Z algorithm (Linear time pattern searching Algorithm) - GeeksforGeeks

Source: https://www.geeksforgeeks.org/z-algorithm-linear-time-pattern-searching-algorithm/

Z algorithm (Linear time pattern searching Algorithm) Last Updated : 5 Aug, 2025 In many programming problems involving strings, we often need to search for occurrences of a pattern inside a text . A classic approach like the naive string matching algorithm checks for the pattern at every index of the text, leading to a time complexity of O(n·m), where n is the length of the text and m is the length of the pattern. This is inefficient for large inputs. To address this, several efficient string matching algorithms exist one of them is the Z-Algorithm , which allows us to perform pattern matching in linear time . That means, it can find all matches in O(n + m) time. Table of Content What Is the Z-Array? Calculation of Z Array How Z-array Helps in Pattern Matching Advantages of Z-Algorithm Real-Life Applications Related Problems What Is the Z-Array? The Z-Algorithm revolves around computing something called the Z-array . Let’s say we have a string s of length n. The Z-array Z[0..n-1] stores, for each index i, the length of the longest substring starting at i that is also a prefix of the string s. In simpler terms: Z[i] tells us how many characters from position i onwards match with the beginning of the string. Try it on GfG Practice Let’s take a small example to understand this better before we move on to the construction of the Z-array. Example: s = "aabxaab". C++ Z [0] = 0 // by definition; we don't compare the full string with itself Now, we compute the remaining values from index 1 to 6. Z[1] = 1 → Only the first character 'a' matches with the prefix. Z[2] = 0 → 'b' does not match the first character of the prefix 'a'. Z[3] = 0 → 'x' does not match the first character of the prefix 'a'. Z[4] = 3 → The substring "aab" matches the prefix "aab". Z[5] = 1 → Only 'a' matches with the first character of the prefix. Z[6] = 0 → 'b' does not match the first character of the prefix 'a'. Final Z-array C++ z [] = [0 , 1 , 0 , 0 , 3 , 1 , 0] Calculation of Z Array The naive approach to compute the Z-array checks, for each index i, how many characters from s[i...] match the prefix starting at s[0]. This can lead to O(n²) time in the worst case. However, using the Z-algorithm , we can compute all Z[i] values in O(n) time. While computing the Z-array, we maintain a window [l, r], known as the Z-box , which represents the rightmost substring that matches the prefix of the string. l is the starting index of the current Z-box (where the prefix match begins). r is the ending index (the farthest position that matches the prefix). Specifically, s[l...r] matches s[0...(r - l)]. This window helps us reuse previous computations to optimize Z-array construction. Why is Z-box helpful? When processing index i, there are two possibilities: If i > r (Outside the Z-box) : => Start comparing the prefix and the substring beginning at i. => Count the number of matching characters and store this length in Z[i]. => Update the window [L, R] to represent this new matching segment. If i ≤ r : => Let k be the position corresponding to i within the prefix (k = i - L). => Use the value Z[k] as a reference. -> If Z[k] is strictly less than the remaining length in [L, R], assign Z[i] = Z[k]. -> Otherwise, begin comparing characters beyond the current window to extend the match. => After extending, update the window [L, R] if a longer match was found. C++ #include <iostream> #include <string> #include <vector> #include <algorithm> using namespace std ; vector < int > zFunction (string s) { int n = s . length (); vector < int > z (n); int l = 0 , r = 0 ; for (int i = 1 ; i < n ; i ++) { if (i <= r) { int k = i - l ; // Case 2: reuse the previously computed value z [i] = min (r - i + 1 , z [k]); } // Try to extend the Z-box beyond r while (i + z [i] < n && s [z [i]] == s [i + z [i]]) { z [i] ++ ; } // Update the [l, r] window if extended if (i + z [i] - 1 > r) { l = i ; r = i + z [i] - 1 ; } } return z ; } int main (){ string s = "aabxaab" ; vector < int > z = zFunction (s); for (int i = 0 ; i < z . size (); ++ i){ cout << z [i] << " " ; } } Java import java.util.ArrayList ; import java.util.Arrays ; public class GfG { public static ArrayList < Integer > zFunction (String s) { int n = s . length (); ArrayList < Integer > z = new ArrayList <> (n); for (int i = 0 ; i < n ; i ++) { z . add (0); } int l = 0 , r = 0 ; for (int i = 1 ; i < n ; i ++) { if (i <= r) { int k = i - l ; // Case 2: reuse the previously computed value z . set (i , Math . min (r - i + 1 , z . get (k))); } // Try to extend the Z-box beyond r while (i + z . get (i) < n && s . charAt (z . get (i)) == s . charAt (i + z . get (i))) { z . set (i , z . get (i) + 1); } // Update the [l, r] window if extended if (i + z . get (i) - 1 > r) { l = i ; r = i + z . get (i) - 1 ; } } return z ; } public static void main (String [] args) { String s = "aabxaab" ; ArrayList < Integer > z = zFunction (s); for (int x : z) { System . out . print (x + " "); } } } Python def zFunction (s): n = len (s) z = [0] * n l , r = 0 , 0 for i in range (1 , n): if i <= r : k = i - l # Case 2: reuse the previously computed value z [i] = min (r - i + 1 , z [k]) # Try to extend the Z-box beyond r while i + z [i] < n and s [z [i]] == s [i + z [i]]: z [i] += 1 # Update the [l, r] window if extended if i + z [i] - 1 > r : l = i r = i + z [i] - 1 return z if __name__ == "__main__" : z = zFunction ("aabxaab") print (" " . join (map (str , z))) C# using System ; using System.Collections.Generic ; public class GfG { public static List < int > zFunction (string s) { int n = s . Length ; List < int > z = new List < int > (new int [n]); int l = 0 , r = 0 ; for (int i = 1 ; i < n ; i ++) { if (i <= r) { int k = i - l ; // Case 2: reuse the previously computed value z [i] = Math . Min (r - i + 1 , z [k]); } // Try to extend the Z-box beyond r while (i + z [i] < n && s [z [i]] == s [i + z [i]]) { z [i] ++ ; } // Update the [l, r] window if extended if (i + z [i] - 1 > r) { l = i ; r = i + z [i] - 1 ; } } return z ; } public static void Main () { string s = "aabxaab" ; List < int > result = zFunction (s); Console . WriteLine (string . Join (" " , result)); } } JavaScript function zFunction (s) { let n = s . length ; let z = new Array (n). fill (0); let l = 0 , r = 0 ; for (let i = 1 ; i < n ; i ++) { if (i <= r) { let k = i - l ; // Case 2: reuse the previously computed value z [i] = Math . min (r - i + 1 , z [k]); } // Try to extend the Z-box beyond r while (i + z [i] < n && s . charAt (z [i]) === s . charAt (i + z [i])) { z [i] ++ ; } // Update the [l, r] window if extended if (i + z [i] - 1 > r) { l = i ; r = i + z [i] - 1 ; } } return z ; } // Driver Code const z = zFunction ("aabxaab"); console . log (z . join (" ")); Output 0 1 0 0 3 1 0 Time Complexity: O(n) Auxiliary Space: O(n) Why This Works in Linear Time The key to linear time complexity is that every time we do character comparisons (manual matching), we extend r the right end of the Z-box. Since r only moves forward and never backward, the total number of such comparisons is at most n. How Z-array Helps in Pattern Matching Given two strings text (the text) and pattern (the pattern), consisting of lowercase English alphabets, find all 0-based starting indices where pattern occurs as a substring in text. Example: Input: text = " aabxaabxaa ", pattern = " aab " Output: [0, 4] Explanation : The key idea is to preprocess a new string formed by combining the pattern and the text , separated by a special delimiter (e.g., $) that doesn’t appear in either string. This avoids accidental overlaps. We construct a new string as: Kotlin s = pattern + '$' + text We then compute the Z-array for this combined string. The Z-array at any position i tells us the length of the longest prefix of the pattern that matches the substring of the text starting at that position (adjusted for offset due to the pattern and separator). So, whenever we find a position i such that: Perl Z [i] == length of pattern it means the entire pattern matches the text at a position: Perl match position = i - (pattern length + 1) Illustrations: C++ #include <iostream> #include <vector> using namespace std ; // Z-function to compute Z-array vector < int > zFunction (string & s) { int n = s . length (); vector < int > z (n); int l = 0 , r = 0 ; for (int i = 1 ; i < n ; i ++) { if (i <= r) { int k = i - l ; // Case 2: reuse the previously computed value z [i] = min (r - i + 1 , z [k]); } // Try to extend the Z-box beyond r while (i + z [i] < n && s [z [i]] == s [i + z [i]]) { z [i] ++ ; } // Update the [l, r] window if extended if (i + z [i] - 1 > r) { l = i ; r = i + z [i] - 1 ; } } return z ; } // Function to find all occurrences of pattern in text vector < int > search (string & text , string & pattern) { string s = pattern + '$' + text ; vector < int > z = zFunction (s); vector < int > pos ; int m = pattern . size (); for (int i = m + 1 ; i < z . size (); i ++) { if (z [i] == m){ // pattern match starts here in text pos . push_back (i - m - 1); } } return pos ; } int main () { string text = "aabxaabxaa" ; string pattern = "aab" ; vector < int > matches = search (text , pattern); for (int pos : matches) cout << pos << " " ; return 0 ; } Java import java.util.ArrayList ; import java.util.Arrays ; public class GfG { // Z-function to compute Z-array static ArrayList < Integer > zFunction (String s) { int n = s . length (); ArrayList < Integer > z = new ArrayList <> (); for (int i = 0 ; i < n ; i ++) { z . add (0); } int l = 0 , r = 0 ; for (int i = 1 ; i < n ; i ++) { if (i <= r) { int k = i - l ; // Case 2: reuse the previously computed value z . set (i , Math . min (r - i + 1 , z . get (k))); } // Try