Love Polygon ♡

In essence, we have a directed graph where each vertex has exactly one outgoing arc. We are asked to redirect some of the arcs so that each vertex is in a "pair" and we are asked to do it with the minimum number of redirects. This type of graph, called a "functional graph", inevitably takes the form where each connected component is a directed cycle with directed trees branching off of it.

Note that some sort of solution is always possible if N is even, and always impossible if N is odd. Therefore, if N is odd we can immediately output `-1`. The following explanations deal with the case where N is even.

Subtask 1

Let G be the set of all people; let S be a subset of G and let $T = G \setminus S$. In order for S to be the set of people ♡-shot in any solution, the following conditions must hold:

- If we remove the edges originating from people in S, all connected components in the resulting graph must have at most 2 vertices. This is because the final graph can be constructed by only adding edges to this graph, and in the final graph all connected components have 2 vertices.
- No person who loves themselves can be in T, because then they will love themselves in the final arrangement, which is not permitted.

If these conditions hold, S can be used to construct a solution by pairing off the people in connected components of size 2 and pairing everyone else off randomly. $|S|$ arrows will be used. Therefore, a set S can be the set of people shot if and only if those conditions are met.

To solve the problem, we can iterate over all sets S and check for this property, then pick the smallest fitting set S^* and output $|S^*|$. There are 2^N subsets of G, checking each set can be done in $\mathcal{O}(N)$ time. The complexity is therefore $\mathcal{O}(N2^N)$.

An alternative approach is to use dynamic programming on subsets in $\mathcal{O}(N2^N)$ time.

Subtask 2

In order for everyone to be loved by someone, everyone must be loved by exactly one person. In this subtask, each connected component of the graph takes the form of a cycle. Let's process each component separately, let C be the number of vertices in the component. It is easy to see that if the component has an even number of vertices, then it is optimal to pair each vertex off with one of its neighbours, using $\frac{C}{2}$ arrows, unless the component has 2 vertices, in which case no arrows are needed. On the other hand, if the component has an odd number of vertices, then it is optimal to pair each vertex but one off with one of its neighbours, pairing the last vertex off with a vertex outside of that component, using $\lfloor \frac{C}{2} \rfloor + 1$ arrows. The problem can be solved in $\mathcal{O}(N)$ time by counting the vertices in each component.

Subtask 3

Since there are no "love polygon", that must mean the cycle in each connected component of the graph consists of one character loving themselves. This means each connected component takes the form of a directed tree with all edges directed towards the root.

We call a set of vertices T in the forest *lucky* if and only if:

- For each vertex in T, its parent is not in T;
- For each vertex in T, none of its children are in T;
- For each vertex in T, none of its siblings are in T.

Let S^* be the set of vertices that are not shot with a love arrow in the optimal solution. Then S^* is clearly a lucky set: any character in that set will end up in a relationship with the character they initially loved; that is, they will be paired off with their parent in the tree. Let v be in S^*, then:

- The parent of v must be shot with an arrow to love v. Therefore the parent of v is not in S^*.
- All children of v must be shot with an arrow, otherwise they would end up in a pair with v, but we know v will be paired off with its parent. Therefore no children of v are in S^*.
- All siblings of v must be shot with an arrow, otherwise they would end up in a pair with the parent of v, but we know the parent of v will be paired off with v. Therefore no siblings of v are in S^*.

Hence, S^* is a lucky set. Note that the number of love arrows required is $N - |S^*|$. Furthermore, given any lucky set S that doesn't contain roots, we can construct a solution using $N - |S|$ arrows by pairing the vertices in S off with their parents and pairing everyone else off randomly. Therefore, if we define R as the set of lucky sets that don't contain roots, the solution to the problem is

$$\min_{S \in R} (N - |S|) = N - \max_{S \in R} |S|. \tag{1}$$

Our task is therefore to calculate the size of the largest lucky set that doesn't contain any roots. This can be done using dynamic programming.

Let L_v denote the set of initial lovers (children) of vertex v, excluding vertex v itself if v is a root. Define:

- $\text{mls}(v)$ to be the size of the maximum lucky set within the subtree of v whose member v itself is not.
- $\overline{\text{mls}}(v)$ to be the size of the maximum lucky set within the subtree of v whose member v itself is.

