first strategy if the first level took only 10 seconds, and for the second strategy when the first level took 20 seconds (riskier, but necessary to get the record). So which strategy to choose for a level critically depends on the result of attempts at previous levels, as that changes the in-game time of arrival to the current level.

Thus, we see that a formalization of "global strategy" has to take into account these kind of "chained-ifs". For a fixed global strategy, ignoring resets for a while, a natural representation is a binary tree of possibilities: each node corresponds to a decision to take when arriving at a hard section, and thus would be labeled with a strategy to select at that point (either first or second strategy). The two children of each node then correspond to the possible branching of that run's "history", depending on whether the attempted strategy succeeded or failed. Then crucially, the global strategy can select different strategies on the same game level (which corresponds to all nodes at the same depth or "level" in the tree) depending on the actual path of history up to that level (as those different nodes can be labeled with different strategies). So for 4 levels, there are 15 nodes in this decision tree, and so we have 2^{15} possible strategies without resets. Finally, adding the possibility of resets into this representation is not too hard: a reset will always be done right after a decision is made, upon knowing the unfavorable result, and so corresponds to one of our previous 2^{15} trees but with some pruned branches.

We can actually count all such trees by the recursion:

$$\begin{aligned} f(n) &= 2(f(n-1)^2 + f(n-1)) \\ f(0) &= 1 \end{aligned}$$

The factor of 2 accounts for choosing first or second strategy at the root node, and then once that is fixed there are $f(n-1)$ ways to complete the tree if we choose to reset upon bad result on the root, or $f(n-1)^2$ ways if we choose to continue without resetting at the root. It is trivial to check that $f(4) = 21523360$ so only about 20 million strategies have to be checked. The actual general solution to the recursion is $f(n) = \frac{3^{2^n}-1}{2}$, which can be checked by induction but is unnecessary to solve the problem. Interestingly, it proves that the intended solution has complexity $O(3^{2^N})$.

The calculation of the expected time for a fixed strategy tree is an interesting exercise in probability. First, computing the probability of a run beating the record is relatively straightforward, as once the global strategy is fixed we can just track the probability of arriving at each node from the start and add up "winning nodes".

However, to compute the final expected value we will need to compute two values of conditional expectations for the time to traverse the tree (make a run of the game): an expected time given that the run is successful in beating the record, and an expected time given that the run fails. All of these values can be computed with some care by propagating cumulative probabilities and times from parents to children while branching in the tree, starting from a probability of 1 and S time invested in the root node.

Once we get the global expected time when success e_g , expected time when fail e_b and probability of success p, the expected time until winning is $e_g + (\frac{1}{p} - 1)e_b$.

For efficiency and possibly for ease of code, it is best to implement the process of generating strategies as a simple backtracking over a global vector of "nodes still waiting to be expanded", thus completely avoiding storing the actual tree. The starting state is the root node, and the backtracking always selects the last node in the vector and "branches" itself depending on the different strategies to choose for that node. For each of those strategies, the corresponding node is deleted from vector and new nodes are generated for the children nodes and pushed into vector: either 0 new nodes (for leaf nodes at level N), 1 (for a choice of strategy that restarts on a bad outcome), or 2 new nodes (for a strategy that will not restart at that node).

Note that this way during the backtracking, there are actually 4 choices of "strategy" in each node: we can choose either the first or second strategy of the level, and then we can choose either not to restart, or to restart if unsuccessful. If one does not notice that restarting upon successful attempt makes no sense (one should have restarted earlier in that case), these are 6 choices.

Finally, note that $N = 5$ is already too much for this approach, as there are about 3^{30} strategies.

About numerical stability

If using floating point, care should be put to the possibility of numerical error. All involved computations should be numerically stable with typical implementations, except for the subtraction in the $(\frac{1}{p} - 1)$ factor of the final formula. This subtraction carries a real risk of catastrophic cancellation and a significant loss of precision in a case where p is very close to 1. However, in all implementations we tried for this particular problem, it can be proved that double precision turns out to be enough even when performing this $1-p$ computation (although not by a large margin).

This is easily avoided if the code also computes the probability of fail $f = 1 - p$ by directly accumulating (adding positive probabilities) along the tree, in exactly the same way that we described p can be computed. Then the formula becomes $\frac{1}{p}-1 = \frac{f}{p}$, and division is a numerically stable operation. Exponent underflow (in probabilities) or overflow (in the final expected value) does not seem to be a serious concern, as $N = 4$ means that the lowest positive probabilities involved are around 10^{-8}. Perhaps the best alternative is to just work with integers until the very final formula $e_g+(\frac{1}{p}-1)e_b$, so that the subtraction can be made in an exact way. This can

be done easily because the probability calculations of e_g, e_b, p, f values only leaves integers due to the probability $\frac{1}{100}$ factors when adjusting the input percentages. We can get rid of those factors by "multiplying" the real values by 10^8 and just doing all computations in integers. This is equivalent to using a decimal, fixed point representation with 8 digits after decimal dot, which is enough to exactly represent all values up until the division by p in the very final formula.

