MAXimal

Page Authors

Z-function and its calculation

Suppose we are given a string s of length n. The **Z-function** for this string is an array of length n where the i-th element is equal to the greatest number of characters starting from the position i that coincide with the first characters of s.

In other words, $z[i]$ is the length of the longest common prefix between s and the suffix of s starting at i.

Note. In this article, to avoid ambiguity, we assume 0-based indexes; that is: the first character of s has index 0 and the last one has index $n-1$.

The first element of Z-function, $z[0]$, is generally not well defined. In this article we will assume it is zero (although it doesn't change anything in the algorithm implementation).

This article presents an algorithm for calculating the Z-function in $O(n)$ time, as well as various of its applications.

Examples

For example, here are the values of the Z-function computed for different strings:

$s =$ 'aaaaa'

```
z[0] = 0,
z[1] = 4,
z[2] = 3,
z[3] = 2,
z[4] = 1.
```

$s =$ 'aaabaab'

```
z[0] = 0,
z[1] = 2,
z[2] = 1,
z[3] = 0,
z[4] = 2,
z[5] = 1,
z[6] = 0.
```

$s =$ 'abacaba'

```
z[0] = 0,
z[1] = 0,
z[2] = 1,
z[3] = 0,
z[4] = 3,
z[5] = 0,
z[6] = 1.
```

Trivial algorithm

Formal definition can be represented in the following elementary $O(n^2)$ implementation.

```
vector<int> z_function_trivial(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1; i < n; ++i)
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
    return z;
}
```

We just iterate through every position i and update $z[i]$ for each one of them, starting from $z[i] = 0$ and incrementing it as long as we don't find a mismatch (and as long as we don't reach the end of the line).

Of course, this is not an efficient implementation. We will now show the construction of an efficient implementation.

Efficient algorithm to compute the Z-function

To obtain an efficient algorithm we will compute the values of $z[i]$ in turn from $i = 1$ to $n - 1$ but at the same time, when computing a new value, we'll try to make the best use possible of the previously computed values.

For the sake of brevity, let's call **segment matches** those substrings that coincide with a prefix of s. For example, the value of the desired Z-function $z[i]$ is the length of the segment match starting at position i (and that ends at position $i + z[i] - 1$).

To do this, we will keep **the $[l; r]$ indices of the rightmost segment match**. That is, among all detected segments we will keep the one that ends rightmost. In a way, the index r can be seen as the "boundary" to which our string s has been scanned by the algorithm; everything beyond that point is not yet known.

Then, if the current index (for which we have to compute the next value of the Z-function) is i, we have one of two options:

- $i > r$ -- the current position is **outside** of what we have already processed.

  We will then compute $z[i]$ with the **trivial algorithm** (that is, just comparing values one by one). Note that in the end, if $z[i] > 0$, we'll have to update the indices of the rightmost segment, because it's guaranteed that the new $r = i + z[i] - 1$ is better than the previous r.

- $i \le r$ -- the current position is inside the current segment match $[l; r]$.

  Then we can use the already calculated Z-values to "initialize" the value of $z[i]$ to something (it sure is better than "starting from zero"), maybe even some big number.

  For this, we observe that the substrings $s[l \dots r]$ and $s[0 \dots r - l]$ **match**. This means that as an initial approximation for $z[i]$ we can take the value already computed for the corresponding segment $s[0 \dots r - l]$, and that is $z[i - l]$.

  However, the value $z[i - l]$ could be too large: when applied to position i it could exceed the index r. This is not allowed because we know nothing about the characters to the right of r: they may differ from those required.

  Here is **an example** of a similar scenario:

  $$s = \text{‘aaaabaa’}$$

  When we get to the last position ($i = 6$), the current match segment will be $[5; 6]$. Position 6 will then match position $6 - 5 = 1$, for which the value of the Z-function is $z[1] = 3$. Obviously, we cannot initialize $z[6]$ to 3, it would be completely incorrect. The maximum value we could initialize it to is 1 -- because it's the largest value that doesn't bring us beyond the index r of the match segment $[l; r]$.

  Thus, as an **initial approximation** for $z[i]$ we can safely take:

  $$z_0[i] = \min(r - i + 1,\ z[i - l])$$

  After having $z[i]$ initialized to $z_0[i]$, we try to increment $z[i]$ by running the **trivial algorithm** -- because in general, after the border r, we cannot know if the segment will continue to match or not.

Thus, the whole algorithm is split in two cases, which differ only in **the initial value** of $z[i]$: in the first case it's assumed to be zero, in the second case it is determined by the previously computed values (using the above formula). After that, both branches of this algorithm can be reduced to the implementation of **the trivial algorithm**, which starts immediately after we specify the initial value.

The algorithm turns out to be very simple. Despite the fact that on each iteration the trivial algorithm is run, we have made significant progress, having an algorithm that runs in linear time. Later on we will prove that the running time is linear.

Implementation

Implementation turns out to be rather laconic:

```
vector<int> z_function(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min (r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1;
    }
    return z;
}
```

Comments on this implementation

The whole solution is given as a function which returns an array of length n -- the Z-function of s.

Array z is initially filled with zeros. The current rightmost match segment is assumed to be $[0; 0]$ (that is, a deliberately small segment which doesn't contain any i).

