Z-function

For given S (of length n) let's define array z. z[0] is not defined, all other items are defined as follows: z[i] is the length of the largest common prefix of string S and string S[i..n-i].

Let's calculate z[1] in O(n). Now let's consider z[k], for such k that answer is calculated for all smaller k. Let R stand for the maximum among all i + z[i] for i < k, and L - is i corresponding to it. Then from definition of Z-function *s[0..R-L-1] = s[L..R-1]* and *s[R-L]* != *s[R]*. Hence, *z[k] >= min(z[k - L], R - k)*.

Proof: By the definition of Z-function strings *s[0..z[k - L] - 1]* and *s[(k - L)..(k - L) + z[k - L] - 1]* are equal (so, all their prefixes of same length as well). Hence *s[0..min(z[k - L], R - k) - 1] = s[(k - L)..(k - L) + min(z[k - L], R - k) - 1] = s[k..k + min(z[k - L], R - k) - 1]*.

In last equality we used the fact that *s[0..R - L - 1] = s[L..R - 1]* and added *L* to the both borders of the second string getting borders of third one by this. But we needed to change *z[k - L]* in *min(z[k - L], R - k)*, in order to not get out of *R - L - 1* in left part of equality and out of *R - 1* in the right one. □

Hence, we can initialize *z[k] = min(z[k - L], R - k)*, and calculate z-function in naive way after this incrementing it by 1 till *s[k + z[k]] = s[z[k]]*. After this we can see that if *k + z[k - L] < R*, then *z[k] = z[k - L]*, i.e., we will calculate *z[k]* in O(1), in the other case every time we will increase *z[k]*, *R* will be increased as well.

To determine the complexity of this algorithm we should remember than on each step we either calculate z[k] in O(1) or increase R. We also never decrease R and R = n in the end so it can be increased no more than n times. Hence algorithm works in O(n).

For curious reader: Consider two strings S and P, you are to find all occurrences of P in S in linear time with the help of z-function. Tip: Consider string P#S.

Note: Similar idea is used in Manacher's algorithm of finding all subpalindromes of given string in O(n) time. You can find it here: http://codeforces.com/blog/entry/12143
It is worth noting that algorithm can be simplified to the case when we interested only in subpalindromes of odd length. We should use following trick: insert meta-character '#' between every pair of letters. Hence code will be simplified a lot, because every palindrome of initial string has exact center now - it is usual character for palindromes of odd length and meta-character '#' for palindromes of even length.

Prefix-function

For given string S let’s call P its border if it is both prefix and suffix of string S. We will also use this word talking about length of P.

For given string S let’s define integer array pi. pi[0] equals 0 by definition (from now and on we will use 0-indexation), other elements are defined as follows: pi[i] is the maximum border of string S[0..i]. Array pi is called prefix-function.

Let’s calculate pi[k] after pi[i] is calculated for every i < k. Consider following properties if prefix-function:

1) if string s[0..k] has border t, then string s[0..k-1] has border t - 1

2) if string s[0..k-1] has border t-1 and s[k] = s[t], then s[0..k] has t. Hence we can check in O(1) whether we can extend the border to the value t.

So we can brute-force all borders of s[0..k-1] in descending order and after that choose the largest among them which can be extended to be a border of s[0..k]. In this way we will obtain the maximum border of s[0..k]. For string s[0..k-1] maximum border equals pi[k - 1], next one equals pi[pi[k - 1] - 1] and so on.

Now we can more strictly formulate the algorithm of prefix-function calculation. Let’s calculate prefix-function one by one for each prefix of given string. To calculate next p[k] let’s brute-force all borders of s[0..k-1] in descending order (p[k - 1], p[p[k - 1] - 1] and so on), until we find one we can extend. After that let’s stop the cycle and extend it getting p[k].

Consider length of current border. We can make one of two following actions with border:

1) replace it with the next in descending order (which decreases length)
2) extending current border (which increases length by 1)

Second type of action was used no more than n times. Hence we can use first action no more than n times (because length is always non-negative)

Hence, algorithm works in O(n).

For curious reader: Consider two strings S and P, you are to find all occurrences of P in S in linear time with the help of prefix-function. Tip: Consider string P#S.

Note: On every step this algorithm can take O(n) time. There are algorithms which take O(1) time on each step. To obtain this you should use so-called pattern matching automaton or automaton of prefix-function. About this and also about generalization of prefix-function on the case of many strings (well-known Aho-Corasick algorithm) you can read in the following article:

http://codeforces.com/blog/entry/14854