An efficient solution

Even though it is not the intended solution, the problem can be solved with dynamic programming + a binary search trick.

Let $dp(i, t)$ be the expected remaining time when playing optimally, given that we are currently at level i and have already used t seconds in this run of the game. Note that in our problem there are $O(2^N)$ possible values of t. Our desired final answer is $E = dp(0, S)$.

It is possible to express $dp(i, t)$ based on the $dp(i + 1, t')$ values and, due to resets, the $E = dp(0, S)$ value. Crucially, the only dependency that goes "backwards" in time for the recursion is the dependency on E. Thus, if we fix the value of E by guessing it, it is possible to solve for all of the other $dp(i, t)$ values, including the $dp(0, S)$ itself, by using the recursive dp formula replacing any mention of $dp(0, S)$ by the guessed value of E.

The key property (the proof of which is left as an exercise to the reader) is that if our guess for E was too high, then the computed value $dp(0, S)$ will be lower than E. And conversely, if the guess E was too low, then the computed value $dp(0, S)$ will be higher than E. Thus we can continue guessing by using binary search to get the correct value of E to the desired precision.

The complexity is either $O(BNT)$ or $O(B2^N)$ if only the reachable states are used, where B is the number of steps of the binary search.

Problem C – Clever Cell Choices

Author: Miguel Angel Ortiz Merida, Bolivia

Analyzing the game

If there is a perfect matching in the graph formed by the empty cells, then the second player wins. Otherwise, the first player wins.

Let's consider the scenario where we have found a perfect matching in the grid. In this case, no matter which empty cell the first player selects, the second player can always place a stone on the cell matched to the one chosen by the first player. This guarantees a win for the second player because every empty cell is matched in the perfect matching, ensuring that there is always a valid move available.

If there is no perfect matching, we find a maximum matching in the graph. Then, the first player selects an empty cell that is not matched. Subsequently, the first player adopts the strategy of always choosing an empty cell matched to the one selected by the second player. This strategy ensures that the first player can always make a move and ultimately win the game. This is because if the second player ever chooses an empty cell that is not matched during the game, it implies the existence of a path starting at an unmatched cell, passing through zero or more pairs of matched cells, and ending at an unmatched cell. In such a case, we could increase the cardinality of the matching by shifting the matching in one direction along the path, indicating that the matching found initially was not of maximum cardinality.

Counting winning starting cells

Following from the analysis above, winning starting cells are the empty cells that do not belong to every maximum matching. A cell belongs to every maximum matching if max matching in G and "*G without that cell*" differ.

Given the input constraints ($1 \leq N, M \leq 50$) we could simply run $O(NM)$ matchings using an efficient algorithm such as Hopcroft-Karp / Dinitz for a total complexity of $O(N^2M^2\sqrt{NM})$.

Faster approach

A slightly faster approach is running one BFS per empty cell after calculating an initial matching.

After the first flow/matching, we work on a max-matching residual network to test nodes one by one. If the node under consideration is unmatched, then it does not belong to every matching. Otherwise, we try to modify the flow/matching by passing flow against that matching edge through a cycle. We should find such a path if and only if there is a max matching without that node.

For this approach, the final complexity with an efficient matching algorithm is $O(N^2M^2)$.

Even faster approach

Another way to find the cells that do not belong to every maximum matching is by running two max flows.

We can find the "leftmost" and "rightmost" cuts, which are those vertices "reachable from the source" and "that reach the sink" in the final residual network in terms of flow. This information tells us which nodes are always in some side of the matching.

This can be proved by looking at the symmetric difference of current matching and a hypothetical matching not containing a certain vertex.

With this approach, the final complexity reduces to $O(NM\sqrt{NM})$.

Problem D – DiviDuelo

Author: Pablo Blanc, Argentina

The answer depends on the factorization of N. We consider different cases:

- If $N = p^\alpha$: All the divisors are multiples of p except for 1. Then the starting player wins except if he is forced to pick 1 as his last pick. This will happen only if the number of divisors is odd (α even).
- If $N = p^\alpha q^\beta$: We distinguish between the cases $N = pq$ (with $\alpha = \beta = 1$) and the remaining situations.
  - If $N = pq$: The starting player selects pq in the first turn. Then in his second (and last) turn picks p or q, he has secured that the GCD is not 1.
  - Otherwise: If $N = p^\alpha q^\beta$ and $N \neq pq$, we assume $\alpha \geq 2$. The second player has a winning strategy.
    - If the total number of divisor is odd the second player can force the starting player to pick 1.