The size of the largest lucky set is then $\sum \text{mls}(r)$ over all roots r. If vertex v is a leaf, then clearly $\text{mls}(v) = 0$ and $\overline{\text{mls}}(v) = 1$. Let v be a nonleaf vertex. Then the equation

$$\overline{\text{mls}}(v) = 1 + \sum_{u \in L_v} \text{mls}(u) \tag{2}$$

clearly holds. Lucky sets within the subtree of v that don't contain the vertex v can either:

- Not contain any of the children of v. The largest of them has size $\sum_{u \in L_v} \text{mls}(u)$.
- Contain exactly one of the children of v. The largest lucky set containing w, a child of v, has size $\left(\sum_{u \in L_v} \text{mls}(u)\right) + \overline{\text{mls}}(w) - \text{mls}(w)$.

All those kinds of lucky sets exist, the largest of them has therefore size

$$\begin{aligned}
&\max\left\{\sum_{u\in L_v}\operatorname{mls}(u),\quad \max_{w\in L_v}\left(\left(\sum_{u\in L_v}\operatorname{mls}(u)\right)+\overline{\operatorname{mls}}(w)-\operatorname{mls}(w)\right)\right\}=\\
&=\max\left\{\sum_{u\in L_v}\operatorname{mls}(u),\quad \sum_{u\in L_v}\operatorname{mls}(u)+\max_{w\in L_v}\left(\overline{\operatorname{mls}}(w)-\operatorname{mls}(w)\right)\right\}=\\
&=\max\left\{\overline{\operatorname{mls}}(v)-1,\quad \overline{\operatorname{mls}}(v)-1+\max_{w\in L_v}\left(\overline{\operatorname{mls}}(w)-\operatorname{mls}(w)\right)\right\}=\\
&=\max\left\{0,\quad \max_{w\in L_v}\left(\overline{\operatorname{mls}}(w)-\operatorname{mls}(w)\right)\right\}+\overline{\operatorname{mls}}(v)-1.
\end{aligned}$$

Therefore,

$$\operatorname{mls}(v)=\max\left\{0,\quad \max_{w\in L_v}\left(\overline{\operatorname{mls}}(w)-\operatorname{mls}(w)\right)\right\}+\overline{\operatorname{mls}}(v)-1. \tag{3}$$

To sum it up, for any vertex v:

$$\operatorname{mls}(v)=\begin{cases}0, & \text{if } v \text{ is a leaf;}\\ \max\left\{0,\quad \max_{w\in L_v}\left(\overline{\operatorname{mls}}(w)-\operatorname{mls}(w)\right)\right\}+\overline{\operatorname{mls}}(v)-1 & \text{otherwise}\end{cases} \tag{4}$$

and

$$\overline{\operatorname{mls}}(v)=\begin{cases}1, & \text{if } v \text{ is a leaf;}\\ 1+\sum_{u\in L_v}\operatorname{mls}(u) & \text{otherwise}\end{cases} \tag{5}$$

hold. Using those recurrences, we can iterate over all connected components of the graph, do a depth-first search on them and calculate the values of $\overline{\operatorname{mls}}(v)$ and $\operatorname{mls}(v)$ for all vertices v. Finally, we can output $N-(\sum \operatorname{mls}(r))$ over all roots r. Each vertex needs to be traversed only once, which gives a runtime of $\mathcal{O}(N)$.

Subtask 4

The subtask is solved similarly to subtask 3. We will process each connected component separately. If the cycle of the current component consists of just one character, we will process it the same way as in subtask 3. If the cycle is longer, we pick one arbitrary character. In the optimal solution, that character either is in a relationship with the person they love, or isn't. In both cases we can eliminate some arcs from the component so that the component becomes a forest of directed trees. Using the solution from subtask 4, we can calculate the number of love arrows needed in both situations and pick the better one. This solves the subtask in $\mathcal{O}(N)$. ♡

Martian DNA

Subtask 1

Let $s=s_1s_2\ldots s_N$ denote the DNA string. Given a substring $t=s_is_{i+1}\ldots s_j$, we can check in time $\mathcal{O}(N\cdot R)$ whether it contains sufficiently many of all nucleobases, by simply looping over each required nucleobase and counting how many occurrences there are of that nucleobase in the substring. Suppose we have a function `HasEnough`(t) which returns `true` if the substring t

contains sufficiently many of all nucleobases. Then we can find the optimal substring by looping over all substrings (there are $\binom{N+1}{2} = \mathcal{O}(N^2)$ of them) and picking the shortest substring t such that $\texttt{HasEnough}(t) = \texttt{true}$. The time complexity of this algorithm is $\mathcal{O}(N^3 R)$.

