Problem A. Alternating Paths

There are three fundamentally different cases:

1. The graph is bipartite.
2. The graph is not bipartite, but can be made bipartite by removing one edge.
3. The graph is not bipartite and it can't be made bipartite by removing a single edge.

Because of the number of sub-cases and the extreme difficulty level of the task, I won't present a complete solution here. The full solution can be found in https://arxiv.org/abs/1907.00428. Implementing case 3 using this is somewhat complicated though. The authors assume without loss of generality that no two edge-disjoint odd cycles exist, but when you implement the algorithm from the paper, there are many ways in which they will arise, and you need to check for them every time. There might also be some missed edge cases in the paper, it's hard to tell.

Case 1: Bipartite graphs

Suppose we are given a bipartite graph (with black and white vertices) and a valid edge coloring (with red and blue edges). If we start our path from a white vertex u and take a red edge, we will arrive at a black vertex and will be forced to take a blue edge. After that, we will be at a white vertex again and be forced to take a red edge again, and so on. In other words, after the first move, every edge can only ever be traversed in one direction.

Assigning edge colors on a bipartite graph is therefore the same thing as orienting the edges of a bipartite graph: red edges are directed from white to black and blue edges are directed from black to white. We have to do this in such a way that for every pair u, v of vertices, there is either a directed path $u \rightsquigarrow v$ or a directed path $v \rightsquigarrow u$.

If the bridge tree of the graph has any vertices with degree greater than 2, this is clearly impossible. Therefore, for this to be possible, the bridge tree of a graph must be a line. Let's define *bipartite rope* to mean a bipartite graph whose bridge tree is a line.

On a bipartite rope, orient every biconnected component in a way that makes it strongly connected (it's well-known that this is always possible) and orient all the bridges in the "same direction". It's easy to see that this is a valid solution.

Case 2: Almost bipartite graphs

We define an edge to be *stubborn* if its removal makes the graph bipartite. Let S denote the set of stubborn edges. We'll call the connected components of $G \setminus S$ *stubborn-free components*. Clearly, each stubborn-free component is bipartite.

Claim 1. *Compress each stubborn-free component to a single vertex. The resulting graph H is a cycle.*

Proof. Clearly, every vertex in H will have a degree of at least two (although there might be a vertex that is only connected to two endpoints of the same edge).

If H is not a single cycle, then you can always find an odd circuit that doesn't visit every stubborn edge, and therefore the removal of any stubborn edge that this circuit doesn't visit will not result in G becoming bipartite. See more details in the paper. □

Claim 2. *A solution can only exist if there is a stubborn-free component K such that $G \setminus K$ is a bipartite rope.*

Proof. Omitted, see paper. □

Let then K be a stubborn-free component such that $G \setminus K$ is a bipartite rope. We will use the same model of orienting edges as in case 1:

- A blue edge between a black vertex and a white vertex is oriented from black to white.
- A red edge between a black vertex and a white vertex is oriented from white to black.

We must be able to move from any vertex to any other vertex. We can choose whether we are moving backwards or forwards at the beginning, but we can't change that arbitrarily midway through.

This time though, no matter how we color the vertices, we will end up with two adjacent vertices with the same color somewhere. Therefore, we need to introduce two additional edge types:

- A blue edge between two black vertices and a red edge between two white vertices are *forwards-reversing edges*: they can only be entered while moving forwards and they will change the direction of movement to moving backwards.
- A blue edge between two white vertices and a red edge between two black vertices are *backwards-reversing edges*: they can only be entered while moving backwards and they will change the direction of movement to moving forwards.

It may be that $G \setminus K$ is empty. Then the only stubborn edge connects two vertices in K. This is an edge case and you may need to consider it separately. However it is very similar to the general case, so we won't discuss it here.

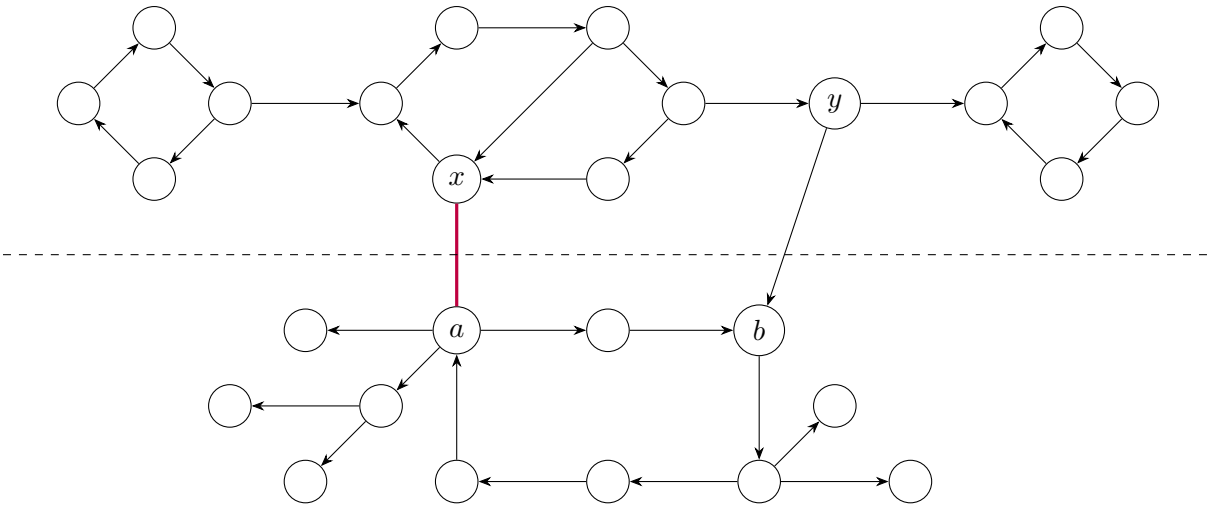
There are two edges connecting $G \setminus K$ and K: let these edges be xa and yb, with $x, y \in G \setminus K$ and $a, b \in K$. Color the vertices of $G \setminus K$ in a standard way and then orient the edges of $G \setminus K$ as in case 1. Without loss of generality, suppose a path $x \rightsquigarrow y$ exists in $G \setminus K$.

Now color the vertices of K in a standard way. Among the two possible options, choose the one that makes x and a have the same color. Because G is not bipartite, this means that y and b have opposite colors.

Claim 3. *There can be no bridge on the path between a and b in K.*

Proof. Suppose there was a bridge e. Then if we remove e, it will split K into two connected, bipartite components. Each component would be connected to $G \setminus K$ by only one edge, which would mean that $G \setminus e$ is bipartite, implying that e is stubborn. But $e \in K$ and K is a stubborn-free component. □

Therefore, there exist two edge-disjoint paths between a and b in K. If we concatenate these paths, we get a cycle C. Orient the edges of C in a cyclic way, and orient all other edges of K in such a way that every vertex of K is reachable from C when moving forwards. Finally, orient the edge yb from y to b and let xa be a backwards-reversing edge.

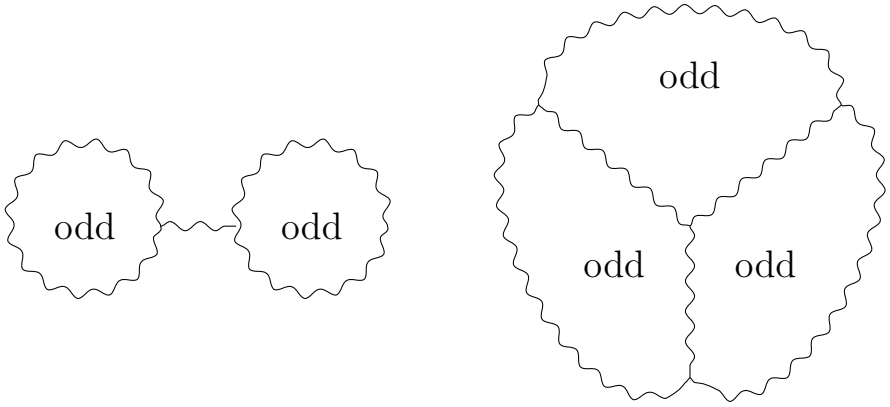


We check that this is a valid solution.

