Problem Tutorial: "Z-function to prefix-function"

For each i if $z[i] \neq 0$ there is a substring $s[i + z[i] - 1 \dots i]$ matching with the prefix of the same length. It means that $p[i + z[i] - 1]$ is at least $z[i]$.

So for each i if $z[i] > 0$ do $p[i + z[i] - 1] = \max(z[i], p[i + z[i] - 1])$. Also for any i: $p[i] \geq p[i + 1] - 1$. Iterate over i from $n - 2$ to 0 and do $p[i] = max(p[i], p[i + 1] - 1)$.

Problem Tutorial: "Prefix-function to z-function"

It is easy to construct such string s which prefix-function is exactly the given array p. After it use classic z-algorithm to find z-function by s.

Problem Tutorial: "Prefix-function"

Use classic linear algorithm to solve the problem:

```
for i=1..n-1:
    k = b[i - 1]
    while k > 0 && s[k] != s[i]:
        k = b[k - 1]
    if s[k] == s[i]:
        b[i] = k + 1
```

Problem Tutorial: "Z-function"

Use classic linear z-algorithm to solve the problem:

```
for i = 1..n-1:
    if r >= i:
        z[i] = min(z[i - l], r - i + 1)
    while z[i] + i < n && s[z[i]] == s[z[i] + i]:
        z[i]++
    if i + z[i] - 1 > r:
        l = i, r = i + z[i] - 1
```

Problem Tutorial: "Prefix-palindromes"

Construct new string $t = s +$ '#' $+ reverse(s)$. Find prefix-function of t. The prefix-function for the last (the rightmost) index is exactly the length of the expected palindrome.

Problem Tutorial: "Two Strings"

At first let's solve the simplified version of the problem: given the only string a find such shortest s that the infinite string $ss \dots s \dots$ contains a. It is easy to see that the prefix of a of length $|a| - p[|a| - 1]$ is the answer, where p is the prefix-function of a. Let's call a function returning the answer of the simplified problem a $compress(a)$.

If you are given two strings a and b. There are four possible cases:

- a is a substring of b, return $compress(b)$;
- b is a substring of a, return $compress(a)$;
- let's write a and b in such a way that a is on the left, b is on the right and suffix of a is merged maximally with prefix of b (see the picture below);
- as previous item, but the leftmost string is b and rightmost is a (symmetric case to the previous).

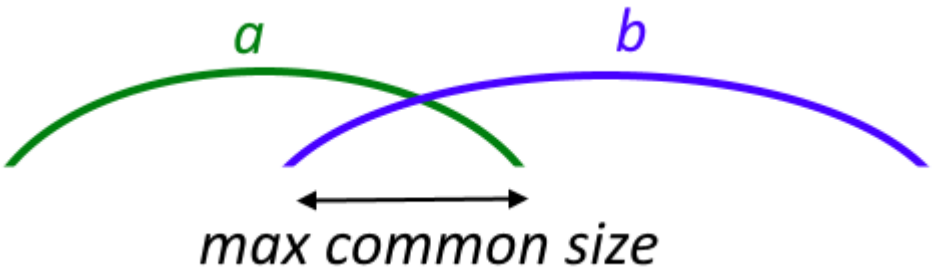

Illustration for the third case.

Let's look on the third case in details. To inject maximally the string a into the string b in the manner as on the picture above, construct new string $t = b + \text{'\#'} + a$. The last (rightmost) value of prefix-function of t is the longest prefix of b matching with the suffix of a. Now construct u, the shortest string of the form like on the picture, and return $compress(u)$.

The fourth case is exactly the same as the third.

In case of no string is a substring of other, consider both cases (the third and the fourth) and print the shortest result.

Problem Tutorial: "Cubes"

Actually the problem is to find all such prefixes of s which have even length and which are palindromes. To find them construct new string $t = s + \text{'\#'} + reverse(s)$ and calculate p — its prefix-function. Then $p[|t| - 1]$ (rightmost element of p) is the length of the longest prefix-palindrome of s. If $p[|t| - 1] > 0$ then $p[p[|t| - 1] - 1]$ is the length of the second longest prefix-palindrome of s and so on. It means that there is a way to iterate over all prefix-palindromes of s in the length decreasing order. Consider only even length prefix-palindromes to print the output.

Problem Tutorial: "Words"

If strings have different length, print '`No`' and abort the solution. Now assume $|s| = |t| = n$.

Let's try all possible values k from 0 to $|s|$ in increasing order. You should check two conditions:

- the prefix of length k of s is the reversed suffix of t;
- $t[0 \dots n - 1 - k] = s[k \dots n - 1]$.

The first conditions can be checked on the fly if values k increases. If $k = 1$ check $s[0] = t[n - 1]$. If $k = 2$ check $s[1] = t[n - 2]$. And so on. If this condition breaks for some k then no reason to consider greater values of k, just abort the solution.

To check the second condition let's precompute z-function of $f = t + s$. The condition $t[0 \dots n - 1 - k] = s[k \dots n - 1]$ holds if and only if $z_f[n + k] = n - k$.

Problem Tutorial: "Minimal String Period"

This string can be solved with naive square-time algorithm, but linear algorithm is as follows. Print first $n - p[n - 1]$ characters of the given string, where n is the string length and p is its prefix-function.

Problem Tutorial: "Minimal String Period (Hard)"

Print first $n-p[n-1]$ characters of the given string, where n is the string length and p is its prefix-function.

Problem Tutorial: "Inaccurate Search"

Assume the length of text t is n and the length of the pattern p is m, $n \geq m$.

For each offset i let's check if it corresponds to an inaccurate matching, i.e. if $t[i \dots i+m-1]$ and p differs in a window of length at most k. To do it, find the length a of the longest common prefix of p and $t[i \dots i+m-1]$. Similarly, find the length b of the longest common suffix of p and $t[i \dots i+m-1]$. The offset i corresponds to an inaccurate matching if and only if $m-a-b \leq k$.

