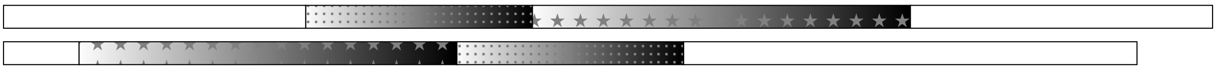
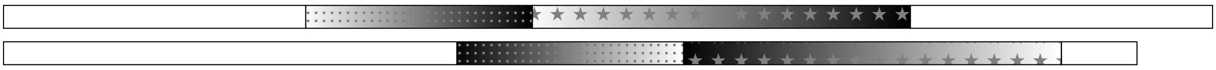
Necklace

Spoiler

Let's represent the strings given to the girls by S and T. A pair of matching necklaces can be found as concaternation of strings A and B such that AB is a substring of S and BA is a substring of T.



You might need to reverse T first. This converts the following case into the previous one.



2-approximation. As each necklace match consist of two substring matches, at least one of them has to be no shorter than half the length of the necklace. Let $L_{SS}(i,j)$ be the length of the common suffix of $S[:i]$ and $T[:j]$. Depending on if $S[i] = T[j]$, $L_{SS}(i+1, j+1)$ is $L_{SS}(i,j)+1$ or 0. To find the longest common substring, we try all possible $d = j - i$ and for each loop over k in increasing order calculating $L_{SS}(k+1, k+1+d)$ from $L_{SS}(k, k+d)$. This takes $\mathcal{O}(N^2)$ time, $\mathcal{O}(1)$ extra memory.

$\mathcal{O}(N^4)$ and $\mathcal{O}(N^3)$. As we have seen, a necklace match can be decomposed into two substring matches by cutting the substrings that give the necklace match at some points. For each possible pair of cut points (i,j) (all pairs of indexes of S and T), we'll find the longest necklace that has these cut points. To find it, we can maximize length of the halves of the necklace separately. Let $L_{SP}(i,j)$ be the length of longest suffix of $S[:i]$, that is a prefix of $T[j:]$. Similarly let $L_{PS}(i,j)$ be the length of longest prefix of $S[i:]$ that is a suffix of $T[:j]$. The longest necklace with cut points (i,j) has length $L_{SP}(i,j) + L_{PS}(i,j)$. To find $L_{SP}(i,j)$ we can check all lengths naively in $\mathcal{O}(N^2)$, giving an $\mathcal{O}(N^4)$ solution overall. Comparing equal length prefixes and suffixes with a rolling polynomial hash gives an $\mathcal{O}(N^3)$ solution overall.

Full DP solution. To get a faster solution, we need to find $L_{SP}(i,j)$ for many pairs of indexes at once. To do this, we will use $L_{SS}(i,j)$. If $L_{SS}(i,j) = l$ then $L_{SP}(i, j-l) \geq l$, $L_{SP}(i, j-l+1) \geq l-1$, etc. Passing the length from $L_{SS}(i,j)$ to $L_{SP}(i, j-l), L_{SP}(i, j-l+1), \ldots, L_{SP}(i, j-1)$ for all (i,j) is enough to calculate L_{SP}. Doing this naively would take $\mathcal{O}(N^3)$ time. We can optimize it by doing $L_{SP}(i, j - L_{SS}(i,j)) = \max(L_{SP}(i, j - L_{SS}(i,j)), L_{SS}(i,j))$ for all (i,j) and then $L_{SP}(i,j) = \max(L_{SP}(i,j), L_{SP}(i, j-1) - 1)$ for all (i,j). This gives an $\mathcal{O}(N^2)$ solution. To improve the memory usage to $\mathcal{O}(N)$ you need to analyze the DP transitions carefully.

Full randomized solution. Choose a pair of indexes randomly. Extend (i,j) to $([l_1, r_1), [l_2, r_2))$ describing the longest substring match that (i,j) is part of. This takes time proportional to the length of the substring match. If the longest common substring has length l, then it takes on average $\frac{N^2}{l}$ attempts to find it. So, this is a randomized $\mathcal{O}(N^2)$ solution to finding the longest common substring.

To find necklaces, we'll generate substring matches this way. For a match of length l, we'll try to extend it with strings of length up to l to get a necklace match. We can check all lengths naively in $\mathcal{O}(l^2)$, giving an $\mathcal{O}(lN^2)$ solution. Using a rolling polynomial hash gives an $\mathcal{O}(N^2)$ solution. The memory usage is $\mathcal{O}(N)$. This solution is on average faster than the DP solution.

Credits

- Task: Jakub Radoszewski (Poland)
- Solutions and tests: Oliver Nisumaa, Andres Unt (Estonia)