- $u, v \in G \setminus K$. Covered by case 1.
- $u, v \in K$. Start moving backwards from u. Navigate to C. Move to b via the cycle, then to y. Move to x. Now enter xa and start moving forwards. a is on the cycle, so v is reachable.
- $u \in G \setminus K, v \in K$, *$y$ is reachable from u moving forwards.* Move to y, then to a. a is on the cycle, so v is reachable moving forwards.
- $u \in G \setminus K, v \in K$, *$x$ is reachable from u moving backwards.* Start moving backwards to x. Enter xa and start moving forwards. a is on the cycle, so v is reachable.

Case 3: Not at all bipartite graphs

The graph will always contain one of the following structures as a subgraph: either two edge-disjoint odd cycles or three conjoined odd cycles.



At a first glance, the algorithm for finding this structure doesn't seem too complicated. A basic overview is:

1. Pick an odd cycle C_1.
2. Let $e \in C_1$, let C_2 be an odd cycle without e.
3. Adjust C_1 and C_2 such that their intersection is a path P.
4. Let C_3 be an odd cycle that doesn't visit the first edge of P.
5. If $C_1 \cap C_2 \cap C_3 = \varnothing$, adjust C_3 to make the structure above. Otherwise, adjust C_3 to create a pair (C_1', C_2') whose intersection is a path shorter than P. Go to step 4.

This doesn't sound too bad, but there are a couple caveats:

- Most steps above have exceptional cases where two disjoint cycles arise.
- These "adjustments" aren't as simple as they seem. This is in part due to repeated vertices: even if C_1 and C_2 have a path as an edge intersection, they may still have common vertices outside. That can lead to unintuitive degenerate cases when improperly handled.

First, I'll define a couple of operations that will be very useful to us. Here, a *circuit* is a closed walk (may visit the same edge twice), and a *cycle* is a circuit that doesn't visit any vertex twice.

First, suppose we have an odd circuit C. It's always possible to find an odd cycle that consists only of the edges of C. I'll call this *eliminating repeated vertices of* C.

Suppose we have two simple odd cycles C_1 and C_2, which are not equal but have a nonempty intersection. Their intersection can be quite complicated. However, the edges of C_2 can be partitioned into an even number of segments, where even-numbered segments are shared with C_1 and odd-numbered segments are disjoint from C_1. Pick an odd-numbered segment Q. The endpoints of Q partition C_1 into two paths P_1 and P_2 with different parities. One of $C_2 \cup P_1$ and $C_2 \cup P_2$ is odd. Take the odd one of them and eliminate repeated vertices; let that cycle be C. Let's call this the *normalization of* C_2 *w.r.t.* C_1.

The intersection of C and C_1 is now simply a path. Also, the symmetric difference $C \triangle C_1$ is an circuit (rather than a disjoint union of multiple circuits).

Now we are ready to present the algorithm to find two edge-disjoint odd cycles or three conjoined odd cycles.

1. Find an odd cycle C_1.
2. Let $e \in C_1$. Find an odd cycle C_2 that doesn't visit C_1. This always exists because G doesn't have any stubborn edges.
3. If C_1 and C_2 are edge-disjoint, we have a solution.
4. Let C_2 be the normalization of C_2 w.r.t. C_1. Now their intersection is a path P.
5. Let C_3 be an odd cycle that doesn't use e, the first edge of P. If C_3 is disjoint from C_1, we are done.
6. Let C_3 be the normalization of C_3 w.r.t C_1. Use a normalization that doesn't contain the edge e. This is always possible.
7. If C_3 and C_2 are edge-disjoint, we are done.
8. If $C_1 \cap C_2 \cap C_3 = \varnothing$, we are done.
9. Now, the intersection of C_1 and C_3 is also a path, Q. Since we didn't find a solution in the previous step, $Q \cap P \neq \varnothing$. But since Q doesn't contain e, also $P \neq Q$. If $Q \subset P$, then set $C_2 \leftarrow C_3$ and go to step 5.
10. Now, we know that P and Q are subpaths of C_1 where P starts somewhere in the middle of C_1 and goes on a bit further. Let e' be the last edge of $C_1 \triangle C_2$ that C_3 visits before entering P. Between e' and P, this forms a path W that doesn't visit C_1 or C_2 at all.
    - If $e' \in C_1$, then this actually means that C_3 doesn't have any common edges with C_2 outside P. Then, the intersection of C_3 and C_2 is a path strictly shorter than P. Set $C_1 \leftarrow C_3$ and go to step 5.
    - If $e' \in C_2$, then the endpoints of W partition C_2 in two walks with different parities. One of them can be joined up with W to form an odd circuit C_4, whose intersection with C_1 is a strict subpath of P. Eliminate repetitions in C_4. After that, it may be entirely disjoint from C_1, in which case we are done. Otherwise, the intersection of C_1 and C_4 is a strict subpath of P again. Set $C_2 \leftarrow C_4$ and go to step 5.

Notice that this algorithm always terminates: every time we went back to step 5, the intersection between C_1 and C_2 became a path strictly shorter than before. Therefore, this algorithm terminates after $O(n)$ iterations.

The algorithm above gave us either two edge-disjoint odd cycles or three odd cycles C_1, C_2 and C_3 such that their intersection is empty. However, they might not look as nice as in the image above because although the intersection of C_1 and C_2 is guaranteed to be a path, and the intersection of C_1 and C_3 is also guaranteed to be a path, the intersection of C_2 and C_3 may be quite messy. Let's simplify them.

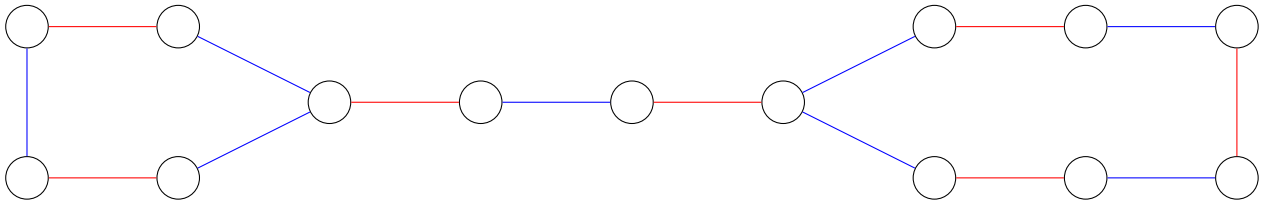
We want to partition C_3 into segments again, based on whether or not they appear in $C = C_1 \triangle C_2$. However, there is a problem: C is an even circuit, but not necessarily a cycle. It has no repeated edges, but may have repeated vertices. If you can start from a vertex v, move along C and get back to v in an odd number of steps, C can be partitioned into two odd circuits. Eliminate repeats and we have two odd cycles, so we'll use the solution for that instead. If, however, each vertex can only be reached from itself in an even number of steps, do nothing.

Now C is an even cycle. Partition C_3 into segments, where even-numbered segments consist of edges that are shared with C and odd-numbered ones consist of edges that are not. Notice that the even-numbered segments are not necessarily segments of C (because of repeated vertices in C), but their lengths do have the same parities. At least one odd-numbered segment has length of a different parity than that of the underlying segment in C (either of them). Let that segment be W.

