Maratona de Programação da SBC 2020

This problem set is used in simultaneous contests:
Maratona de Programação da SBC 2020
Segunda Fecha Gran Premio de México 2020
Primera Fecha Gran Premio de Centroamérica 2020
Torneo Argentino de Programación 2020

November 14th, 2020

Editorial

This editorial was prepared by the following volunteers:

- Agustín Santiago Gutiérrez
- Fernando Fonseca Andrade Oliveira
- Naum Azeredo Fernandes Barreira
- Teodoro Freund

Problem A
Sticker Album

As in most expected value problems, it's useful to name the expected value of the number of packets one must buy to fill an album that has x empty slots left: let this value be E_x. We have $E_0 = 0$, as it is not necessary to buy any packet to fill a full album.

As the number of cards per packet is uniformly random, the probability of a packet containing any number of cards between A and B is the same, $\frac{1}{B-A+1}$. To make the next expressions simpler, let's denote $L = B - A + 1$.

If the album needs x more cards, after buying a packet, with probability $\frac{1}{L}$ the album will need $x - A$ cards, with the same probability it's going to need $x - A - 1$, and so on. Therefore we can write E_x as a function of other values of E:

$$E_x = 1 + \frac{1}{L}E_{x-A} + \frac{1}{L}E_{x-A-1} + \cdots + \frac{1}{L}E_{x-B}$$

Using the equation above to calculate $E_1, E_2, \cdots, E_N$ allows us to solve the problem in $O(N(B - A))$, which is too slow. To make it faster, we can use a sliding window of sums of $B-A+1$ consecutive values of E to get the value of the right side sum quickly. Alternatively, we can also calculate prefix sums of E to get the sum in constant time, but we need to be careful with double precision issues when using this approach. Both approaches make the final solution work in $O(N)$.

A detail is that A can be zero, and in this case in the equation above E_x would depend on itself. This is not a big problem, because we can still solve the equation for E_x:

$$E_x = 1 + \frac{1}{L}E_x + \frac{1}{L}E_{x-1} + \cdots + \frac{1}{L}E_{x-B}$$

$$E_x = \frac{L}{L-1}\left(1 + \frac{1}{L}E_{x-1} + \cdots + \frac{1}{L}E_{x-B}\right)$$

Problem B

Battleship

For this problem, it is enough to place each ship in the board, one by one, and check if each ship can be placed according to the conditions in the statement.

At the start, we can represent an empty board by a 10x10 two-dimensional array filled with zeros, indicating there are no occupied squares. Whenever a new ship is placed, we can use a for or while loop to iterate over all squares that the ship occupies, and mark all of those squares in the matrix with a 1, indicating that those squares are now full. If at any point we try to place a ship in a square that was already full, or a square that is outside the 10x10 board, we report that the configuration is not good. Otherwise, the configuration is good.

Problem C
Concatenating Teams

The key observation is to concentrate, within university A, on the triplets (x, x', S) such that $x = x'S$ and both x and x' are team names from university A.

Similarly, we can consider the triplets (z, z', S) such that $z' = Sz$ and both z and z' are team names from university B.

A certain name X is not peculiar, exactly when there exists such a triplet (x, x', S) for university A and such (z, z', S) for university B, both having the same S and such that X is one of the 4 different teams involved in this pair of triplets.

One might think at first that there can be too many triplets to explicitly generate them all, but this is not so: For each x for example, there is at most one triplet (x, x', S) for each prefix of x, and so the total number of triplets is at most the total number of characters in the input strings.

Hashing seems to be the easiest way to efficiently generate and group the triplets: Iterate all prefixes of all the words in the first set. If a prefix of a word is found to be some other word from the set (which can be efficiently checked by putting all hashes in a set/map), the remaining suffix (whose hash can also be efficiently computed) is the S involved in the triplet. The same can be done for set B, and then all triplets can be grouped by S and iterated. Once grouped, for each S such that at least one triplet with S exist for each university, then **all** of the x, x', z, z' involved in any such triplet can be marked as nonpeculiar. Those names remaining unmarked after all of this are precisely the peculiar names.

