Z algorithm (Linear time pattern searching Algorithm)

This algorithm finds all occurrences of a pattern in a text in linear time. Let length of text be n and of pattern be m, then total time taken is O(m + n) with linear space complexity. Now we can see that both time and space complexity is same as KMP algorithm but this algorithm is Simpler to understand.
In this algorithm, we construct a Z array.

What is Z Array?

For a string str[0..n-1], Z array is of same length as string. An element Z[i] of Z array stores length of the longest substring starting from str[i] which is also a prefix of str[0..n-1]. The first entry of Z array is meaning less as complete string is always prefix of itself.

Example:

Index	0	1	2	3	4	5	6	7	8	9	10	11
Text	a	a	b	c	a	a	b	x	a	a	a	z
Z values	X	1	0	0	3	1	0	0	2	2	1	0

More Examples:

```
str  = "aaaaaa"
Z[]  = {x, 5, 4, 3, 2, 1}

str = "aabaacd"
Z[] = {x, 1, 0, 2, 1, 0, 0}

str = "abababab"
Z[] = {x, 0, 6, 0, 4, 0, 2, 0}
```

How is Z array helpful in Searching Pattern in Linear time?

The idea is to concatenate pattern and text, and create a string “P$T” where P is pattern, $ is a special character should not be present in pattern and text, and T is text. Build the Z array for concatenated string. In Z array, if Z value at any point is equal to pattern length, then pattern is present at that point.
Example:

```
Pattern P = "aab",  Text T = "baabaa"


The concatenated string is = "aab$baabaa"


Z array for above concatenated string is {x, 1, 0, 0, 0,
```

3, 1, 0, 2, 1}.
Since length of pattern is 3, the value 3 in Z array
indicates presence of pattern.

How to construct Z array?

A Simple Solution is to run two nested loops, the outer loop goes to every index and the inner loop finds length of the longest prefix that matches the substring starting at the current index. The time complexity of this solution is $O(n^2)$.

We can construct Z array in linear time.

The idea is to maintain an interval [L, R] which is the interval with max R

such that [L,R] is prefix substring (substring which is also prefix).

Steps for maintaining this interval are as follows –

1) If i > R then there is no prefix substring that starts before i and

ends after i, so we reset L and R and compute new [L,R] by comparing

str[0..] to str[i..] and get Z[i] (= R-L+1).

2) If i <= R then let K = i-L, now Z[i] >= min(Z[K], R-i+1) because

str[i..] matches with str[K..] for atleast R-i+1 characters (they are in

[L,R] interval which we know is a prefix substring).

Now two sub cases arise –

a) If Z[K] < R-i+1 then there is no prefix substring starting at

str[i] (otherwise Z[K] would be larger) so Z[i] = Z[K] and

interval [L,R] remains same.

b) If Z[K] >= R-i+1 then it is possible to extend the [L,R] interval

```
        thus we will set L as i and start matching from str[R]
onwards  and

        get new R then we will update interval [L,R] and calculate
Z[i] (=R-L+1).
```

For better understanding of above step by step procedure please check this animation – http://www.utdallas.edu/~besp/demo/John2010/z-algorithm.htm
The algorithm runs in linear time because we never compare character less than R and with matching we increase R by one so there are at most T comparisons. In mismatch case, mismatch happen only once for each i (because of which R stops), that's another at most T comparison making overall linear complexity.
Below is the implementation of Z algorithm for pattern searching.

- C++
- Java
- Python3
- C#
- Javascript

```
// A C++ program that implements Z algorithm for pattern searching
#include<iostream>
using namespace std;

void getZarr(string str, int Z[]);

// prints all occurrences of pattern in text using Z algo
void search(string text, string pattern)
{
    // Create concatenated string "P$T"
    string concat = pattern + "$" + text;
    int l = concat.length();

    // Construct Z array
    int Z[l];
    getZarr(concat, Z);

    // now looping through Z array for matching condition
    for (int i = 0; i < l; ++i)
    {
```

```
        // if Z[i] (matched region) is equal to pattern
        // length we got the pattern
        if (Z[i] == pattern.length())
            cout << "Pattern found at index "
                << i - pattern.length() -1 << endl;
    }
}

// Fills Z array for given string str[]
void getZarr(string str, int Z[])
{
    int n = str.length();
    int L, R, k;

    // [L,R] make a window which matches with prefix of s
    L = R = 0;
    for (int i = 1; i < n; ++i)
    {
        // if i>R nothing matches so we will calculate.
        // Z[i] using naive way.
        if (i > R)
        {
            L = R = i;

            // R-L = 0 in starting, so it will start
            // checking from 0'th index. For example,
            // for "ababab" and i = 1, the value of R
            // remains 0 and Z[i] becomes 0. For string
            // "aaaaaa" and i = 1, Z[i] and R become 5
            while (R<n && str[R-L] == str[R])
                R++;
            Z[i] = R-L;
            R--;
        }
        else
        {
            // k = i-L so k corresponds to number which
            // matches in [L,R] interval.
            k = i-L;

            // if Z[k] is less than remaining interval
            // then Z[i] will be equal to Z[k].
            // For example, str = "ababab", i = 3, R = 5
```

```
            // and L = 2
            if (Z[k] < R-i+1)
                Z[i] = Z[k];

            // For example str = "aaaaaa" and i = 2, R is 5,
            // L is 0
            else
            {
                // else start from R and check manually
                L = i;
                while (R<n && str[R-L] == str[R])
                    R++;
                Z[i] = R-L;
                R--;
            }
        }
    }
}

// Driver program
int main()
{
    string text = "GEEKS FOR GEEKS";
    string pattern = "GEEK";
    search(text, pattern);
    return 0;
}
```


Output:

```
Pattern found at index 0

Pattern found at index 10
```

Time Complexity: O(m+n), where m is length of pattern and n is length of text.

Auxiliary Space: O(m+n)