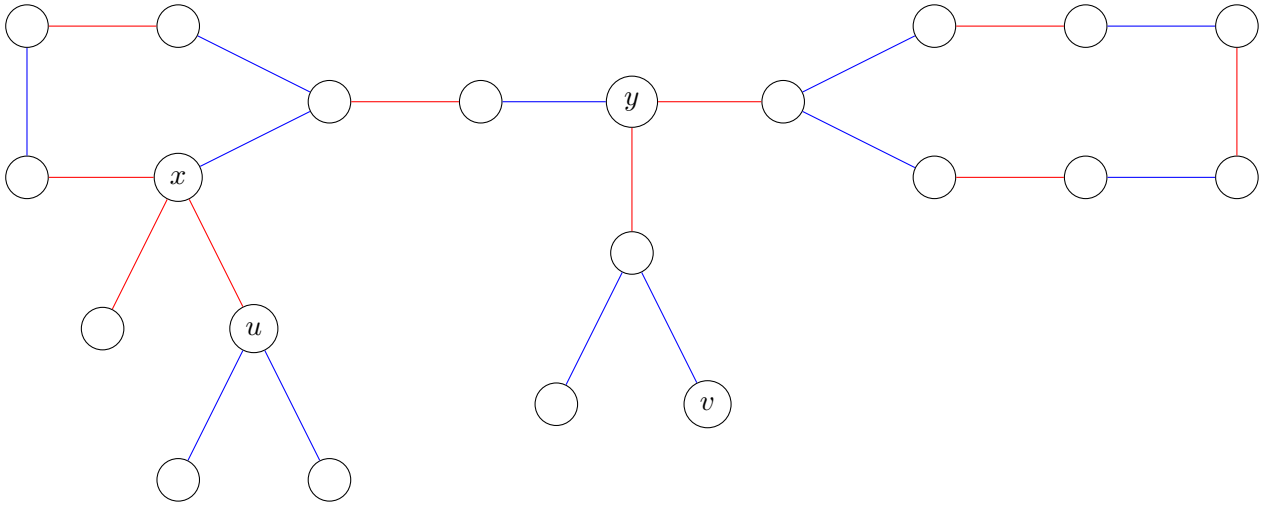
If C_3 visited C_1 immediately after and before the segment W, then the one of the underlying walks in C of W doesn't contain any edge of C_2, thus we can construct an odd circuit that doesn't visit C_2 at all. After eliminating repeats, we have two disjoint odd cycles, use the solution for that. The same happens if C_3 visited C_2 immediately after and before. Otherwise, we finally get that nice "three conjoined cycles" structure that we have in the picture above.

The coloring itself

Suppose we have two disjoint odd cycles. The graph is connected, so there must be a path connecting them. We color the edges as follows:



You can check that wherever we start on this structure, we can move to any other vertex. Importantly, we can choose in advance the first color and the last color along our walk, and it will still be possible. To add access to other vertices, connect them up with "trees" as follows:



Other edges can be colored arbitrarily. Now from any vertex u, you can first go up the tree to a vertex x on the structure. Then, go to another vertex y on the structure, turning around on the cycle if necessary, and then go down the tree again.

The solution for three conjoined cycles is actually very similar: pretend that the common edges of C_1 and C_2 are different edges, and use the same solution as above. The parities will ensure that this is well-defined and won't assign two different colors to the same edge.

This problem originally appeared in the Estonian Open Contest 2018, as an open tests problem. The number of different cases and the different strategies needed for such cases made it a very suitable problem for open tests format. We think we developed a full solution afterwards, but no comprehensive description or full implementation exists. The paper above appeared in June 2019, and was accepted a year later.

The model solution is 1216 lines long (but I write sparse code). Initially, I didn't expect it to be so long, but as more and more special cases popped up, the code rapidly became longer and longer. Case 3, specifically extracting suitable subgraphs from the graph is really the culprit here, the other two cases are more or less reasonable. That case takes 717 lines, almost two-thirds.

I still think it might be possible to make the implementation much shorter, by somehow streamlining the process of finding either two disjoint odd cycles or three odd cycles with no edge common to all of them (using DFS tree perhaps?).

Problem B. Binary Sequence

Let s_n denote the sequence and $f(s)$ denote the "look-and-say operation", i.e. $f(s_n) = s_{n+1}$. It turns out that for $n > 2$, s_{n-2} is a suffix of s_n.

We prove a slightly stronger claim by induction: that s_n is of the form $x_n 0 s_{n-2}$ for some string x_n. Cases $n = 3$ and $n = 4$ can be checked manually. For $n \geq 5$:

$$s_n = f(s_{n-1}) = f(x_{n-1}0s_{n-3}).$$

Since $x_{n-1}0$ ends with 0 and s_{n-3} starts with 1, they share no contiguous sequence of similar digits in $x_{n-1}0s_{n-3}$; the look-and-say operation treats them completely separately. Therefore,

$$s_n = f(x_{n-1}0)f(s_{n-3}) = f(x_{n-1}0)s_{n-2}.$$

Since $x_{n-1}0$ ends with 0, so does $f(x_{n-1}0)$, thus we can write $s_n = x_n 0 s_{n-2}$.

Therefore, it is sufficient to calculate the first few elements of the sequence until we find two (one for each parity) that has length longer than 10^6.

Problem C. Yet Another Balanced Coloring Problem

We model a leaf being red as having charge -1, and being blue as having a charge of $+1$. The condition asks to produce a coloring such that for each vertex u, the sum of charges of leaves in the subtree u is in $\{-1, 0, 1\}$.

Equivalently, we can define the charge of a vertex as the sum of charges of its children and require that the charge of each vertex is in $\{-1, 0, 1\}$. Notice that if a vertex u has an even number of leaves in its subtree, this means that the charge of u must be 0. Therefore, the edge between u and its parent p can be removed without changing the substance of the constraint. Observe that after this transformation, each internal vertex (that is, each vertex except the root and leaves) has an even degree: it either has an odd number of children and an edge up, or an even number of children and no edge up.

Now, let's focus on the first forest. Look at a vertex u and its parent p. We model u having a charge of $+1$ as the edge *up* being directed away from u, and u having a charge of -1 as the edge *up* being directed into u. Because the charge of an internal vertex u must be the sum of its children's charges, the number of incoming edges to u must equal the number of outgoing edges. In a root vertex, these numbers may differ by at most one.

We do the same thing to the second forest, except orient the edges the other way: u having a charge of $+1$ is the parent-edge directed into u. Now, let's merge the forests: relabel the vertices $k + 1 \ldots m$ to

$n+1 \dots m+n-k$ in the second forest. Leaves now also have degree 2 and must have one incoming and one outgoing edge.

The problem has been transformed to the following: given a graph where all vertices except either 0 or 2 have even degree, orient the edges such that all vertices have an equal number of incoming and outgoing edges (except for the vertices with odd degree, where the difference must be 1). This is simply finding an Eulerian circuit.

Here's some theoretical background (and also how I came up with the problem) on tasks of this type.

A matrix A is called *totally unimodular* if the determinant of each square submatrix is in $\{-1, 0, +1\}$. Totally unimodular matrices have some nice properties in integer programming. For example, given a totally unimodular matrix A and an integer vector b, all vertices of the polyhedron

$$\{x \in \mathbb{R}^m \mid Ax \leq b\}$$

have integer coordinates. This implies that optimization problems of the form

$$\max c^\top x \text{ s.t. } Ax \leq b$$

have an optimal solution with integer coordinates. (For general matrices, the problem of finding an optimal integer solution is NP-hard.)

There is a very nice criterion for checking whether a matrix is totally unimodular.

Theorem 1 (Ghouila-Houri)**.** *A matrix $A \in \mathbb{R}^{n \times m}$ is totally unimodular if and only if for every subset $J \subseteq [m]$ can be partitioned into two sets J_+ and J_- such that all entries of the vector*

$$\sum_{j \in J_+} A_{*,j} - \sum_{j \in J_-} A_{*,j}$$

are in $\{-1, 0, +1\}$.

