Chapter 4

String Matching: the Linear Algorithm

In this chapter we will introduce a linear-time algorithm to solve the string matching computation problem. Unlike the usual way to proceed, where algorithms are presented as they were published, here we will put forward a more recent linear-time algorithm. Although, this algorithm was not the first linear algorithm, it is conceptually relatively simple and fundamental and will help the reader understanding forthcoming algorithms later on. This algorithm is called the Z **algorithm** and is attributed to Gusfield [Gus96, Gus97].

Many string matching algorithms try to avoid computing some of the $n - m + 1$ shifts of the naive algorithm. To do so, they spend a small amount of time (as compared to the overall complexity) learning about the complexity of the pattern or the text. That learning step is called **pre-processing**. Algorithms such as the Knut-Morris-Pratt algorithm [KMP77], the Boyer-Moore algorithm [BM77] or the Apostolico-Giancarlo algorithm [AG86] are examples where the pattern undergoes pre-processing before actually being searched in the text. On the other hand, methods such as suffix trees are based on text pre-processing. Some of those algorithms are **alphabet-dependent**, that is, their time complexity depends on the size of Σ.

4.1 Definitions

Since pre-processing is used in more contexts other than string pattern recognition, we will denote by S an arbitrary string instead of using the usual letter P. Let S be a string and $i > 1$ one of its positions. Starting at i, we consider all the substrings $S[i..j]$, where $i \leq j \leq |S|$, so that $S[i..j]$ matches a prefix of S itself. Among all the possible values j may take, select the greatest one, $j_{\max}$ and denote the number $j_{max} - i + 1$ by $Z_i(S)$. If $S[i]$ is different from $S[1]$, then such a j does not exist and $Z_i(S)$ is set to 0. In other words, $Z_i(S)$ is defined as the maximum length such that $S[i..i + Z_i(S) - 1] \sqsubset S[1..Z_i(S)]$. When S is clear from the context, we will simplify the notation and just write $Z(S)$.

As an example, consider string $S = \{$`aabadaabcaaba`$\}$. The table below shows the $Z_i(S)$ values for S.

i	$S[i..i+Z_i(S)-1]$	$S[1..Z_i(S)]$	Does $Z_i(S)$ exist?	Possible values	$Z_i(S)$
2	`a`	`a`	Yes	1	1
3	`b`	`a`	No	0	0
4	`a`	`a`	Yes	1	1
5	`d`	`a`	No	0	0
6	`aab`	`aab`	Yes	$\{1, 2, 3\}$	3
7	`a`	`a`	Yes	1	1
8	`b`	`a`	No	0	0
9	`c`	`a`	No	0	0
10	`aaba`	`aaba`	Yes	$\{1, 2, 3, 4\}$	4
11	`a`	`a`	Yes	1	1
12	`b`	`a`	No	0	0
13	`a`	`a`	Yes	1	1

Table 4.1: Computation of $Z_i(S)$ values.

For positive values of Z_i, define the Z-**box** at position i as the interval starting at i and ending at $i + Z_i - 1$. Geometrically, this is equivalent to drawing a box whose base is the interval $[i, i + Z_i - 1]$. Figure 4.1 displays the Z-boxes corresponding to string $S = \{$`aabadaabcaabc`$\}$. When $Z_i = 1$, the box is reduced to a line segment.

For a fixed $i > 1$, consider all the Z-boxes starting at j, where $2 \leq j \leq i$. Among all those Z-boxes, select the rightmost endpoint and call it r_i. For the box realizing value r_i, call l_i its starting point. Figure 4.2 shows all the Z-boxes associated to string S. Arrows point to the Z-boxes of each element. The values of r_i and l_i are displayed in the table below the figure.

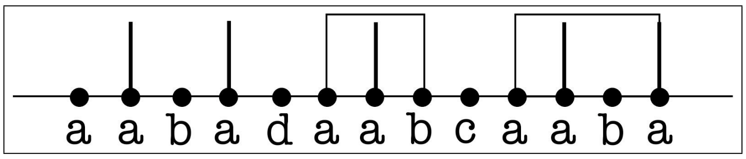
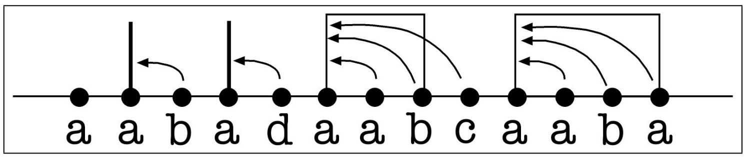


Figure 4.1: Z-boxes.



i	1	2	3	4	5	6	7	8	9	10	11	12	13
l_i	–	2	2	4	4	6	6	6	6	10	10	10	10
r_i	–	2	2	4	4	8	8	8	8	13	13	13	13

Figure 4.2: Values r_i and l_i for string S.

Exercises

$\boxed{1}$ Given string $S = \{\texttt{aaabcaabdaabcaaab}\}$, compute its values $Z_i(S)$, draw its Z-boxes and obtain values r_i and l_i, as done in the text.

4.2 The Pre-Processing Step

We carry on by describing the Z algorithm in detail. When computing values Z_k, the algorithm only needs values r_{k-1} and l_{k-1}. Because of this, we will remove subscripts from variables r_k and l_k, thus simplifying the notation. The algorithm needs to know no more than two values Z_i, but those may be arbitrary, and subscripts cannot be removed.