to extend the Z-box beyond r while (i + z . get (i) < n && s . charAt (z . get (i)) == s . charAt (i + z . get (i))) { z . set (i , z . get (i) + 1); } // Update the [l, r] window if extended if (i + z . get (i) - 1 > r) { l = i ; r = i + z . get (i) - 1 ; } } return z ; } // Function to find all occurrences of pattern in text static ArrayList < Integer > search (String text , String pattern) { String s = pattern + '$' + text ; ArrayList < Integer > z = zFunction (s); ArrayList < Integer > pos = new ArrayList <> (); int m = pattern . length (); for (int i = m + 1 ; i < z . size (); i ++) { if (z . get (i) == m){ // pattern match starts here in text pos . add (i - m - 1); } } return pos ; } public static void main (String [] args) { String text = "aabxaabxaa" ; String pattern = "aab" ; ArrayList < Integer > matches = search (text , pattern); for (int pos : matches) System . out . print (pos + " "); } } Python def zFunction (s): n = len (s) z = [0] * n l , r = 0 , 0 for i in range (1 , n): if i <= r : k = i - l # Case 2: reuse the previously computed value z [i] = min (r - i + 1 , z [k]) # Try to extend the Z-box beyond r while i + z [i] < n and s [z [i]] == s [i + z [i]]: z [i] += 1 # Update the [l, r] window if extended if i + z [i] - 1 > r : l = i r = i + z [i] - 1 return z def search (text , pattern): s = pattern + '$' + text z = zFunction (s) pos = [] m = len (pattern) for i in range (m + 1 , len (z)): if z [i] == m : # pattern match starts here in text pos . append (i - m - 1) return pos if __name__ == '__main__' : text = 'aabxaabxaa' pattern = 'aab' matches = search (text , pattern) for pos in matches : print (pos , end = ' ') C# using System ; using System.Collections.Generic ; public class GfG { // Z-function to compute Z-array static List < int > zFunction (string s) { int n = s . Length ; List < int > z = new List < int > (new int [n]); int l = 0 , r = 0 ; for (int i = 1 ; i < n ; i ++) { if (i <= r) { int k = i - l ; // Case 2: reuse the previously computed value z [i] = Math . Min (r - i + 1 , z [k]); } // Try to extend the Z-box beyond r while (i + z [i] < n && s [z [i]] == s [i + z [i]]) { z [i] ++ ; } // Update the [l, r] window if extended if (i + z [i] - 1 > r) { l = i ; r = i + z [i] - 1 ; } } return z ; } // Function to find all occurrences of pattern in text static List < int > search (string text , string pattern) { string s = pattern + '$' + text ; List < int > z = zFunction (s); List < int > pos = new List < int > (); int m = pattern . Length ; for (int i = m + 1 ; i < z . Count ; i ++) { if (z [i] == m){ // pattern match starts here in text pos . Add (i - m - 1); } } return pos ; } public static void Main () { string text = "aabxaabxaa" ; string pattern = "aab" ; List < int > matches = search (text , pattern); foreach (int pos in matches) Console . Write (pos + " "); } } JavaScript function zFunction (s) { let n = s . length ; let z = new Array (n). fill (0); let l = 0 , r = 0 ; for (let i = 1 ; i < n ; i ++) { if (i <= r) { let k = i - l ; // Case 2: reuse the previously computed value z [i] = Math . min (r - i + 1 , z [k]); } // Try to extend the Z-box beyond r while (i + z [i] < n && s [z [i]] === s [i + z [i]]) { z [i] ++ ; } // Update the [l, r] window if extended if (i + z [i] - 1 > r) { l = i ; r = i + z [i] - 1 ; } } return z ; } function search (text , pattern) { let s = pattern + '$' + text ; let z = zFunction (s); let pos = []; let m = pattern . length ; for (let i = m + 1 ; i < z . length ; i ++) { if (z [i] === m){ // pattern match starts here in text pos . push (i - m - 1); } } return pos ; } // Driver Code let text = 'aabxaabxaa' ; let pattern = 'aab' ; let matches = search (text , pattern); console . log (matches . join (" ")); Output 0 4 Time Complexity: O(n + m), where n is the length of the text and m is the length of the pattern, since the combined string and Z-array are processed linearly. Auxiliary Space: O(n + m), used to store the combined string and the Z-array for efficient pattern matching. Advantages of Z-Algorithm Linear Time Complexity for pattern matching. Uses prefix comparison, avoiding re-evaluation of matched characters. Easier to code than KMP; works directly with prefix matches. Useful for preprocessing in multiple string problems beyond pattern matching. Real-Life Applications Search Tools in Text Editors (e.g., VsCode, Sublime) Plagiarism Detection Systems (detect repeated blocks) Bioinformatics (finding exact DNA/RNA pattern matches) Intrusion Detection Systems (match known threat signatures) Compilers (identifying repetitive sequences or keywords) Information Retrieval (document or keyword scanning) Related Problems Search Pattern Find All Occurrences of Subarray in Array Find the Longest Prefix which is also a Suffix Minimum Characters to Add at Front for Palindrome Strings Rotations of Each Other Comment Article Tags: Article Tags: Pattern Searching DSA