In other words, you are given a matrix and a subset of its columns. You must give a charge of -1 or $+1$ to every column such that in the charged sum, every entry is in $\{-1, 0, +1\}$.

The claim is that if this is possible for every subset of columns, then the matrix is totally unimodular. But notice that the transpose of a totally unimodular matrix is also totally unimodular. Therefore, a corresponding problem with rows is also possible for every subset of rows if and only if the matrix is totally unimodular.

This gives rise to a kind of duality: there is a "row problem" and a "column problem", and each is solvable only if the other is. This is related to, but not exactly the same as LP duality.

Let's try to formulate this problem in terms of theorem 1. We make a column for each leaf, and a row for each other vertex. We set $A_{i,j} = 1$ if j is in the subtree of i, and 0 otherwise.

The dual problem is then:

> Given two trees with a shared set of leaves (as in the original task). Give a charge of either $+1$ or -1 to each internal vertex such that for each leaf, the sum of the charges of its ancestors (in both trees) is in $\{-1, 0, +1\}$.

This dual problem is much easier! All we have to do is assign charges in both trees such that each child has a different charge than its parent, and make sure that the roots of the trees have different charges. This implies that:

- The primal problem is always solvable.
- The primal problem can be solved in polynomial time. (this is implied by the proof of theorem 1).
- It also gives hope that the primal problem might be solvable in (near-)linear time (since the dual is).

As another example, consider this rather well-known problem:

> Given a set of points and a set of intervals in 1D space. Give a charge of either $+1$ or -1 to each interval such that for each point, the sum of charges of the intervals that cover it is in $\{-1, 0, +1\}$.

Again, the dual problem is much simpler and has the same implications for the primal problem:

> Given a set of points and a set of intervals in 1D space. Give a charge of either $+1$ or -1 to each point such that for each interval, the sum of charges of the points that it covers is in $\{-1, 0, +1\}$.

The solution in this case is to simply sort the points and give them alternating charges.

Problem D. Filesystem

We are given a subset S of items and two permutations, which we will call the red permutation and the blue permutation, corresponding to the order of files by file name and by creation date. Note that in the problem statement, the first one is fixed to be just the identity permutation. We have to construct a family of disjoint subsets covering exactly S such that each subset is a contiguous segment in one of the permutations.

We consider the problem as a graph problem. For each vertex, record its "next" vertex on the blue permutation and its "next" vertex on the red permutation. Delete all vertices corresponding to files you do not wish to upload. We must decompose this graph into a set of disjoint paths, where each path is either all-red or all-blue.

Imagine deciding, for every vertex, whether it should be red or blue. After making this choice, we add to the decomposition all existing edges between vertices of the same color. We must pay for all vertices that are the last in their path: i.e. if the next vertex for that color is either missing or in the opposite color.

We model the system as min-cut. Being red is modeled as being in the same component as s, being blue is modeled as being in the same component as t. If a vertex u is red but nxt_red$[u]$ is blue, we must pay: add an edge $u \to$ nxt_red$[u]$. If a vertex u is blue but nxt_blue$[u]$ is red, we must also pay: add an edge nxt_blue$[u] \to u$. If the next vertex doesn't exist, it can similarly be treated by adding an edge from s or to t. Now the answer is precisely the capacity of the minimum cut.

Yes, this is a problem from real life. There was a very annoying bug that prevented me from using Ctrl+click and thus forced me to only upload contiguous segments at a time. I believe this is the bug that affected me: https://gitlab.gnome.org/GNOME/gtk/-/issues/5779.

Problem E. Freshman's Dream

Observe that the equation simplifies to $a + b = a \oplus (n + b)$. If n is odd, there can be no solution because the sides have opposite parities.

For even n, there is always a solution. We describe one possible construction. Let k be such that $\frac{n}{2} < 2^k$. We set $a = \frac{n}{2}$ and $b = 2^k - a$. Then $a + b = 2^k$. The right side is

$$a \oplus (n + b) = \frac{n}{2} \oplus \left(n + 2^k - \frac{n}{2}\right) = \frac{n}{2} \oplus \left(2^k + \frac{n}{2}\right).$$

Since $\frac{n}{2} < 2^k$ and 2^k has only one bit set, the two have no bits in common. Therefore, addition is equivalent to bitwise XOR in this case and we obtain

$$\frac{n}{2} \oplus \left(2^k + \frac{n}{2}\right) = \frac{n}{2} \oplus 2^k \oplus \frac{n}{2} = 2^k,$$

hence the sides are equal.

This pattern can be easily discovered by brute-forcing the lexicographically smallest answer for small n.

Problem F. When Anton Saw This Task He Reacted With 😩

The cross product is not associative, but it is linear. For given vectors $v = (v_x, v_y, v_z)$ and $w = (w_x, w_y, w_z)$,

$$v \times w = M_w v,$$

where

$$M_w = \begin{bmatrix} 0 & w_z & -w_y \\ -w_z & 0 & w_x \\ w_y & -w_x & 0 \end{bmatrix}.$$

We can also exchange the roles of v and w here and turn v into a matrix instead. But we can't expect to find a similar formula where both vectors are represented only as matrices, because that would imply associativity.

This observation gives us directly a solution for some simple special cases. For example, consider a tree that corresponds to the expression

$$((((a_1 \times a_2) \times a_3) \times \cdots) \times a_{n-1}) \times a_n,$$

i.e. the tree looks like this:

Using the observation above, the value of the root node is

$$M_{a_n} M_{a_{n-1}} \cdots M_{a_3} M_{a_2} a_1.$$

This is much simpler to work with because matrices are associative. To solve the problem in this case, we can use a segment tree to maintain the product $M_{a_n} M_{a_{n-1}} \cdots M_{a_3} M_{a_2}$. Changing a value is a standard point update, and after each query we have to calculate the product of all matrices and multiply it with a_1.

Now, let's attempt to generalize this solution to all trees.

- In each internal vertex, maintain M_v, where v is the value of the right child.
- After each update, we need to update these matrices. Observe that the number of matrices we have to update is equal to the number of ancestors that are right children. The algorithm to update all necessary matrices could look something like this.
  1. Suppose we are at a right child r with parent p and know the value of r.
  2. Convert the value of r to a matrix, store that matrix in p.
  3. Let l be the "leftmost leaf" reachable from p and q the lowest ancestor of p that is a right child or a root.
  4. Calculate the value of q by multiplying the value of l by all the matrices on the path from l to q.
  5. Now we are again at a right child. Repeat.

  If we start at a left child, start at step 3.

Step 4 can be done efficiently in $O(\log n)$ time. For each left leaf l, maintain a segment tree of matrices; the matrices on the path from l to the lowest ancestor of l that is a right child. Every vertex is on only one path, so this is fine. Other steps can also be done in $O(\log n)$ time. In total, the complexity of the entire update is $O(k \log n)$, where k is the number of right children on the path from the leaf to the root.

But the number of matrices we have to update and hence the number of steps depends on the number of ancestors that are right children. This algorithm only works on trees where the number of ancestors of u that are right children is small for every u.

By now you may have noticed the similarities between this algorithm and HLD. The problem would be solved if each left child would be a heavy child and each right child would be a light child. Observe that $v \times w = -w \times v$. Look at each vertex; if the left child is light, swap it with the right. Count the number s of such swaps made and multiply the final answer of each query by $(-1)^s$. Now we have complexity $O(q \log^2 n)$.

It is also possible to use HLD without this swapping trick because cross multiplying with w from the left also corresponds to a matrix. Then you have to use the right type of matrix on light edges depending on whether you are at a left child or a right child. But I think the trick of swapping children to make a "left-aligned" HLD makes the solution much cleaner.

In case you're seeing this problem on a platform/archive that doesn't support this, the "official" name of the problem is "When Anton Saw This Task He Reacted With 😩", with the emoji.