Using a simple trie, all the triplets can be generated in the same time, but while x and x' can be stored explicitly, S will be computed as a pair of starting and ending indices i, j. To then group the triplets having identical S from this representation is not quite simple.

There is that detail that, while there are $O(input)$ different S, and they can be identified and grouped by equality using hashing, the total sum of their lengths can be very big, so explicitly creating a trie or similar with all of them can be too large.

This solution runs in a linear number of map operations.

By creating a suffix array / suffix tree of the words and working with it carefully, a similar fully deterministic efficient solution can be written, although it is quite harder and longer than just using hashing.

Problem D
Divisibility Dance

Since the only movements allowed are rotations of an entire circle, the final configuration must also be some rotation of the initial pairing. To solve this problem, we need to answer two questions: which of the rotations satisfy the sum condition, and in how many ways can we reach each of those valid final configurations.

For the first question, it's useful to notice that, if the pairwise sum is constant when summing corresponding values of arrays A and B, then if going from position i to position $i+1$ the element in A increases by 3, the element of B must decrease by 3. In other words, the difference array of array B (the array D where $D_i = B_{i+1} - B_i$) is equal to the difference array of A with all elements negated.

Let's take the difference array of A, D_A, and the negative of the difference array of B, $-D_B$. We want to know for what rotations of D_A it becomes equal to $-D_B$. This is a classic string problem: we can concatenate two copies of D_A next to each other such that the N elements beginning at position i in the concatenated array correspond to D_A rotated i times, and then any string matching algorithm can be used to determine in which positions $-D_B$ matches the concatenated array.

Finally, we must determine the number of ways to reach each valid rotation. Note that from one configuration we can reach all rotations of that configuration except for the configuration itself, so by symmetry all rotations that are not the initial one are equivalent and can be reached in the same number of ways.

Let $f(x, 0)$ the number of ways to rotate x times and finish in the starting configuration, and $f(x, 1)$ the number of ways to rotate x times and reach some other configuration (as the number of ways is the same for all other configurations, it doesn't matter which one). We can then write:

$$f(x, 0) = (N - 1)f(x - 1, 1)$$
$$f(x, 1) = f(x - 1, 0) + (N - 2)f(x - 1, 1)$$

In other words, the initial configuration can be reached by any of the $N - 1$ other ones, and each of the other ones can be reached by the initial configuration and the other $N - 2$ configurations that are not themselves.

This is a linear recurrence, so it can be solved by matrix exponentiation in $O(\log K)$.

Problem E
Party Company

For each party i, first use binary lifting to find what is the earliest supervisor S_i of the owner O_i whose age is still in the range $[L_i, R_i]$. The owner of the party is always in the correct age range, so by the property that all supervisors are at least as old as their subordinates, we know that all employees in the chain between O_i and S_i are also in the correct age range, so S_i will be invited to the party i.

Since all partygoers are connected, we can consider any of them as the owner of the party and the list of invited people will not change. Therefore, for every party i, consider that S_i now is the owner of that party. This makes the upper limit of age not relevant anymore, since we know S_i's manager is too old and none of the subordinates of S_i is too old.

Also, by the same reasoning we used to argue that all employees between S_i and O_i are in the age range, we can show that any employee that is a subordinate of S_i and is inside the age range will be invited, since all employees in the chain between them and S_i will also be in the age range.

Therefore, the condition to be invited to a party can be rephrased as "all subordinates of S_i with age at least L_i are invited to the party i", which is much simpler than the original conditions.

We can now finish the problem in several ways: one of them is to do a DFS on the tree and add parties owned by the current employee in some structure that allows to query, in logarithmic time, how many parties have a lower age limit that is lower than the age of the current employee (a binary indexed tree is a good choice, for example). When the DFS leaves node v, we remove all parties owned by v from this structure.

Problem F

Fastminton

It is enough to keep the current number of points and games for both players and simulate all of the rules exactly as described in the statement. See the official solutions for more details on how to implement all of the checks.