  - Otherwise, observe that the list of divisors contains the numbers $1, q, p$, and p^2. The second player can force the starting player to pick one number from the set $\{1, q\}$ and another from the set $\{p, p^2\}$, and therefore the GCD of the numbers of the starting player will be 1.
    To force the starting player, the second player will not pick a number from those sets until the starting player picks one of the numbers, then the second player picks the other. The starting player will be forced to pick one number in those sets before the other player because the total number of divisors is even. This way the starting player will end up picking one number from each set.

- If N has at least 3 different prime divisors: The second player has a winning strategy. Let p, q, and r be distinct prime factors of N. Then, as in the previous case, the second player can force the starting player to pick one number from the set $\{1, p\}$ and another from the set $\{q, r\}$, thereby securing a victory.

We conclude that the starting player has a winning strategy iff $N = p^\alpha$ with α odd or $N = pq$. Finally, the problem can be solved by factoring the number in $O(\sqrt{N})$.

Problem E – Expanding STACKS!

Author: Giovanna Kobus Conrado, Brasil

Two customers a and b (where a enters the restaurant first) can have been in the same line if and only if their relative order in the input is:

```
+a -a +b -b
```

or:

```
+a +b -b -a
```

as opposed to:

```
+a +b -a -b
```

A set of customers can have been in the same line if pairwise they could have been in the same line.

It is sufficient to view the customers as vertices, quadratically iterate through the pairs of customers, and add an edge between them if they cannot be in the same line.

Now an assignment of customers to lines is a coloring of such a graph, thus the problem boils down to determining whether the graph is bipartite and assigning the customers to lines according to their color.

The complexity is $O(N^2)$, as that will be the maximum number of edges in the generated graph.

A faster approach

The problem can also be solved in $O(N)$.

An *item* is a chronologically ordered list of integers x_i, indicating a sequence of "push x_i" events that go all into the same stack in that order. Assume that two items can be concatenated in $O(1)$ and accessed like a stack (such as using a linked list).

A *Node* is a pair of items, such that both items must go into different stacks. Items in a Node might be empty. A Node with one empty item is a *Simple Node*, representing a sequence of pushes that must all be together on the same stack, but it can be any of the two stacks. A Node with two empty items is itself empty and irrelevant, and can be deleted at any time.

A *NodeStack* is a stack of Nodes. A NodeStack represents a set of many possible specific states of a two-stack system: a consistent two-stack system is obtained by choosing for each Node, precisely to which stack goes each of its items, then once that is chosen each stack contains all the corresponding items concatenated in the NodeStack order.

We can actually keep the NodeStack dynamically when adding push / pop events: a push event is easy, we can simply push a `([], [push x])` Node into the NodeStack. When a `pop x` event appears, the corresponding `push x` event must be the top element (next to be popped) in one item of the NodeStack. Additionally, all Nodes from the top of the NodeStack up to the Node containing the `push x` event must be Simple Nodes (otherwise, no consistent two-stack configuration has the `push x` event on the top of a stack, so it cannot be popped). Then we pop the old `push x` event from its item, and concatenate all the Simple Nodes that were on the stack on top of the `push x` Node, into the other item of that Node. That means: all of those push events that occurred after `push x` must have occurred in "the other" stack, otherwise `push x` cannot be on top and `pop x` would be impossible.

Problem F – Fair Distribution

Author: Victor de Sousa Lamarca, Brasil

To determine whether a fair distribution of blueprints is possible, we first need to consider all possible total height differences that can arise due to the ground floors. We need to consider every way to partition the blueprints into two non-empty sets, denoted by A and B. The total height difference due to ground floors for such a partition would be the difference between the sum of ground floor heights in set A ($\sum_{i \in A} G_i$) and the sum of ground floor heights in set B ($\sum_{i \in B} G_i$).

These unavoidable differences due to ground floors represent the contributions to the overall height difference between Alice and Bob's buildings. Once the blueprints are assigned, these differences are fixed, and the number of residential floors chosen will need to compensate for them.

The key question is: which kinds of total height differences are achievable depending on how the number of residential floors is set?

It turns out that a fair distribution is possible if and only if the GCD of the heights of residential floors divides any of the previously listed height differences. This comes from Bezout's identity, or alternatively from understanding of Euclid's algorithm and some math on modular arithmetic.