The emoji probably ended up being more trouble than it was worth. Polygon doesn't let you insert an emoji in the problem name, as and it turns out, LaTeX doesn't really like it either. You can insert emoji into LaTeX documents using the `emoji` package, however this is only supported with the LuaTeX compiler. So, to compile a statement PDF with 😩 in it, I had to download the contest package from Polygon and recompile the statements with LuaTeX. LuaTeX however doesn't seem to handle fonts the same way: the `\usepackage [T2A] {fontenc}` in Polygon's LaTeX template resulted in some very ugly fonts. Removing `[T2A]` mostly fixed it, but as you can see, the statements in this contest has a different font for problem and section headings than almost all contests.

Having to manually compile the statements has a very obvious drawback: if the statements have to be changed, I now have to make the changes twice: once in Polygon and once in my own .tex file.

And of course, emoji support varies elsewhere as well. If this contest ends up on the Codeforces gym, there will likely be no way to insert the emoji in the problem name. On the other hand, eolymp handled the emoji perfectly. This isn't too surprising: they likely have much less legacy code preventing weird Unicode symbols in names in the first place.

Problem G. LCA Counting

An upper bound for the answer is $2k-1$. This is reached if and only if the following condition holds for every vertex u:

> If there are colored vertices in the subtree of u, then u has at most 2 children whose subtrees contain colored vertices.

Any additional child which has colored vertices in its subtree decreases the answer by one. The answer is therefore

$$2k-1-\sum_{u \text{ has colored leaves in its subtree}} \max(0, (\#\text{ of children of } u \text{ that have colored leaves in their subtrees})-2).$$

We obtain the following alternative way to think about the problem:

For every vertex one can choose two child edges and paint them bold for free. Painting any additional edge bold costs 1 coin.

For each k, find the minimum cost of making at least k leaves reachable via bold edges from the root.

Let $\mathrm{dp}[u]$ be the maximum number of leaves freely reachable from u. It can be calculated by considering the dp-s of the children of u and taking the sum of the two largest ones. In particular, for a child v of u, we always have $\mathrm{dp}[v] \leq \mathrm{dp}[u]$.

At any point in time we can increase the number of reachable leaves by choosing a vertex u such that u is unreachable, its parent p is reachable, and painting the edge up bold. This increases the number of reachable leaves by $\mathrm{dp}[u]$ and costs 1 coin.

Notice that although performing such an operation may open up additional possible operations, they all give fewer new leaves per coin. Therefore the most efficient operation is always available and we can use a greedy algorithm.

Problem H. Minimum Cost Flow²

Denote the edge set by E. For an edge $e = (u, v)$ and a flow f, denote $f_{uv} = f_e$ and $f_{vu} = -f_e$. Similarly, denote $c_{uv} = c_{vu} = c_e$. As there are no parallel or antiparallel edges, this is well-defined.

We first formulate a necessary condition for a flow f to be optimal.

Claim 4. *Let f be an optimal flow and $C = v_0 v_1 \dots v_k$ (where $v_k = v_0$) be a cycle in the underlying undirected graph. Then*

$$r(C) := c_{v_0v_1} f_{v_0v_1} + c_{v_1v_2} f_{v_1v_2} + \cdots + c_{v_{k-1}v_k} f_{v_{k-1}v_k} = 0. \tag{1}$$

Proof. Let f be a flow with value 1 and $C = v_0 v_1 \dots v_k$ a cycle such that this condition doesn't hold, i.e.

$$c_{v_0v_1} f_{v_0v_1} + c_{v_1v_2} f_{v_1v_2} + \cdots + c_{v_{k-1}v_k} f_{v_{k-1}v_k} = a$$

for some $a \neq 0$. Define Δ_e for all edges as follows, with $e = (u, v)$:

$$\Delta_e = \begin{cases} 1 & \text{if } (u,v) \text{ appears in the cycle;} \\ -1 & \text{if } (v,u) \text{ appears in the cycle;} \\ 0 & \text{otherwise.} \end{cases}$$

Now for all $\lambda \in \mathbb{R}$, $f + \lambda\Delta$ is also a flow with value 1. The cost of $f + \lambda\Delta$ is

$$\sum_{e \in E} c_e (f_e + \lambda\Delta)^2 = \sum_{e \in E} c_e f_e^2 + 2\lambda \sum_{i=0}^{k} c_{v_i v_{i+1}} f_{v_i v_{i+1}} + \lambda^2 \sum_{i=0}^{k} c_{v_i v_{i+1}} = \text{cost}(f) + 2\lambda a + \lambda^2 b$$

for some known $a \neq 0$ and $b \in \mathbb{R}$. At $\lambda = 0$, the function $\lambda \mapsto 2\lambda a + \lambda^2 b$ has value 0 and derivative $2a \neq 0$, so it is possible to choose a λ such that $2\lambda a + \lambda^2 b < 0$. Then $f + \lambda\Delta$ has lower cost than f. □

We have now shown that a flow can only be optimal if for all cycles, the sum above is 0. But "all cycles" is actually an overkill. There is a set of "basis cycles", and if the sum is 0 on these basis cycles, it is 0 on every cycle. To generate this set of basis cycles, take a spanning tree of the graph (for implementation, it is convenient if this spanning tree is the DFS tree). For every edge uv not in the spanning tree, take the unique path $u \rightsquigarrow v$ in the spanning tree. That makes a cycle.

To prove that it's sufficient to make (1) equal to 0 for only basis cycles, consider an arbitrary cycle C and suppose that (1) is equal to 0 for all basis cycles. Let e be a non-span edge that C visits, and let C_e be the basis cycle corresponding to that edge. Depending on the orientation of e in C, either subtract or add $r(C_e)$ to $r(C)$. This turns the expression (1) for C to the same expression for another cycle C' that uses non-span edges in total a smaller number of times. And since $r(C_e) = 0$, we get that $r(C) = r(C')$. Apply the same procedure to C' until you get a cycle C'' that doesn't use any non-span edges at all. C'' is either empty or just walks back and forth in the spanning tree. Either way, clearly $r(C'') = 0$, which shows that also $r(C) = 0$.

The task statement provides $n - 1$ linear equations; each basis cycle gives us another one. There are $m - (n - 1)$ basis cycles; in total we get m linear equations in m variables. It can be proven that this system of linear equalities has an unique solution in $\mathbb{Q}$. Solve it with Gaussian elimination.

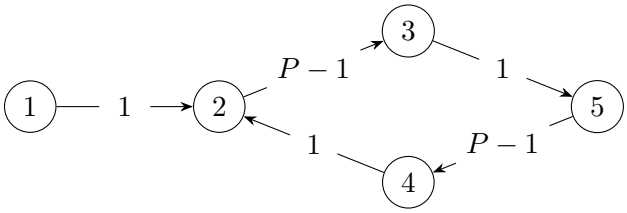
Proving that this system of linear equations has an unique solution is annoying, but doable. If you look at the physical interpretation below, you can find some keywords that you can search for.

About the modulo

Now, we need to talk about modulo. To be clear, I don't really intend for you to think about this during the contest. I think that the best way to solve this is to just yolo it and hope everything works out. It's annoying that we have to talk or think about this at all.

Above, we kind of implicitly did everything over $\mathbb{Q}$ (or $\mathbb{R}$). But the problem statement actually asks you to print the optimal cost of the flow in $\mathbb{Z}_P$, where $P = 998\,244\,353$, with a promise that there exists an optimal flow that can actually be represented in $\mathbb{Z}_P$. Indeed, if you actually implemented Gaussian elimination over $\mathbb{Q}$, you would quickly reach a point of overflow.