Problem G
Game Show!

Ricardo's decision to continue or stop depends on only the value of the future reward that Ricardo can get: if he knows that he can get a positive value of sbecs by continuing to play, he should continue; otherwise, if he is going to lose sbecs, he should stop.

This suggests a dynamic programming approach to solve this problem, in which dp_i is the best value Ricardo can get if the last i boxes are still left in the game. Before each play, Ricardo can choose to stop, which gets him no sbecs, or continue, which gets him the value of the current box and brings him to the situation in which $i - 1$ boxes are left:

$$dp_i = \max(0, dp_{i-1} + value_i)$$

Our final answer is the best value Ricardo can get if all N boxes are left, plus his initial balance of 100 sbecs, giving a final answer of $100 + dp_N$.

Problem H

SBC's Hangar

First observe that the number of combinations that have a final weight in the interval $[A, B]$ is the number of combinations that have weight at most B, minus the number of combinations that have weight at most $A - 1$. Therefore, it suffices to find an algorithm to calculate how many combinations have weight at most W, for some W.

We can think of some backtracking solution, in which we would process the boxes in order, decide if the current box is going to be included in the final combination or not, then for each of the two possibilities recurse for the next boxes. This is too slow, as it has an exponential complexity of $O(2^N)$.

However, the boxes follow a very particular property, that for any two boxes, the larger box weighs always at least twice as much as the smaller box. This implies that the weight of each box is greater than the combined weight of all lighter boxes.

We can prove this by induction: the property holds for the two lightest boxes, since the smaller box weighs at most half of the larger box. Now assume the property is true for the k-th smallest box. For the $k + 1$-th smallest box, the weight of all boxes that weigh less is the weight of the k-th box, plus the weight of all boxes lighter than the k-th box. As we know the combined weight of all boxes lighter than the k-th box is lighter than the k-th box, the combined weight of all boxes lighter than the $k + 1$-th box is less than two times the weight of box k, and therefore less than the weight of box $k + 1$.

This property can be used to cut several possibilities of box selection in the backtracking approach. Order the boxes from heaviest to lightest, and process the boxes in this order. Then:

- If the box is heavier than the maximum allowed remaining weight, it can never be in any configuration, so we can simply ignore this box and move on.
- If the box is lighter than the maximum allowed remaining weight, we now have a choice: we can include this box in the final group or not. However, note that if we don't include this box, then *any* combination of the remaining boxes is valid, because their combined weight is less than the weight of this box and this box is lighter than the maximum. So if we choose to include this box we continue processing the following boxes, and if we choose not to include this box, we can add $\binom{n}{k}$ to the answer, where n is the number of remaining boxes and k is the number of boxes that still need to be included.

We now never recurse twice for any box, and in fact the algorithm above can be implemented with a single loop. The complexity is $O(N)$.

Problem I
Interactivity

The size of the minimal set of queries is the number of leaves of the tree. This can be proven by a recursive approach (hard to understand proof):

A subtree is *fully determined*, meaning you can know all the values in this subtree with all queries, if:

1) Every children of the root of this subtree is *fully determined*. So you can just sum the value of the children to determine the value of the root.

2) The root value is determined by a query, and every children of the root of this subtree is *fully determined*, except one, which can become *fully determined* if you knew the value of its root. So you can subtract the value of the current root from the sum of values of the children that are determined, thus calculating the value of the child's root that is not *fully determined*.

It's easy to see that, in the 2nd case, you can't have more than one subtree that isn't *fully determined*, since it would only be possible to calculate the sum of its root's values, which would lead to multiple possible trees. Also it's not optimal to have both root and children subtrees *fully determined*, because the root value can be calculated in case you have the value of children's roots.

Then you can use this definition going up from leaves until the root of the tree for the proof:

1) Leaves that were queried are *fully determined*, and leaves that were not queried are not.

2) The internal node that is a parent of a leaf that is not *fully determined* (and all other children being *fully determined*, otherwise it can't be uniquely determined), has to be determined by a query to make its subtree *fully determined* or it is a subtree that is not *fully determined* but it can become in case you can know the value of its root (which is part of the case 2 of the definition).