All that's left is efficiently listing all possible total height differences due to ground floors. We exploit the fact that the sum of the heights of the ground floors of all blueprints is at most $O(N)$, where N is the number of blueprints. This implies that the number of distinct ground floor heights is $O(\sqrt{N})$, allowing us to optimize the knapsack algorithm. By adapting the knapsack algorithm, we can achieve a final complexity of $O(N\sqrt{N})$. More details on optimizing the knapsack algorithm can be found in page 254 of `https://cses.fi/book/book.pdf` or `https://codeforces.com/blog/entry/59606`.

Problem G – Greek Casino

Author: Célio Passos, Brasil

The DP recurrence

We should first ensure that the expected value is indeed finite (this also implies that the process ends in a finite number of steps with probability 1). Let X denote the random variable indicating how many coins are awarded and let p_y denote the probability that number y is sampled (that is, $p_y = W_y / (\sum_i W_i)$). We can think of the process as a random walk over the numbers $1, 2, \ldots, N$. A path ending on a can only include divisors of a itself. Thus:

$$\begin{aligned}
\mathbb{E}(X) &= \sum_{k \geq 1} \mathbb{P}(X \geq k) \\
&\leq \sum_{k \geq 1} \sum_{a=1}^{N} \left(\sum_{b \text{ divides } a} p_b \right)^k \\
&\leq N \sum_{k \geq 1} \left(1 - \min_a p_a\right)^k \\
&< \infty
\end{aligned}$$

where we are summing over all possible last numbers a of a path length of k.

Now let's define

$$E_a = \sum_P \mathbb{P}(P)$$

where the summation goes over all paths P ending on a. Note that E_a is exactly the expected number of times the token will go to (or stay at) slot a, therefore

$$\mathbb{E}(X) = \sum_{a=1}^{N} E_a - 1.$$

We need to subtract 1 here because the contestant doesn't get a coin in the beginning when the token is at slot 1. Also,

$$E_c = \sum_{a,b:\text{LCM}(a,b)=c} p_b E_a$$

for $c > 1$ and $E_1 = 1 + p_1 E_1$. We can rewrite this as $E = p * E + \mathbf{1}$, where $*$ denotes the LCM convolution, and rearrange it:

$$(\mathbf{1} - p) * E = \mathbf{1}$$

The number $\mathbf{1}$ here is actually the array $(1, 0, 0, \ldots, 0)$, which acts as the multiplicative identity.

LCM convolution technique

Let's denote by T the transformation (on arrays) defined by

$$T(A)_k = \sum_{d \text{ divides } k} A_d$$

It is easy to compute its inverse (writing a formula may be complicated, but given $T(A)$ we can easily compute the original value A). Now, with the fact that a and b both divide c if and only if $LCM(a, b)$ divides c, we can write

$$T(A) \odot T(B) = T(A * B)$$

where $\odot$ denotes pointwise multiplication. Thus, given $T(A * B)$ (non-zero in our case) and $T(A)$, we can easily compute $T(B)$ and, inverting the transformation, also B itself.

Final solution

Just let $A = \mathbf{1} - p$ and $B = E$ and apply the previous facts. The complexity is $O(N \log N)$.

Alternative approach

There's also a more elementary approach. First, compute the sorted list of divisors of all numbers up to N using a standard $O(N \log N)$ sieve.

For each number x, $1 \leq x \leq N$, we can compute the probability that the token reaches slot number x and the expected amount of coins awarded until then (given that it reaches x).

For the transition from x to y (y is a multiple of x), look at the factorization of x and y. When going from x to y, some exponents grow (possibly from 0 to a positive value for "new" primes), while others remain the same. Those that grow must be present in z for $\text{LCM}(x, z) = y$. Those that stay the same must be lower or equal in z than in x. So that means $z = f \cdot d$, where f is "a fixed integer composed of the power primes with larger exponent in y than in x", and d is any divisor of "the other primes" in x, which is itself a divisor of x and thus a number at most x that has precomputed list of divisors by our $N \log N$ sieve.

The number of possible z are the number of divisors d such that $f \cdot d \leq N$, and the sum of their weights defines the transition probability.

If we consider all the triples (x, y, d) that we iterate, all of them are different. This means that the number of iterations is bound by the number of triples (d, x, y) such that d divides x and x divides y (and all these numbers are $\leq N$). There are $O(N \log^2 N)$ triples in total and that is the complexity of this approach.

$$\begin{aligned}
\sum_{d=1} \sum_{x=ad \leq N} \sum_{y=bad \leq N} 1 &\leq \sum_{d=1}^{N} \sum_{ad \leq N} \frac{N}{ad} \\
&= N \sum_{d=1}^{N} \frac{1}{d} \sum_{a \leq \frac{N}{d}} \frac{1}{a} \\
&= N \sum_{d=1}^{N} \frac{1}{d} O(\log \frac{N}{d}) \\
&= O(N \log^2 N)
\end{aligned}$$

Problem H – Harmonic Operations

Author: Agustín Santiago Gutiérrez, Argentina

First of all we study the composition of allowed transformations.