Thus, you must also do the Gaussian elimination over $\mathbb{Z}_P$, as we always do. But this comes with complications. The problem statement tells you that the optimal flow can be represented in $\mathbb{Z}_P$. This guarantees that the system of linear equations always has a solution. But what if there are multiple solutions? We non-proved above that the system always has an unique solution over $\mathbb{Q}$. One might get the impression that it also always has an unique solution over $\mathbb{Z}_P$. This is not so! Consider this example:



The system of linear equations looks like this:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & -1 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 \\ 0 & P-1 & 1 & P-1 & 1 \end{bmatrix} \cdot f = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{2}$$

Over $\mathbb{Q}$, the matrix on the left is invertible and all is well. But modulo P, the matrix is clearly degenerate: the fifth row is just a sum of the third and fourth. The set of solutions to (2) in $\mathbb{Z}_p$ is

$$\begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \lambda \begin{bmatrix} 0 \\ -1 \\ -1 \\ 1 \\ 1 \end{bmatrix}, \qquad \lambda \in \mathbb{Z}_P.$$

Miraculously, no matter which λ we pick, the actual sum $\sum c_e f_e^2$ is always the same. This is not a coincidence.

To see why, let's first look at an alternative solution to the problem. The space of all valid flows is an affine subspace of $\mathbb{R}^m$ with dimension $\mathbb{R}^{m-n+1}$, of the form

$$v_0 + \operatorname{span}(v_1, v_2, \ldots, v_k),$$

where $k = m - n + 1$. You can solve for the first $n - 1$ rows of the previous linear system to find it, but we actually already know a nice set of vectors $v_0, \ldots, v_{m-n+1}$: let v_i be the circulation of 1 unit of flow through the i-th basis cycle and take v_0 to be the unique path from the source to the sink in the spanning tree we used to generate the basis cycles.

For a given flow $v_0 + \cdots + x_k v_k$, we have that the amount of flow on edge e is

$$f_e = v_{0e} + \sum_{i=1}^{k} x_i v_{ie}.$$

Now, if you expand $\sum_e c_e f_e^2$, you get that it is a sum, where each term is of the form $x_i x_j$ or x_i for some i (and possibly j), and each term has a known coefficient. We can write it as

$$F(x) = \frac{1}{2} x^\top A x + b^\top x + c \tag{3}$$

for some symmetric matrix $A \in \mathbb{R}^{k \times k}$, vector $b \in \mathbb{R}^k$ and scalar c. We must find a vector x that minimizes $F(x)$.

We'll do it using partial derivatives. If you do the math, you see that setting the partial derivative of $F(x)$ to 0 with respect to each x_i yields $Ax = b$. Again, this has an unique solution in $\mathbb{Q}$, but might have multiple solutions in $\mathbb{Z}_P$. However, suppose that $Ax \equiv_P b \equiv_P Ay$. Then,

$$x^\top Ax \equiv_P x^\top Ay \equiv_P y^\top Ay$$

and

$$b^\top x \equiv_P y^\top Ax \equiv_P y^\top b \equiv_P b^\top y.$$

Therefore, even though there may be multiple solutions to the system of linear equations, they all give the same result, so we can pick any.

You may have noticed that this solution is actually very similar to the first solution presented here. We used basis cycles in both cases and we in fact implicitly used partial derivatives with respect to the basis cycles in the first solution. To conclude, let's formalize the connection.

In the first half, we found a solution f to the system

$$\begin{bmatrix} C \\ D \end{bmatrix} f \equiv_P \begin{bmatrix} e \\ 0 \end{bmatrix},$$

where C has $n-1$ rows and D has $m-n+1$ rows. Because $Cf \equiv_P e$, we know that it satisfies the node law and can thus be written $f \equiv_P v_0 + \sum v_i y_i$, or $f \equiv_P v_0 + Vy$ for short. It's possible to show that $v_0, v_1, \ldots, v_k$ are still linearly independent even modulo P, and that the rows of A are also still linearly independent even modulo P. Now, notice that

$$D_{ie} = c_e v_{ie}, \qquad A_{ij} = \sum_e c_e v_{ie} v_{je}, \qquad b_i = \sum_e c_e v_{ie} v_{0e},$$

which implies that $A = DV$ and $b = Dv_0$. Now

$$Ay \equiv_P DVy \equiv_P D(f - v_0) \equiv_P Dv_0 \equiv_P b,$$

which proves that $\frac{1}{2} y^\top Ay + b^\top y + c$ is equivalent modulo P to the actual solution of the problem. Finally, you can check that

$$f^\top \Delta_c f \equiv_P \frac{1}{2} y^\top Ay + b^\top y + c,$$

i.e. the directly calculated weighted sum of squares is the same as the solution to the problem we calculated.

This is a physics problem. We look at the graph as an electrical circuit, where an edge with cost c_e corresponds to a resistor with resistance c_e. The cost function is then in fact the power dissipated by a current f_e. The principle of least energy then claims that of all the possible steady states satisfying the voltage and current constraints, the real steady state dissipates the least amount of power, which is equivalent to minimizing the total cost function. Therefore, the flow minimizing this cost function must be a current. This fact is known as Thomson's principle. Now, currents always satisfy Kirchhoff's circuit law, i.e. (1). The other linear equations are exactly Kirchhoff's node law. From there we get the system of linear equations we used to solve the problem.

Sorry about the long editorial for this problem. I really wanted to show that introducing the modulo doesn't screw the solution up. As you can see, it is not completely obvious because the system of linear equations can be degenerate, but this is salvaged by the fact that the sum of squares modulo P doesn't actually change.

It is much easier to prove this with the second solution, but I wanted to show the first solution because of its neat connection to to physics. I still feel like my proof must have taken a very circuitous route. Maybe you can come up with a direct proof that the first solution is always correct?

Problem I. Rebellious Edge

In a directed spanning tree (DST) there is exactly one incoming edge for each non-root vertex. Choosing a DST means choosing one incoming edge for each non-root vertex such that there is a path from 1 to every vertex.

The directed minimum spanning tree either uses the backwards edge or it doesn't. We will calculate both cases separately.

Ignoring the backwards edge. Delete the backwards edge. Choose the cheapest incoming edge for each vertex. If for some vertex there isn't any, then it isn't possible to construct a DST without the backwards edge. Otherwise, there always exists a path $1 \rightsquigarrow v$ for all v: the incoming edge to v comes from a vertex with a smaller index, the incoming edge to that vertex comes from a vertex with an even smaller vertex, and so on. Eventually, we must reach 1. Also, this is clearly the cheapest choice we can make without the backwards edge.

Forcing the backwards edge. Let $u \leftarrow v$ be the backwards edge. We again have to choose an incoming edge for each vertex other than 1 and u.

The result will be a DST if and only if there is a path $1 \rightsquigarrow v$. Indeed, for any vertex w, the incoming edge either comes from a vertex with a smaller index or from vertex v. Iterating as in the previous case, we see that we eventually either reach 1 or v, and if we reach v, then there is a path from 1 as well.

For each vertex other than 1 and u, look at its incoming edges. Let c be the cost of the cheapest incoming edge. Add c to the answer and subtract c from all incoming edges: we have to pay c no matter what.

The set of chosen edges must contain a path $1 \rightsquigarrow v$, so the amount we add to the answer after that must be at least the length of the shortest path $1 \rightsquigarrow v$. But choosing the shortest path is also sufficient: for each vertex not in the path we can choose the cheapest incoming edge.

And of course, as the statement mentions: you can also solve this problem using a general-purpose directed minimum spanning tree algorithm if you have one in your library. It is probably faster to do it this way if you can copy and paste, but probably not if you have to manually retype it from a team notebook. In Osijek, around one in four teams submitted a solution with heavy machinery, the rest had elementary solutions.

In any case, this is meant to be a rather easy warm-up problem, so I don't see a problem with such solutions being possible.

Problem J. 'Ello, and What Are You After, Then?

First, let's replace the condition of "you never go below 0 points" with "the expected point gain is nonnegative". This new condition is weaker, so the maximum possible expected XP gain doesn't decrease. It's possible to show that they are in fact equal.