This means that for each leaf that is not queried, another node on the path from it to the root must be queries to be able to *fully determine* the whole tree, and this query on the internal node uniquely compensates a single leaf not being queried. So the size of the minimal set of queries must be the number of leaves of the tree.

Using the definition we can formulate a dynamic programming solution, similar to Minimum Vertex Cover, to calculate the amount of different minimal sets we can have. Let $dp(u, k)$ be the total number of different minimal sets of the subtree of the node u that needs k extra queries to become *fully determined*. k can only be 0 or 1, since either the subtree is already *fully determined* or we need a single extra query to compensate one leaf still not compensated (if we have more queries not being compensated than this subtree can't become *fully determined*, as described).

Leaf case: $dp(u, 0) = 1$, $dp(u, 1) = 1$
Internal node case:

Let $S1 = \prod dp(v, 0)$, where v are children nodes of u, which means the sum of all children's arrangements in case every single one is *fully determined*.

Let $S2 = \sum S1 \times dp(v, 0)^{-1} \times dp(v, 1)$, which means the sum of all possible arrangements where only a single children's subtree needs an extra query to become *fully determined*.

$dp(u, 0) = S1 + S2$
$dp(u, 1) = S2$

Explanation:

If $k = 0$, the subtree is *fully determined*, so the definition is directly applied: either all children's subtrees are *fully determined* (which is $S1$), or a single child's subtree needs an extra query and we have to do this query at u, since $k = 0$, (which is $S2$).

Else ($k = 1$) one child's subtree must be needing an extra query (extra queries must come from children since it means leaves that don't have queries compensated) (which is $S2$).

The tricky case is to calculate $S2$ in modulo arithmetic. There can be a problem in case $S1$ is multiple of the modulo (in case $10^9 + 7$), so $S1$ would be zero, and $S2$ would also be zero, even when the only term multiple of the modulo is the swapped term.

To avoid this you can preprocess the prefixes and suffixes of $S1$:
$pre_i = \prod_0^{i-1} dp(v, 0)$
$suf_i = \prod_{i+1}^{c} dp(v, 0)$
and change the $S2$ expression to $S2 = \sum_{i=0}^{c} pre_{i-1} \times dp(v, 1) \times suf_{i+1}$.

Problem J
Collecting Data

This problem has many details, so it is very useful to eliminate some edge cases by treating them separately. First, if all values are the same, then all pairs are the same point and we have exactly one configuration. We can check if horizontal or vertical lines are possible by verifying if at least half of the values are the same, and then we can ignore horizontal and vertical lines for the remainder of the solution. Extra care must be taken with the case in which two values appear in the input with frequency of $\frac{N}{2}$ each, which is another case better handled separately.

After these edge cases are handled, we can now make two simplifying assumptions: for every configuration of points, there is exactly one line that goes through all points in this configuration, and for every value of x there is exactly one value of y that is in this line, and vice-versa. This is very useful because it helps avoid double-counting: we can now count for every possible line how many configurations form that line, and we know that a configuration can only be counted once as it only forms a single line.

Our strategy therefore is to form a list of candidate lines, and then for each line count in how many ways we can pair coordinates so that the line goes through all points (possibly zero, if it is impossible to make such a pairing).

The most straightforward strategy to choose candidate lines is to iterate through all ordered tuples of 4 values (v_a, v_b, v_c, v_d) chosen from the input, and take the line that goes through the points (v_a, v_b) and (v_c, v_d) as a candidate line. This will reach the right answer, but is too slow as there would potentially be $O(N^4)$ lines generated by this process.

To speed this up, we need to use the fact that line doesn't go through only two of the points, but through all of them. One idea is to filter only lines that were generated many times in the above process, which can work but is tricky to get right. A more direct approach is to note that since the line goes through all points, it must also go through the point that uses the smallest of the coordinates v_1. Similarly, it must also go through a point that uses the smallest coordinate v_2 that is greater than v_1. In this manner, it suffices to iterate only through tuples that contain both v_1 and v_2 in any position. This guarantees that the number of tuples that will be analyzed is $O(N^2)$.