We can convert $L(t, D)$ transformations into equivalent $R(t, D')$ transformations by utilizing the fact that $L(t, D) = R(t, |t| - D)$.

When composing a series of operations like the ones given, the resulting total transformation is either a single rotation, or a single inversion followed by a single rotation. The rule for composing two of these total transformations is quite simple and easy to find and check by examples:

- $R(t, a) = t'$ followed by $R(t', b)$ is $R(t, a + b)$

- $I(t) = t'$ followed by $I(t')$ is the identity, "no transformation" or simply $R(t, 0)$
- $R(t, a) = t'$ followed by $I(t')$ is the same as $I(t) = t''$ followed by $R(t'', |t| - a)$
- $I(t) = t'$ followed by $R(t', a)$ simply stays like that and is a possible total transformation

The rule for the inverse transformation is also simple: the inverse of $R(t, a)$ is $R(t, |t| - a)$ and any transformation of the form $I(t) = t'$ followed by $R(t', a)$ is its own inverse.

These properties define the dihedral group: `https://en.wikipedia.org/wiki/Dihedral_group`. Previous knowledge of group theory, more specifically of the dihedral group itself, can help in finding this problem's solution faster, but such special knowledge is not a prerequisite for solving the problem as no advanced group-theoretic result is used, only the structure of the particular dihedral group which is quite ad hoc and can be discovered by doing examples and reasoning "how much information must I keep", so not specially harder or different than very similar techniques used in segment tree or other competitive programming problems where a range of items is aggregated.

After all that, something to notice is what happens if string S is a power $S = A^k$, that is, S is formed by concatenating $k \geq 2$ identical copies of string A. Then, for any transformation f, the resulting new string after applying f will still be k copies of a single string, and that string is $f(A)$. That is, $f(S) = f(A)^k$.

Thus for $S = A^k$, a sublist of transformations leaves S fixed if and only if it leaves A fixed. We can then start by computing the largest k such that we can write $S = A^k$ (possibly $k = 1$), which is a standard and classical problem efficiently solvable using KMP or other string techniques. Then by the previous analysis the test case answer does not change if we assume input is simply A instead of S, so we have reduced the problem to the case where the string is not a power (i.e. we can only write $S = A^k$ if $k = 1$). Note that this effectively shrinks the actual $|t|$ that we will use when composing transformations and computing inverse transformations.

The key thing one might conjecture is that once we have a string of length N that is not a power, then all the $2N$ possible transformations actually give different strings when applied to S. This is clearly true for the N rotations, as if two different rotations give the same string it is easily proved that the string is $S = A^k$ with $k > 1$. But it might fail for the inversions: there might be some $I(S)$ followed by $R(S, a)$ that leaves S fixed. This never happens for a string such as "`cosa`", but it does happen for a string such as "`casa`": even though all its rotations are different, $I($"`casa`"$)$ = "`asac`" followed by $R($"`asac`"$, 1)$ = "`casa`" does not change the string. The critical property of "`casa`" is that its inversion "`asac`" is identical to one of its rotations, in this case $R($"`casa`"$, 3)$.

Critically, there can never be two distinct values a, b such that both $I(S) = S'$ followed by $R(S', a)$ and $I(S) = S'$ followed by $R(S', b)$ fix S. If it were the case, applying one after the other would also not change S, but doing so produces a non-trivial rotation, and since the string was not a power, this is a contradiction. So after initially reducing the string to be a non-power, there will be at most two total transformations that fix it: the trivial transformation $R(S, 0)$ which always works, and possibly one transformation of the form $I(S) = S'$ followed by $R(S', a)$, precisely when one of the rotations of the string matches its reverse. This can also be checked efficiently by duplicating the string and then using any standard string matching algorithm like KMP.

Once we have this structure understood, we need to count the number of pairs. A pair i, j will work only if performing the composition (in order) of all the transformations in the $[i, j]$ range gives one of the (at most two) working transformations as a total result. The number of pairs can be counted in linear time using an histogram and the same idea as "prefix-sum": if we compute the net transformation of every prefix $[0, i)$ in $P(i)$, then for a range $[i, j)$ the net transformation in range is simply the inverse of $P(i)$ followed by $P(j)$. So for a fixed value

of either i or j (lets say j), it is simple to identify which are the one or two possible total transformations that work as $P(i)$, and if we are processing left to right (or right to left if we are fixing i) we can have already counted in an histogram array the total number of previous indexes $i < j$ such that $P(i)$ equals any specific value we desire. So adding all of these counts together for all values of j will give the final answer.