To find the length a of the longest common prefix of p and $t[i \dots i+m-1]$ precompute z-function of the string $x = p+\#+t$. The value $z_x[m+1+i]$ is exactly the longest common prefix of p and $t[i \dots i+m-1]$.

To find the length b of the longest common suffix of p and $t[i \dots i+m-1]$ precompute z-function of the string $y = reverse(p)+\#+reverse(t)$. The value $z_y[n-i+1]$ is exactly the longest common suffix of p and $t[i \dots i+m-1]$.

Problem Tutorial: "Olympiad string"

On each step you should try to increase the length of typed line as much as possible. So greedy algorithm works here. Assume that the already typed part is t and the whole string to type is s. To use the second operation in greedy way you need to find the longest prefix of $s[|t|..|s|-1]$ which occurs in t as a substring. To do it compute z-function of $f = s[|t|..|s|-1]+\#+t$ and choose maximal value $z[j]$ where $j > |s|-|t|$. If such maximal value $z[j']$ is 0, use the first operation, otherwise use the second operation to append $z[j']$ characters at once.

Problem Tutorial: "Retrostring"

Let's compute $rn[0 \dots n-1]$, where $rn[i]$ is the repeatability number of the i-th prefix $s[0 \dots i]$. At first, find $p[0 \dots n-1]$, the prefix-function of s. Then $rn[i] = 0$ if $p[i] = 0$ and $rn[i] = 1+rn[p[i]-1]$ if $p[i] > 0$. So you can compute all values of rn going from 0 to $n-1$.

Now find such maximal j that $rn[j] > rn[i]$ for each i in $0 \dots j-1$.

Problem Tutorial: "Prefix-function of Gray Strings"

Because of regular structure of S, you can calculate prefix-function of a position without classic algorithm usage. Say $p(k,n)$ is the n-th position value of prefix-function of G_k. If $n = 2^k$ (the middle position) the result $p(k,n) = 0$. If $n < 2^k$ then $p(k,n) = p(k-1,n)$. If $n > 2^k$ then $p(k,n) = n-2^k$.

Problem Tutorial: "Electronic Display"

Let's assume that the given strings are a and b, they have the length n and answer is k. It means that the prefix of a of the length $n-k$ equals to the suffix of b of the same length. Out goal is to minimize k, so we can maximize $n-k$ instead. Using z-function of prefix-function find the longest prefix of a equals a suffix of b. For example, you can construct new string $t = a+\#+b$ and calculate its prefix-function. The value in the last position is exactly the required length.

Problem Tutorial: "Fibonacci Strings"

Append separator character to s like $s := s+\#$. After, precalculate p — prefix-function of s. Build finite state machine for the pattern s. You can do it with the following pseudo-code:

```
for c = 'a'..'b':
    A[0][c] = s[0] == c ? 1 : 0
for i = 1..|s|-1:
```

```
for c = 'a'..'b':
    A[i][c] = s[i] == c ? i + 1 : A[p[i - 1]][c]
```

Now calculate two additional arrays $F[i][t]$ and $R[i][t]$, where $F[i][t]$ is the final state if one applies Fibonacci string f_t to the state i, and $R[i][t]$ is the number of new occurrences of the pattern if one applies Fibonacci string f_t to the state i.

The value $R[0][k]$ is the required number of occurrences.

Problem Tutorial: "Epigraph"

Assume the given string is s of length n. Let's calculate answer successively for empty string, for $s[0 \dots 0]$, for $s[0 \dots 1]$, for $s[0 \dots 2]$, ..., for $s[0 \dots n-1]$.

Suppose the answer for $s[0 \dots i-1]$ is a and now let's show how to increment a to be an answer for $s[0 \dots i]$.

Each substring of $s[0 \dots i-1]$ is a substring of $s[0 \dots i]$, so new substrings to count have a form $s[j \dots i]$. Some of the substring of a form $s[j \dots i]$ has been already counted (since they appears in $s[0 \dots i-1]$) and shouldn't be counted, but some of them are not and should be counted.

To find number of substrings of $s[j \dots i]$ which do not appear in $s[0 \dots i-1]$ let's notice that if $s[j' \dots i]$ is appeared in $s[0 \dots i-1]$ then $s[j'' \dots i]$ is appeared for any such j'' that $j < j'' \le i$. And vise versa: if $s[j' \dots i]$ isn't appeared in $s[0 \dots i-1]$ then $s[j'' \dots i]$ isn't appeared for any such j'' that $0 < j'' < j'$.

It means that such j' exists that for all $s[j'' \dots i]$ the substring is already counted ($j' \le j'' \le i$) and for all $s[j'' \dots i]$ the substring isn't counted yet ($0 \le j'' < j'$).

To find such j' you are to find the longest suffix of $s[0 \dots i]$ which appears in $s[0 \dots i-1]$. Reverse the string $s[0 \dots i]$ and find its z-function or prefix-function. The maximal value of z- or prefix-function is exactly the length of the requited suffix. So $j' = i - \max(z[0 \dots i]) + 1$.

So increase a on $(i+1) + i + (i-1) \cdots + (i - j' + 2)$.

Problem Tutorial: "Maximal XOR"

Let's build a trie containing all of the given numbers as binary strings of the same length (add trailing zeroes if needed). Assume, the first element in XOR is a_i, let's find possible maximal value of $x = a_i$ `xor` a_j for the fixed a_i.

The leftmost bit of x is 1 if and only if there is such a_j which starts on bit opposite to the leftmost bit of a_i. You can easily check it by going along edge in the trie started in the root. In the same manner you can find the next bit of x and so on.