Some notes on the correctness of the above approach: note that the point that goes through v_1 might be the same that goes through v_2, but this is not a problem since in this case we can take the remaining two values (v_3, v_4) to be any other point in the line. It is very important that we choose $v_1 \neq v_2$, as we need two distinct points to determine a line. Had we chosen $v_1 = v_2$, we could hit a case in which both of the points that use v_1 and v_2 are in the same location, which would make determining the line impossible. This is not a problem for the case $v_1 \neq v_2$, as if the points (v_1, v_a) and (v_b, v_2) are in the same location, this implies the point (v_1, v_2) is also in the line, so we will add this line as a candidate line later.

It remains to determine in how many ways each line can be formed. Here our second observation comes into play: since we are ignoring horizontal and vertical lines, for each x there is exactly one corresponding y in the line, and for each y there is exactly one corresponding x in the line. This means that for every value it has potentially two other values it could be paired to (depending on which coordinate it is being used as).

If we make a graph in which each node is a coordinate value and an edge means that two coordinates can be paired to form a point, each node will have degree at most 2. A graph in which every node has degree at most 2 is a union of paths and cycles. Paths can be paired greedily because the end of the path has only one possibility, so it is possible to repeatedly pair the end of the path until a contradiction is reached or the path is fully paired.

Cycles would be trickier, but for this particular graph we can show that all cycles are actually very small. Indeed, if we write the line as $y(x) = ax + b$, then a potential cycle of size 3 would imply $y(y(y(x))) = x$, which is a linear equation and therefore has only one solution for x, which must be

the fixed point of the line (the point of the line such that $y(x) = x$). We still have to be careful with edge cases when solving this linear equation, as it might involve a division by zero for two values of a, which have to be analyzed separately. If $a = 1$ then all points are fixed points if $b = 0$, or there are no fixed points if $b \neq 0$. If $a = -1$, then $y(y(x)) = x$ for all x.

Therefore we can only have paths and fixed points for most lines, unless $a = -1$ in which case there may also be cycles of length 2. For fixed points, it's enough to check if the value appears in the input an even or odd number of times; an odd number is impossible to pair and an even number can be paired in exactly one way.

The cycles of length 2 are an interesting case, because they are the only configuration that can be paired in more than one way: if both A and B appear in the input an equal number of times, then we can choose how many points of the form (A, B) and how many points of the form (B, A) will be formed. (Note that we treated cases that are a single 2-cycle separately earlier, so it is not a problem to select all points to be in the same location here).

There are $O(N^2)$ candidate lines, and we can test each candidate line in $O(N)$, so the overall complexity is $O(N^3)$.

Problem K
Between Us

Describe the final partition as a set of N variables $x_1, x_2, \ldots, x_N$, such that for every i, $x_i = 0$ if student i is in the first group and $x_i = 1$ if student i is in the second group.

Let the friends of student i be students $A, B, C, \ldots$. Analyzing the condition "the number of friends of student i that are in the same group as i is an odd number", we have two cases:

- Student i has an even number of friends in total

In this case, note that it does not matter for the condition in which group student i is in: either both groups have an odd number of friends of i, or both groups have an even number of friends of i. To have a valid partition, we want the first condition to hold (both groups have an odd number of friends of i), which is equivalent to saying that an odd number of the variables $x_A, x_B, x_C, \ldots$ is equal to 1. This can be written mathematically as $x_A \oplus x_B \oplus x_C \oplus \cdots = 1$, where $\oplus$ denotes the XOR operation (exclusive or).

- Student i has an odd number of friends in total

In this case, there is exactly one group with an odd number of friends of student i, so the position of student i is uniquely determined by the position of all of their friends. We can write the desired position for student i as $x_i = x_A \oplus x_B \oplus x_C \oplus \cdots$, or after rearranging the terms, $x_i \oplus x_A \oplus x_B \oplus x_C \oplus \cdots = 0$.