Using efficient string algorithms like KMP, the final time complexity is linear in input size $O(|S| + K)$.

Note that if we don't start by reducing the string to a non-power, many rotations and transformation work. The set of precisely which of the $2N$ transformations work can be computed efficiently with the same string algorithms: we need to know which rotations match the string or its reverse, which is practically the same info we compute when reducing to non-power. However, in the last counting step, for each j we would have to check a linear number of relevant $P(i)$ values in the histogram, leading to quadratic complexity. The reduction ensures that we just have to check two candidates at most.

Problem I – Insects, Mathematics, Accuracy, and Efficiency

Author: Giovanna Kobus Conrado, Brasil

Cleaning up the input

The first step here is to clean up the input by considering only the points that are vertices of the convex hull. Since the convex hull is the smallest convex polygon containing all the points, any point inside the convex hull can be ignored.

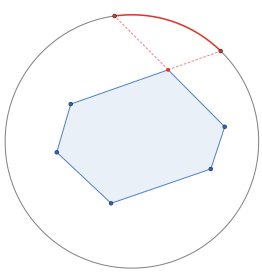
After this cleanup, the number of points that we need to consider is reduced. Intuitively, the resulting number might be considerably lower than the original number of points in input. One way to come up with a bound is through experimentation. For example, counting the number of vertices on the convex hull of all integer points in a circle with a radius of $R = 10000$. This would shows that for our biggest circle the convex hull has ≈ 1600 vertices. Alternatively, a bound of $O(R^{\frac{2}{3}})$ vertices can be derived with more work, as discussed in `https://codeforces.com/blog/entry/62183`.

Note: when computing the convex hull make sure to remove collinear points (otherwise, the hull can potentially keep all the points from the input).

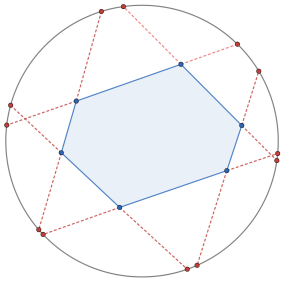
Solving the problem

We need to add a point p to a set of points to maximize the convex hull while being constrained by a circle. It's evident that the point maximizing the area of the new convex hull will always lie on the border of the circle.

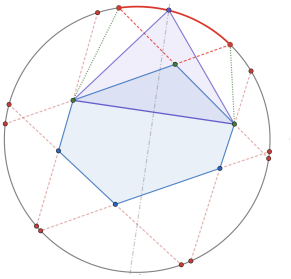
For each vertex v of the original convex hull, consider the segment of the circle in which adding the new point p excludes v from the new convex hull. This segment can be found by intersecting the circle with the lines extending from the edges of the polygon adjacent to v.



By performing this process for every vertex, we partition the circle into $2N$ segments, each representing a unique set of vertices covered if p is added in that segment.



We solve the problem for each segment individually. Finding the point that maximizes the area of the convex hull is equivalent to maximizing the area of the triangle defined by the last and first uncovered vertices. As the base of the triangle is fixed, our goal is to maximize the height, which corresponds to finding the intersection points between the circle and the line perpendicular to the base of the triangle. All such points can be found through straightforward primitives.



For each segment, we find the candidate point, generate the convex hull for the new set of points, and calculate the area, resulting in an $O(N^2 \log N)$ approach.

A numerically stabler approach

The previous approach involves computing the intersection points of lines with the circle, as well as using those intersection points to figure out the first/last uncovered vertices. Ensuring numerical stability for this approach might be challenging.

Alternatively we can try all pairs of "diagonals" in the convex hull. All the N^2 "cut areas" can be computed efficiently and fully in integers, as the original points all have integer coordinates. The answer will be the maximum of trying to add to these areas, a single triangle which has to be computed in floating point.

The key insight here is realizing that we don't need to repeatedly compute the area of the convex hull. Instead, the area can be updated in $O(1)$ when adding/removing a single vertex. This means that as we iterate over all pairs of diagonals, we can keep a "running area" and add to it the area of the triangle formed by the diagonal and the point perpendicular to it in the circle. This results in an $O(N^2)$ approach.

Since such triangles might represent the entire solution (e.g.: for a case with $N = 2$, where the two integer vertices of the triangle are given as input), the overall error of the solution depends on the accuracy of computing this single triangle area. Skinny triangles are numerically ill conditioned, but for any triangle with coordinates up to 10000, absolute error in computation should be at most 10^{-7} for double and 10^{-11} for long double. Since minimum answer for any case $N \geq 2$ is $\frac{10^4}{2}$, that is a maximum relative error of 10^{-11} for double and 10^{-15} for long double.

A faster approach

Notice that combining the two insights above we can solve the problem in $O(N \log N)$ time.