A *pure strategy* is a tuple $\sigma = (i, S, B)$ that describes your action on one turn. Here:

- i is the index of the slayer master you pick;
- $S \subset [m_i]$ is the set of tasks that you will skip, if offered;
- $B \subset [m_i]$ is the set of tasks that you will block.

We assume that $|B| \le b$, $S \cap B = \varnothing$ and $S \cup B \neq [m_i]$. A pure strategy has two important properties:

- The expected XP gain per minute $e(\sigma)$. It can be calculated as follows:

$$e(\sigma) = \frac{\sum_{j \notin B, j \notin S} f_{ij} t_{ij} e_{ij}}{\sum_{j \notin B, j \notin S} f_{ij} t_{ij}}.$$

- The expected point gain per minute $p(\sigma)$. It can be calculated as follows.

$$p(\sigma) = \frac{c \sum_{j \notin B, j \notin S} f_{ij} - s \sum_{j \in S} f_{ij}}{\sum_{j \notin S, j \notin B} f_{ij} t_{ij}}.$$

We now make yet another modification to the formalization of the problem. Instead of specifying an action for every iteration, we use a probability distribution of pure strategies: on each turn, we pick a pure strategy according to this distribution. This is equivalent: since we removed the restriction on the number of points never going to zero, all steps are completely independent of one another. As $q \to \infty$, we can approximate our probability distribution with a finite number of elements, and from any infinite sequence of pure strategies we can derive a probability distribution.

A probability distribution (or convex combination) of pure strategies is called a *mixed strategy*. We write mixed strategies as expressions of the form

$$\lambda_1 \sigma_1 + \lambda_2 \sigma_2 + \cdots + \lambda_r \sigma_r$$

where $\lambda_1 + \cdots + \lambda_r = 1$. Note that it is fine to only consider finite convex combinations since there is only a finite number of possible pure strategies (less than 3^{m_i} for the i-th slayer master).

We write $e\left(\sum_k \lambda_k \sigma_k\right)$ and $p\left(\sum_k \lambda_k \sigma_k\right)$ for respectively the expected XP gain per minute and expected point gain per minute if the mixed strategy $\sum_k \lambda_k \sigma_k$ is used. Let $\sigma_k = (i_k, S_k, B_k)$. It can be shown that

$$e\left(\sum_k \lambda_k \sigma_k\right) = \frac{1}{\sum \lambda_k T(\sigma_k)} \sum_k \lambda_k T(\sigma_k) e(\sigma_k), \qquad p\left(\sum_k \lambda_k \sigma_k\right) = \frac{1}{\sum \lambda_k T(\sigma_k)} \sum_k \lambda_k T(\sigma_k) p(\sigma_k),$$

where $T(\sigma_k)$ is the expected time of completing one step of pure strategy σ_k.

This implies that if we have a set of pure strategies $\sigma_1, \ldots, \sigma_r$ with parameters

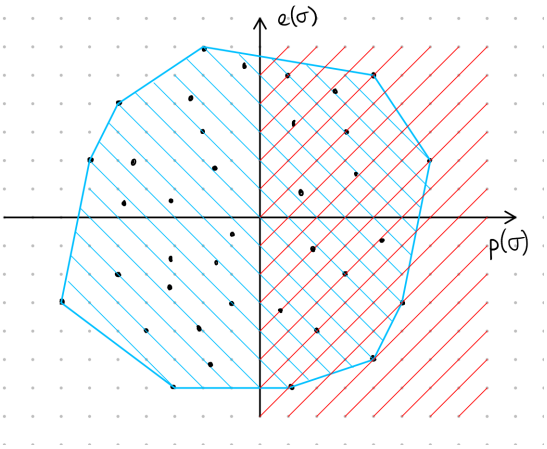
$$(p(\sigma_1), e(\sigma_1)), \ldots, (p(\sigma_r), e(\sigma_r)),$$

then we can create a mixed strategy τ with

$$p(\tau) = \sum \lambda_k p(\sigma_k), \qquad e(\tau) = \sum \lambda_k e(\sigma_k)$$

for $\lambda_1 + \cdots + \lambda_r = 1$. This is done by rescaling the λ_k-s to match the $T(\sigma_k)$-s. This is why we consider "points per time" instead of, say, "points per task": with points per task it wouldn't work (at least so easily).

This allows us to imagine the strategies (pure or mixed) as points $(p(\sigma), e(\sigma))$ on the plane. If we plot all pure strategies, then the achievable mixed strategies correspond precisely to the points in the convex hull of all pure strategies. We have to find a point (p, e) in the convex hull such that $p \geq 0$ and e is maximized. Of course, there is an enormous number of pure strategies, so this calculation can't be done naively.

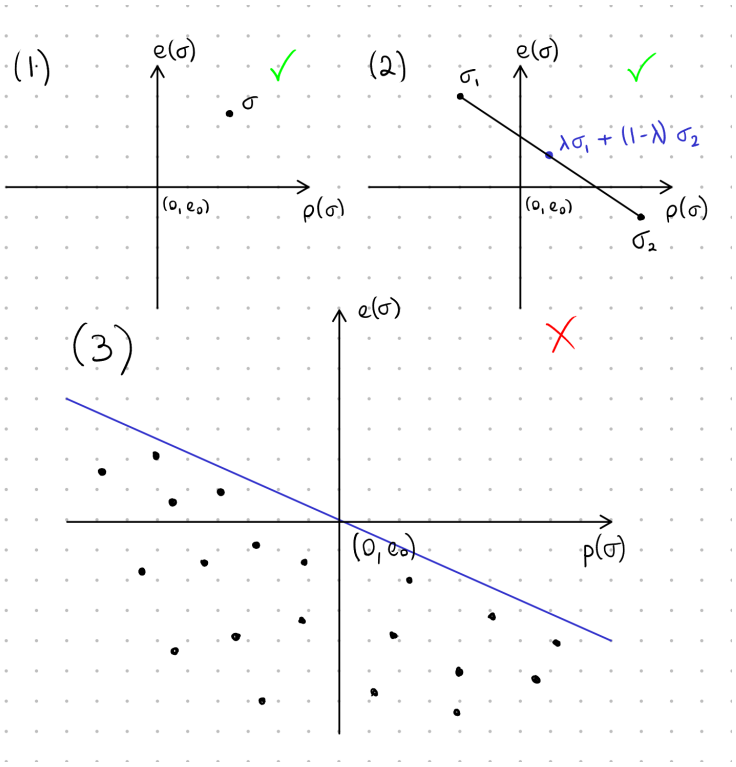


Binary search on e. We have to develop an oracle for answering the question "is there a strategy with $p \geq 0$ and $e \geq e_0$". Center the plot at $e = e_0$ (we will refer to $(0, e)$ as "the origin"). We have to check if there is a strategy in the top-right quadrant. There are two ways this could happen:

1. There is a pure strategy σ in the top-right quadrant.
2. There is a pure strategy σ_1 in the top-left quadrant and a pure strategy σ_2 in the bottom-right quadrant such that the line segment connecting σ_1 and σ_2 passes through the top-right quadrant.