Therefore, for both cases, the condition can be expressed as an equation relating some of our variables. The exclusive or operation is addition under modulo 2, so those are also linear equations, and we can solve the system of linear equations using Gaussian elimination in $O(N^3)$. There is a valid partition if and only if the system has at least one solution.

Problem L
Lavaspar

The brute-force idea would be:

Create an auxiliary matrix A to count the number of words that covers each position.

Create an auxiliary matrix B to count, for the current word being processed, which positions have a matching for any anagram of it (this matrix has only zeroes and ones).

Then, for each word in the collection ($O(N)$), reset matrix B to zeroes ($O(L \times C)$), generate all anagrams of the word and ($O(P!)$), for each anagram, iterate in the original matrix for each initial position ($O(L \times C)$) and try to match to right, down and diagonal, settings 1 on every position that has a match ($O(P)$). After all anagrams, sum the matrix B into A, and go to next word ($O(L \times C)$).

This would lead to a $O(N \times (L \times C + (P! \times L \times C \times P))) = O(N \times P! \times P \times L \times C)$ complexity, which is totally above the time limit.

To improve it, we can't iterate on each anagram. We count the number of appearances of each letter in the current word $O(P)$ and we count the number of appearances of each letter in the matrix for each interval of length P ($O(L \times C \times P)$), going to right, left or diagonal, and for each interval we can compare if the amount of each letter is the same or not ($O(26)$). In case the interval matches, we have an anagram and we can update the auxiliary matrix B with ones ($O(P)$).

This would lead to a complexity of $O(26 \times N \times P^2 \times L \times C)$. Which may be enough to pass, but we can optimize some details. We just need to use one of the optimizations below, but we can use all of them together:

1) To update the auxiliary matrix B we can use the concept of sum in interval with prefix sums: sum 1 at the start and -1 one positions after the end. We have to split horizontal, vertical and diagonal tests to use this, though, but it would remove P from the complexity.

2) We could also improve the check. Instead of checking each one of the 26 letters for each interval, we can create a *letters_matching* variable that counts the amount of letters that have the same value between the interval being checked and the word, and if it's 26 we know that the interval matches some anagram of the word. To calculate it, when we iterate over the desired interval, we add each letter to the amount of that letter the interval has, then we can check if this value is equal to the value in our word values (then we sum 1 to *letters_matching*) or if it was equal before and it's not equal anymore (then we subtract 1). After going through every letter of the interval, *letters_matching* stores how many letters, from 'a' to 'z', have the same number of appearances between the interval and the current word. This removes the 26 from the complexity.

3) Finally we can also use a sliding window to not have to go through every interval of length P starting in every position of the matrix. For each line, start with an empty interval before the first letter of the line. Expand the interval by adding the letters one by one of this line until we have an interval of length P. Then we check if this interval is an anagram. Then add next letter and remove the first letter, which makes the current interval have length P. Then we check again if they match and repeat this adding/removing until we don't have more letters to add. Then do the same for columns and diagonals. This would remove P from the complexity.

Using all three optimizations we can go down to $O(N \times L \times C)$ complexity.

Problem M
Machine Gun

The angle that a machine gun covers, casts on the $x = 0$ vertical axis a certain interval. If we cast analogous angles from each initial enemy towards the left, each will cast its own interval over this line.

It can be verified that a machine gun kills an enemy precisely when the machine gun interval intersects the enemy interval.

Thus, queries can be answered using an interval tree `https://en.wikipedia.org/wiki/Interval_tree`, in $O((N + Q + m) \lg N)$ time, where m is the total number of killed enemies.