Inside the loop for $i = 1 \dots n - 1$ we first determine the initial value $z[i]$ -- it will either remain zero or be computed using the above formula.

Thereafter, the trivial algorithm attempts to increase the value of $z[i]$ as much as possible.

In the end, if it's required (that is, if $i + z[i] - 1 > r$), we update the rightmost match segment $[l; r]$.

Asymptotic behavior of the algorithm

We will prove that the above algorithm has a running time that is linear in the length of the string -- thus, it's $O(n)$.

The proof is very simple.

We are interested in the nested `while` loop, since everything else is just a bunch of constant operations which sums up to $O(n)$.

We will show that **each iteration** of the `while` loop will increase the right border r of the match segment.

To do that, we will consider both branches of the algorithm:

- $i > r$

  In this case, either the `while` loop won't make any iteration (if $s[0] \neq s[i]$), or it will take a few iterations, starting at position i, each time moving one character to the right. After that, the right border r will necessarily be updated.

  So we have found that, when $i > r$, each iteration of the `while` loop increases the value of the new r index.

- $i \leq r$

  In this case, we initialize $z[i]$ to a certain value z_0 given by the above formula. Let's compare this initial value z_0 to the value $r - i + 1$. We will have three cases:

  - $z_0 < r - i + 1$

    We prove that in this case no iteration of the `while` loop will take place.

It's easy to prove, for example, by contradiction: if the `while` loop made at least one iteration, it would mean that initial approximation $z[i] = z_0$ was inaccurate (less than the match's actual length). But since $s[l \dots r]$ and $s[0 \dots r-l]$ are the same, this would imply that $z[i-l]$ holds the wrong value (less than it should be).

Thus, since $z[i-l]$ is correct and it is less than $r-i+1$, it follows that this value coincides with the required value $z[i]$.

- $z_0 = r-i+1$

  In this case, the `while` loop can make a few iterations, but each of them will lead to an increase in the value of the r index because we will start comparing from $s[r+1]$, which will climb beyond the $[l;r]$ interval.

- $z_0 > r-i+1$

  This option is impossible, by definition of z_0.

So, we have proved that each iteration of the inner loop make the r pointer advance to the right. Since r can't be more than $n-1$, this means that the inner loop won't make more than $n-1$ iterations.

As the rest of the algorithm obviously works in $O(n)$, we have proved that the whole algorithm for computing Z-functions runs in linear time.

Applications

We will now consider some uses of Z-functions for specific tasks.

These applications will be largely similar to applications of prefix function.

Search the substring

To avoid confusion, we call t the **string of text**, and p the **pattern**. The problem is: find all occurrences of the pattern p inside the text t.

To solve this problem, we create a new string $s = p + \diamond + t$, that is, we apply string concatenation to p and t but we also put a separator character $\diamond$ in the middle (we'll choose $\diamond$ so that it will certainly not be present anywhere in the strings p or t).

Compute the Z-function for s. Then, for any i in the interval $[0;\ \text{length}(t)-1]$, we will consider the corresponding value $k = z[i+\text{length}(p)+1]$. If k is equal to $\text{length}(p)$ then we know there is one occurrence of p in the i-th position of t, otherwise there is no occurrence of p in the i-th position of t.

The running time (and memory consumption) is $O(\text{length}(t)+\text{length}(p))$.

Number of distinct substrings in a string

Given a string s of length n. Count the number of distinct substrings of s.

We'll solve this problem iteratively. That is: knowing the current number of different substrings, recalculate this amount after adding to the end of s one character.

So, let k be the current number of distinct substrings of s. We append a new character c to s. Obviously, there can be some new substrings ending in this new character c (namely, all those strings that end with this symbol and that we haven't encountered yet).

Take a string $t = s + c$ and invert it (write its characters in reverse order). Our task is now to count how many prefixes of t are not found anywhere else in t. Let's compute the Z-function of t and find its maximum value $z_{\max}$. Obviously, t's prefix of length $z_{\max}$ occurs also somewhere in the middle of t. Clearly, shorter prefixes also occur.

So, we have found that the number of new substrings that appear when symbol c is appended to s is equal to $\text{length}(t) - z_{\max}$.

Consequently, the running time of this solution is $O(n^2)$ for a string of length n.

It's worth noting that in exactly the same way we can recalculate, still in $O(n)$ time, the number of distinct substrings when appending a character in the beginning of the string, as well as when removing it (from the end or the beginning).

String compression

Given a string s of length n. Find its shortest "compressed" representation, that is: find a string t of shortest length such that s can be represented as a concatenation of one or more copies of t.

A solution is: compute the Z-function of s, loop through all i such that i divides n. Stop at the first i such that $i + z[i] = n$. Then, the string s can be compressed to the length i.

The proof for this fact does not differ from that of the solution which uses the prefix function.

Practice Problems

- UVA # 455 "Periodic Strings" [Difficulty: Medium]
- UVA # 11022 "String Factoring" [Difficulty: Medium]
- UVa 11475 - Extend to Palindrome
- LA 6439 - Pasti Pas!
- Codechef - Chef and Strings
- Codeforces - Prefixes and Suffixes