Subtask 2

If we do some precomputations we can actually compute the function $\texttt{HasEnough}(t)$ in time $\mathcal{O}(R)$. The idea is to use *prefix sums*, which means that we start by computing, for each required nucleobase, how many times that nucleobase occurs in each prefix of the DNA, which allows us to quickly compute how many times that nucleobase occurs in any substring. If we let $\texttt{count}_c(m)$ denote the number of times the nucleobase c occurs in the prefix $s_1 \dots s_m$, then we can compute how many times c occurs in the substring $t = s_i s_{i+1}, \dots, s_j$ as $\texttt{count}_c(j) - \texttt{count}_c(i-1)$, which only takes time $\mathcal{O}(1)$ per type of nucleobase. We can therefore check if we have enough of all required nucleobases in time $\mathcal{O}(R)$.

The precomputation can be done in time $\mathcal{O}(NR)$, and then we can use the $\mathcal{O}(R)$ implementation of `HasEnough` to test all substrings in time $\mathcal{O}(N^2 R)$.

Subtask 3

The key realization needed to solve subtask 3, where N may be as large as 100000, is that we don't actually have to check all substrings. Instead, it would be sufficient to know, for each $i \in \{1, \dots, N\}$, what is the minimum j such that the substring $s_i \dots s_j$ contains sufficiently many of all nucleobases. We can find j using *binary search*, because whenever $\texttt{HasEnough}(s_i \dots s_{j'})$ returns `true` then we get an upper bound $j \leq j'$, and whenever $\texttt{HasEnough}(s_i \dots s_{j'})$ returns `false` we get a lower bound $j \geq j'$. If we also compute `HasEnough` using prefix sums then the algorithm runs in time $\mathcal{O}(NR + NR \log N) = \mathcal{O}(NR \log N)$.

Subtask 4

To solve the problem in time $\mathcal{O}(N)$, we can use an approach based on *two pointers*. At all times we keep track of a substring $s_\ell \dots s_r$. If $\texttt{HasEnough}(s_\ell \dots s_r) = \texttt{false}$ then the substring is too small, so we make it bigger by setting $r \leftarrow r + 1$. If, on the other hand, $\texttt{HasEnough}(s_\ell \dots s_r) = \texttt{true}$ then we might be able to make the substring smaller, so we set $\ell \leftarrow \ell + 1$. This way we only have to check $\mathcal{O}(N)$ different substrings.

The problem is that precomputing all the prefix sums needed for `HasEnough` takes time $\Theta(NR)$, which is too slow. However, we can exploit the fact that when we increment ℓ or r, the substring $s_\ell \dots s_r$ doesn't change very much, so we can keep track of:

1. How many nucleobases there are of each type in the substring $s_\ell \dots s_r$.
2. How many of the required nucleobases that have too few occurrences in the substring $s_\ell \dots s_r$.

Whenever we increment the left pointer ℓ, we first decrement the number of nucleobases of type s_ℓ, and if the number of nucleobases of that type becomes too small we also increment the number of nucleobases with an insufficient number of occurrences.

Similarly, whenever we increment the right pointer r, we first increment the number of nucleobases of type s_{r+1}, and if the number of nucleobases of that type becomes exactly the number we need then we also decrement the number of nucleobases with an insufficient number of occurrences.

If we keep track of the number of nucleobases with too few occurrences, then we can implement `HasEnough`$(s_\ell \dots s_r)$ by simply checking if there are *no* nucleobases with too few occurrences, which is a constant time operation. Hence this algorithm runs in $\mathcal{O}(N)$.

Worm Worries

This problem is really three problems in one, for the cases of one, two and three dimensions. Most solutions work across any number of dimensions, but to get an optimal number of queries (for testgroups 2, 4 and 6), the three cases required separate solutions.

1. For one dimension, we can first try something like a binary search. Let's say we have a grid of size $N \times 1 \times 1$. We can reduce the problem to a smaller one as follows: ask for the value of the middle two points, say $H[N/2]$ and $H[N/2+1]$. If $H[N/2] \geqslant H[N/2+1]$, then we can reduce the problem to finding a local maximum of the first half of the array (including element $N/2$), otherwise to the second half.