Alternatively, coordinate compression + persistent segment tree + sweep line + binary search can be used: after changing the plane coordinates so that machine guns span 90 degrees angles aligned with the coordinate axes, each query asks for the set of given points that are above and to the left of a certain point. Using sum-queries over rectangles and binary searching to quickly find where the points are, the full set of interesting points can be found. This solution has query complexity $\lg^2 N$ instead of $\lg N$, but it should probably be allowed to pass. The technique is more standard and well known, but the implementation is probably trickier than that of an interval tree. Note that this solution is in fact more general: it can handle queries of this kind with points anywhere at all in the plane, while the interval tree solution makes use of the special structure that these queries have (all points are above the $y = x$ line, and all queries corners are below that diagonal).

The offline version of the problem would be quite easier: when all queries are known in advance, they can be added as special points and then a single sweep line performed, keeping all seen enemies in a c++ set so that when each query is found, the answer can be read from the set using a simple lower bound operation.

However, this same idea can be made to work efficiently for the problem by implementing and using a persistent balanced binary search tree, giving a third solution to the problem.

Problem N
Number Multiplication

This problem allows two solutions:

The first one may annoy you, since you don't need to read the whole input to solve it.

Just take a fast factorization algorithm (let's say Pollard's rho), factorize every composite, put every prime factor in a set and print them in order.

The complexity of this solution is something like $O(N * \sqrt[4]{max(composites)} * log(max(composites)))$.

Before moving forward, try to think the other solution. As a clue, the expected complexity is $O(E + \sqrt{(max(composites))})$.

———————≀———————

The idea is to realize that we can do a simple $O(\sqrt{N})$ factorization algorithm going through all primes smaller than $\sqrt{N}$ just once.

We can keep an index `so_far`, the largest number we have tested and repeat the following:

1. If `so_far` is greater than $\sqrt{max(composites)}$ then take all composites values different than 1, and print them in order, they're all primes. There can be repeated values, so be careful. Exit
2. Take any composite node connected to the smallest prime node not yet discovered (remember they're ordered)
3. Starting in `so_far` check what's the next number that divides that composite (it'll be a prime). Check if `so_far` at any point becomes larger than $\sqrt{max(composites)}$, if it does, go back to 1.
4. Once found, print it (it's part of the answer), mark that node as discovered and divide every composite node connected to that prime node as many times as possible (the edge actually has that value). Increase `so_far` and repeat.

Problem O
Venusian Shuttle

Each position in the shuttle is going to receive sunlight in some parts of the path and not receive any sunlight in other parts. For now, we will not consider this detail and assume that a position receives sunlight during the whole trip, and find the position that receives the least amount of sunlight.

Each line in the shuttle path can be described by the parameters L, the length, and α, the orientation of the line. Someone sitting in a position x of the shuttle receives, during this line, $L\cos(x+\alpha)$ units of sunlight. The total sunlight is therefore given by a sum of several cosines.

It is now very useful to know that a sum of several cosines with same frequency is still a cosine multiplied by some constant, that is, we can write

$$A\cos(x+\alpha) + B\cos(x+\beta) + \cdots = C\cos(x+\gamma)$$

for some appropriate value of C and γ. There are several ways to reach this conclusion, including geometrical approaches and using complex numbers. A direct algebraic way is to use the expression for cosine of a sum:

$$\cos(A+B) = \cos A\cos B - \sin A\sin B$$

This expression lets us rewrite $A\cos(x+\alpha)$ as $B\cos x + C\sin x$, which makes adding two cosines with different orientations easy as we can now separately add the coefficients for $\cos x$ and $\sin x$. We can also use the same expression in reverse to convert our final expression $B\cos x + C\sin x$ back into a single cosine, and from there finding the minimum value of the expression is simple.

Remembering that not all positions receive sunlight in all parts of the path, we need to include in our sum only the cosines from parts of the path in which a position x receives sunlight. We can note that a line of the path provides sunlight to an interval of angles $[-\frac{\pi}{2}-\alpha, \frac{\pi}{2}-\alpha]$. We can sort the endpoints of those intervals and use a line sweep on them to consider for every interval of x values, only the cosines that are relevant to those values of x.

Finally, we can note that as each sum only includes positive terms, the minimum for each interval $[x_1, x_2]$ must be in x_1 or x_2, so it suffices to compute the sum for both extremes of all intervals hit by the line sweep.