If instead of iterating on the N^2 diagonals we use the $2N$ critical segments defined in the first approach and update a "running area" that starts with the area of the original convex hull and is updated in $O(1)$ as we adjust first/last uncovered vertices while iterating on the segments. The complexity of the solution is then dominated by the cost of convex hull calculation and sorting the critical points.

Again, ensuring numerical stability for this approach might be challenging.

Problem J – Joys of Trading

Author: Agustín Santiago Gutiérrez, Argentina

This problem is a greedy sorting problem that invites a lot of failed greedy attempts.

If there were no restrictions on the total number of hours invested by each village, then obviously the global minimum is reached if for each resource, we make the village that is faster at that resource produce the whole of it, in sufficient amount for both villages.

This invites one to greedily try something like that but only while there are enough person-hours available, and finally completing all remaining work with the other tribe. Unfortunately, it is not the case in general that a resource should be done by the village that does it faster.

The correct greedy is to sort resources by $\frac{A_i}{B_i}$. Then, it is the case that there will be some cutting point in this sorted array of resources such that village A produces the prefix before the cut, and village B produces the suffix after the cut. Note that the cut might be placed at any continuous position and thus might be "inside some resource", as happens in the first example where the cut is "0,5 units into food".

The proof is by reductio ad absurdum: take a solution that is "as close as possible" to having A produce a prefix and B a suffix in the sorted array. Now assume that in this sorted array, there are two different resources having indexes $i < j$, and such that B produces a quantity $x > 0$ of resource i, while A produces a quantity $y > 0$ of resource j. If we change village B so that it uses $\epsilon \cdot B_i$ less person-hours on resource i, and redirect those person-hours so that B uses them to produce resource j instead, we will get $J = \frac{\epsilon \cdot B_i}{B_j}$ units of resource j, while losing ϵ of resource i.

Because of the sort criteria we have $\frac{A_i}{B_i} \leq \frac{A_j}{B_j}$, that is $\frac{B_i}{B_j} \geq \frac{A_i}{A_j}$, and from this $J \geq \epsilon \cdot \frac{A_i}{A_j}$, so we get more than $\epsilon \cdot \frac{A_i}{A_j}$ units of resource j, a quantity that takes village A a total of $\epsilon \cdot A_i$ person-hours to produce. If we now redirect those $\epsilon \cdot A_i$ person-hours of village A to produce resource i instead, we get ϵ units of resource i.

For small enough ϵ, we are sure to be able to make all of these changes and get a valid solution. So, after these changes, we end up with no less resources, while using exactly the same person-hours in each village, and we move closer to having all work of A being a prefix and B being a suffix, which is absurd because we chose one that was as close as possible.

Once we have that result established, it is possible to try all N resources to see if the cut is in that resource. When moving from cut i to $i+1$ it is simple to update the total cost of A producing every resource in $[0, i)$ and the total cost of B producing every resource in $[i+1, N)$. Then only resource i remains, and a certain amount of available person-hours for each of A and B is known, so this resource being the only left can be greedily solved: use the village that produces it more efficiently, until the whole resource is produced or available person-hours run out, and then switch to the other one.

That way, all cuts can be tried and the cost for all of them computed in $O(N \log N)$ time, so that the best one is finally returned. The implementation is very simple and the most difficult part of the problem is probably coming up with the correct sort among many possible incorrect greedy alternatives.

An interesting corollary of this problem is that specializing and trading is efficient: an optimal solution always exists where there is at most one resource that is produced by both villages.

Problem K – KMOP

Author: Alejandro Strejilevich de Loma, Argentina

BFS/DP solution

The problem can be modeled as a shortest path problem in a graph. Each node/state is defined by three parameters w, ℓ and c, indicating that there is an acronym for the first w words that ends in the ℓ-th letter of the w-th word, terminating with c contiguous consonants. Parameter w ranges from 0 to N, ℓ ranges from 1 to 3, and c ranges from 0 to 2. Each node (w, ℓ, c) has at most two direct successors: $(w, \ell + 1, c_1)$ (include the next letter of the current word), and $(w + 1, 1, c_2)$ (move to the next word, skipping the rest of the current word). Value c_i is either $c + 1$ or 0, depending on whether the target letter is a consonant or a vowel, respectively. The solution is the length of a shortest path from node $(0, 1, 0)$ to node $(N, 1, c)$ (taking minimum over c). Since the graph is an unweighted DAG, the shortest paths can be computed using BFS. There are $O(N)$ nodes and each node has outdegree at most two, so there are $O(N)$ edges; thus the time complexity of the algorithm is $O(N)$. There is no need to explicitly build the graph.