   However, this uses 40 queries for $N = 1\,000\,000$, while testgroup 2 requires at most 35. Getting to 35 queries another approach, based on the idea of *golden section search*.

   Say we want to find a local maximum in the interval $[a, b]$. If $a = b$, then we just return a. Otherwise, we might query two points $m, m+1$, say in the middle of the interval, and recursively find a local maximum in either $[a, m]$ or $[m+1, b]$, depending on which of $H[m], H[m+1]$ is bigger. But we don't need the queried points $m, m+1$ to be adjacent: suppose instead that we query two points x, y, $x < y$. Again, if $H[x] \geqslant H[y]$, we can recurse in $[a, y-1]$; otherwise we recurse in $[x+1, b]$. But when we recurse in $[a, y-1]$, we already have one queried point in the interval, x, so we just need to query one point in the next step. To see why this recursion will indeed produce a local maximum, note that the the point you find in the end will be the best among all the queried points.

   To find the right x, y to start with, we use the golden ratio $\phi = \frac{1+\sqrt{5}}{2}$, or rather its inverse $\frac{\sqrt{5}-1}{2} \approx 0.618$. If we let $x = 0.618a + 0.382b$ (rounded to the nearest integer), $y = 0.382a + 0.618b$ to start with, and recurse in $[a, y-1]$, then $x \approx 0.382a + 0.618(y-1)$, so the points in the middle will always roughly form the same golden ratio.

   This solution uses $29 \approx \log_\phi N$ queries, whereas binary search would use $40 \approx \log_2 N$.

2. For two dimensions, we can reuse some of the ideas of the binary search from the one-dimension case. Let us ask about the values of a line that cuts the rectangle in two. Consider the maximum of the values, and also query the two values to the left and right of this maximum (assuming the cut is vertical). If they are both less or equal to the maximum value, our point is a valid answer. Otherwise, we can recurse on a side where it increases. Intuitively, if we then continued walking to larger and larger values, we could never again cross the cutting line, because the value we walked to was larger than every value on it. There are some subtleties to this, where we have to make sure to recurse on the side that have the previously found maximal query value if that value is larger than the maximum value on the cutting line – otherwise, a local maximum that we found in a subrectangle may not be a local maximum of a full rectangle. Correctly implemented, though, this solves subtasks 3 and 4, assuming you alternate vertical and horizontal splits. It can also be generalized to 1 and 3 dimensions, solving subtasks 1 and 5.

3. One naive idea is to query points at random, and then print the best one, i.e. the one with the highest humidity. Another naive idea is to start from a random point, query adjacent points, and move in the direction of increasing humidity until you find a local maximum.

Neither of these ideas work very well in the worst case, but they can be combined into a solution: First query $Q/2$ points at random. Then choose the best one, and move in the direction of increasing humidity until you arrive at a local maximum.

Why does this work? Suppose in the first stage you find a point among the $Q/12$ best points. Then as you move to better points, you will need at most $Q/12$ steps. For each step, you might need to query 6 points (or 3 on average if you do this cleverly), so you will need $Q/12 \cdot 6 = Q/2$ queries in the second stage, i.e. Q queries in total. So we succeed if we find a point among the $Q/12$ best points.

The probability of not finding one of these points is at most

$$\left(1 - \frac{Q}{12NMK}\right)^{Q/2} \approx \exp\left(-\frac{Q}{12NMK}\frac{Q}{2}\right) = \exp\left(-\frac{Q^2}{24NMK}\right).$$

For group 6, this gives a probability of failure less than 1 in 1800. This solution will also solve group 1, 3, and 5.

Fun facts about this problem:

- The grader for this task was 900 lines long, and implemented 14 different strategies for (on-demand per query) test data generation. This included random test data, space-filling curves, spirals, random line segment paths with slopes leading up to them, and fun 1d functions like randomly rescaled $\sqrt{x}$ and sawtooth functions.
- We had vague plans on extending the task to 4d, but scrapped them. If anyone wants code for random space-filling curves in 4d, just ask.
- Apparently this problem has been fairly well studied by computer scientists, with regards to upper and lower bounds. See for instance:
  - https://arxiv.org/abs/quant-ph/0504085
  - https://epubs.siam.org/doi/pdf/10.1137/S0097539704447237