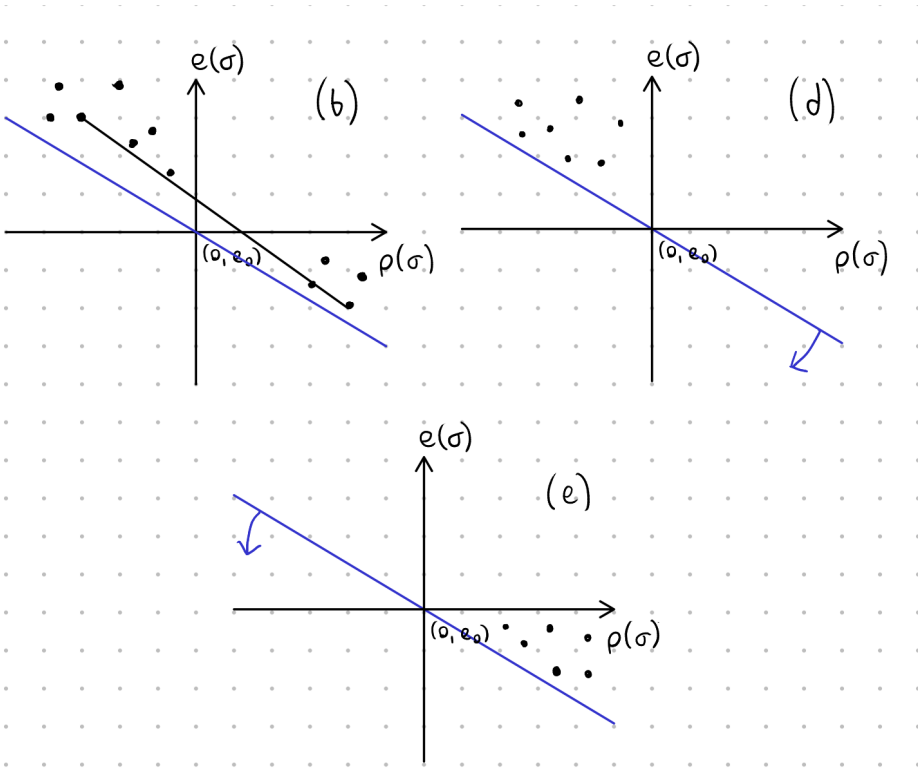
And there is also a necessary and sufficient condition for this to be impossible:

3. There is a line through the origin such that all pure strategies lie below this line.



We propose the following algorithm, a variation of binary search, for either providing a line as in case 3 or strategies as in cases 1 and 2.

1. Set $l = \varepsilon$ and $r = M$ for a very small ε and a very large M.
2. Let $\mu = \frac{l+r}{2}$ (or $\sqrt{lr}$).
3. We are curious about pure strategies that are above the line through the origin with normal vector $(\mu, 1)$. That line is described by $\mu p(\sigma) + e(\sigma) = e_0$. For each slayer master, calculate one pure strategy σ such that $\mu p(\sigma) + e(\sigma) \geq e_0$. We will call such pure strategies *visible*.
4. There are now five cases.
   (a) There is a visible pure strategy in the top-right quadrant. This is condition 1. Output "yes".
   (b) There is a visible pure strategy in the top-left quadrant and a visible pure strategy in the bottom-right quadrant. Visible pure strategies are above the line, so the line segment connecting these two must visit the top-right quadrant. This is condition 2. Output "yes".
   (c) There are no visible pure strategies. This means that we could not produce a pure strategy above the line for any slayer master. Thus there are no pure strategies above the line. This is condition 3. Output "no".
   (d) There is a visible pure strategy in the top-left quadrant, but not in the bottom-right quadrant. Set $r = \mu$ and go to step 2. Intuitively, we are turning the line clockwise.
   (e) There is a visible above the line in the bottom-right quadrant, but not in the top-left quadrant. Set $l = \mu$ and go to step 2. Intuitively, we are turning the line counterclockwise.



Notice that this algorithm has no stopping condition. It might be that we go on indefinitely with smaller and smaller segments (l, r), alternating between cases 4d and 4e. In particular, this will always be the case if there is only one slayer master, with pure strategies in top-left and bottom-right, but not in top-right.

Because of the way we calculate visible strategies, the ray $(0, \infty)$ can be partitioned into a finite number of contiguous ranges such that in each range, the set of visible strategies is the same no matter which μ in the range is chosen. This means that there is some μ and $\varepsilon > 0$ such that in the range $(\mu - \varepsilon, \mu)$ there are visible pure strategies only in the bottom-right quadrant, and in the range $(\mu, \mu + \varepsilon)$ there are visible pure strategies only in the top-left quadrant. Claim 5 in the appendix then provides a mixed strategy in the top-right quadrant. Thus, if we get into an infinite loop of ever-tightening segments, this is also a "yes". The author's solution handles this case differently, by maintaining "best" pure strategies in both quadrants.

We need an algorithm that calculates a pure strategy above the line for a given slayer master, or reports that there is none. The parameter μ can be interpreted as the value of a slayer point in terms of XP. For a given strategy σ, we have that

$$\mu p(\sigma) + e(\sigma) = \frac{\sum_{j \notin B, j \notin S} f_{ij} t_{ij} e_{ij} + \mu c \sum_{j \notin B, j \notin S} f_{ij} - \mu s \sum_{j \in S} f_{ij}}{\sum_{j \notin B, j \notin S} f_{ij} t_{ij}}.$$

Requiring $\mu p(\sigma) + e(\sigma) \geq e_0$ is then equivalent to

$$\sum_{j \notin B, j \notin S} f_{ij} t_{ij} (e_{ij} - e_0) + \mu c \sum_{j \notin B, j \notin S} f_{ij} - \mu s \sum_{j \in S} f_{ij} \geq 0.$$

Therefore:

- Tasks we complete contribute $f_{ij} t_{ij} (e_{ij} - e_0) + \mu c f_{ij}$.
- Tasks we skip contribute $-\mu s f_{ij}$.
- Tasks we block contribute 0.

Assuming we don't block a task, the choice of whether to skip it or not is now clear. Make the choice for each task, then block $\min(b, m_i - 1)$ tasks with the worst contribution. If the resulting strategy has $\mu p(\sigma) + e(\sigma) \geq e_0$, return it; otherwise report that there is no pure strategy above the line.

This concludes the solution. The complexity is $O(m \log m \log^2 C)$, where $m = \sum_{i=1}^{n} m_i$. $m \log m$ comes from sorting the tasks by contribution in the final step, the two other logs come from the two binary searches. The $\log m$ factor can be removed by using quickselect, but this is not necessary to get Accepted.

Appendix: Omitted proofs

Claim 5. *Suppose there are some values $\mu_1 < \mu_2 < \mu_3$ such that:*

- *for $\mu \in (\mu_1, \mu_2)$, there are visible pure strategies in the bottom-right quadrant, but not in the top-left;*
- *for $\mu \in (\mu_2, \mu_3)$, there are visible pure strategies in the top-left quadrant, but not in the bottom-right.*

Then there exists a mixed strategy in the top-right quadrant.

Proof. Again, because of the way we calculate the visible strategy for each slayer master, the ray $(0, \infty)$ can be partitioned into a finite number of contiguous ranges such that the set of visible strategies is the same within a range. Therefore, after possibly redefining μ_1 to be slightly higher, there must be some pure strategy σ_1 such that σ_1 is above the line in the bottom-right quadrant for all $\mu \in (\mu_1, \mu_2)$, i.e.

$$\begin{aligned} \mu p(\sigma_1) + e(\sigma_1) &\geq e_0 && \forall \mu \in (\mu_1, \mu_2) \\ e(\sigma_1) &\leq 0 \\ p(\sigma_1) &\geq 0. \end{aligned}$$

Similarly, we obtain a pure strategy σ_3 such that

$$\begin{aligned} \mu p(\sigma_3) + e(\sigma_3) &\geq e_0 && \forall \mu \in (\mu_2, \mu_3) \\ e(\sigma_3) &\geq 0 \\ p(\sigma_3) &\leq 0 \end{aligned}$$

Affine functions are continuous, so we have that

$$\begin{aligned} \mu_2 p(\sigma_1) + e(\sigma_1) &\geq e_0 \\ \mu_2 p(\sigma_3) + e(\sigma_3) &\geq e_0, \end{aligned}$$

i.e. σ_1 and σ_3 are pure strategies above the line for μ_2, which gives us a mixed strategy in the top-right quadrant. □