An equivalent solution using dynamic programming defines the function $f(w, \ell, c)$ as the minimum length an acronym described by the state (w, ℓ, c) can have. The required answer is $\min_c f(N, 1, c)$. Each value of the function can be computed in $O(1)$, and then the time complexity is $O(N)$.

Greedy solution

The idea of the greedy algorithm is starting with the shortest possible acronym and then adding letters to make it pronounceable. Let $\ell_{i,j}$ be the j-th letter of the i-th word. Start with the acronym $A = \ell_{1,1}, \ell_{2,1}, \ldots, \ell_{N,1}$. While A is not pronounceable do the following.

1. Locate in A the first three contiguous consonants $T = \ell_{k,1}\ell_{k+1,1}\ell_{k+2,1}$ (as we will see, these letters are always the first letters of three contiguous words).

2. If $\ell_{k+1,2}$ is a vowel, insert it in A ($\ell_{k,1}\ell_{k+1,1}\ell_{k+2,1} \to \ell_{k,1}\ell_{k+1,1}\ell_{\mathbf{k+1,2}}\ell_{k+2,1}$). Notice that this is the best option since it breaks the triplet T and possibly another triplet starting at $\ell_{k+1,1}$. Thus, if A is still not pronounceable, the first problematic triplet must start at $\ell_{k+2,1}$ or later.

3. If $\ell_{k+1,2}$ is a not a vowel (it is a consonant or does not exist), T should not be broken by inserting letters between $\ell_{k+1,1}$ and $\ell_{k+2,1}$: if $\ell_{k+1,2}$ does not exist it cannot be inserted, while if it is actually a consonant there is no gain in inserting it. This is because inserting $\ell_{k+1,2}$ would generate another problematic triplet starting where T starts. This new problematic triplet should be broken by inserting additional letters between $\ell_{k,1}$ and $\ell_{k+1,1}$, but this additional letters would also break T.

(a) If $\ell_{k,2}$ is a vowel, insert it in A ($\ell_{k,1}\ell_{k+1,1}\ell_{k+2,1} \to \ell_{k,1}\ell_{\mathbf{k,2}}\ell_{k+1,1}\ell_{k+2,1}$). If A is still not pronounceable, the first problematic triplet must start at $\ell_{k+1,1}$ or later.

(b) If $\ell_{k,2}$ is a consonant and $\ell_{k,3}$ is a vowel, insert them in A ($\ell_{k,1}\ell_{k+1,1}\ell_{k+2,1} \to \ell_{k,1}\ell_{\mathbf{k,2}}\ell_{\mathbf{k,3}}\ell_{k+1,1}\ell_{k+2,1}$). Again, if A is still not pronounceable, the first problematic triplet must start at $\ell_{k+1,1}$ or later.

(c) If $\ell_{k,2}$ is a consonant and $\ell_{k,3}$ is not a vowel (it is a consonant or does not exist), the triplet T cannot be broken (without generating another problematic triplet which would be unbreakable). The same situation occurs if $\ell_{k,2}$ does not exist. In this cases there is no pronounceable acronym, and we are done.

There is no need to explicitly initialize and update A; it is enough to maintain its length. Locating all the problematic triplets can be done in $O(N)$ since they start at increasing values of k. Each time a problematic triplet is found, updating the length of A can be done in constant time. Thus, the time complexity of the algorithm is $O(N)$. The python code below implements this approach.

```python
#!/usr/bin/env python3

import sys

def IsVowel(c):
    return c in 'AEIOUY'

L=N=int(sys.stdin.readline())

Consonants=qTail=0
qWords=[None]*3

while True:
    while Consonants<3:
        if N:
            N-=1
            qTail=(qTail+1)%3
            LastWord=sys.stdin.readline()[:3]
            qWords[qTail]=LastWord+'X' # length is at least 3 because of EOL
            Consonants= 0 if IsVowel(LastWord[0]) else Consonants+1
        else:
            print(L)
            sys.exit(0)
    MiddleWord=qWords[(qTail+2)%3]
    if IsVowel(MiddleWord[1]):
        L+=1
        Consonants=1
    else:
        FirstWord=qWords[(qTail+1)%3]
        if IsVowel(FirstWord[1]):
            L+=1
        elif IsVowel(FirstWord[2]):
            L+=2
        else:
            print('*')
            sys.exit(0)
        Consonants=2
```

Problem L – LED Matrix

Author: Alejandro Strejilevich de Loma, Argentina

The solution is ad hoc. Let m be a row of the matrix and let p be the corresponding row of the pattern. m properly displays p if and only if all the LEDs in m are good or all the LEDs in p are off. This can be checked in $O(C + K)$. The whole matrix properly displays the whole pattern if and only if this happens for every row. The cost of the algorithm is $O(R \times (C + K))$.