The algorithm starts by explicitly computing Z_2. In order to do so, substrings $S[1..|S|]$ and $S[2..|S|]$ are compared until a mismatch occurs. Value Z_2 verifies that $S[1..Z_2] = S[2..Z_2]$ and $S[1..Z_2+1] \neq S[2..Z_2+1]$. Once Z_2 is obtained the algorithm proceeds iteratively. Let us compute the k-th value Z_k. Assuming that previous values are already computed, the algorithm performs the following case analysis.

Case 1: $k > r$. In this case the algorithm cannot use previous values of Z_i to construct the Z-box for k. It simply compares substrings starting at k with prefixes of S until a mismatch is found. Thus, Z_k is set to the length of the matching substring. Furthermore, $l = k$ and $r = k + Z_k - 1$. See Figure 4.3.

Case 2: $k \leq r$. This time some previous information may be employed to compute Z_k. Since $k \leq r$, k must be contained in a Z-box. By the definition of l and r, $S[k]$

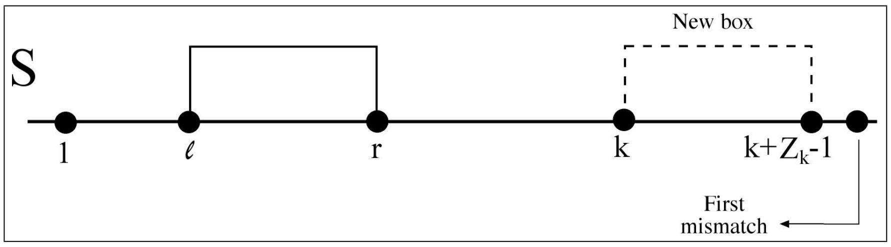


Figure 4.3: Case 1 of the Z algorithm.

belongs to substring $S[l..r]$. Denote by A that substring. Substring A matches a prefix of S. Therefore, character $S[k]$ also appears in substring $S[1..r-l+1]$ at position $k^{'} = k-l+1$. Note that substring $S[k..r]$ is contained in substring $S[1..r-l+1]$ (also note that $Z_l = r-l+1$). We denote substring $S[k..r]$ by B. The copy of substring B in the prefix $S[1..Z_l] = A$ is substring $S[k^{'}..Z_l]$. Figure 4.4 illustrates all these definitions.

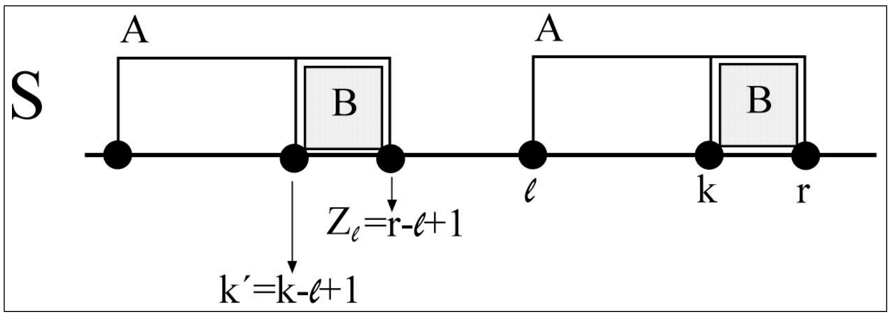


Figure 4.4: Case 2 of the Z algorithm.

When $k^{'}$ was computed, a Z-box of length $Z_{k'}$ was obtained; we will call it C; see Figure 4.5. Such Z-box is also a prefix of S. Therefore, if we produce a substring from character k matching a prefix of S, it will have at least length the minimum of $Z_{k'}$ and $|B|$. Recall that $|B| = r-k+1$.

According to the value of $\min(Z_{k'}, |B|)$ two more cases arise:

Case 2a: $Z_{k'}$ is less than $|B|$. In this case the Z-box at k is the same as the one at $k^{'}$. Hence, Z_k is set to $Z_k^{'}$. Values r and l remain unchanged. See Figure 4.5

Case 2b: $Z_{k'}$ is greater or equal than $|B|$. In this case it substring $S[k..r]$ is a prefix of S and value Z_k is at least $|B| = k-r+1$. However, substring B could be extended to the right while remaining a prefix of S. That would make Z_k be greater than $Z_{k'}$. The algorithm, starting at position $r+1$, searches for the first mismatch. Let q be the position of that mismatch. Then, Z_k is set to $q-1-(k-1) = q-k$, r is set to $q-1$ and l is set to k. See Figure 4.6

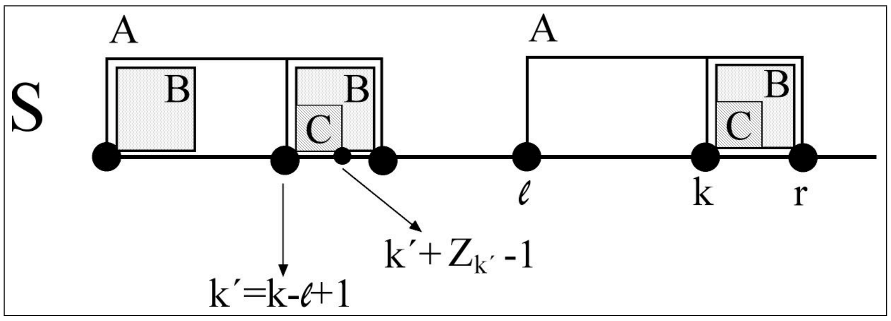


Figure 4.5: Case 2a of the Z algorithm.

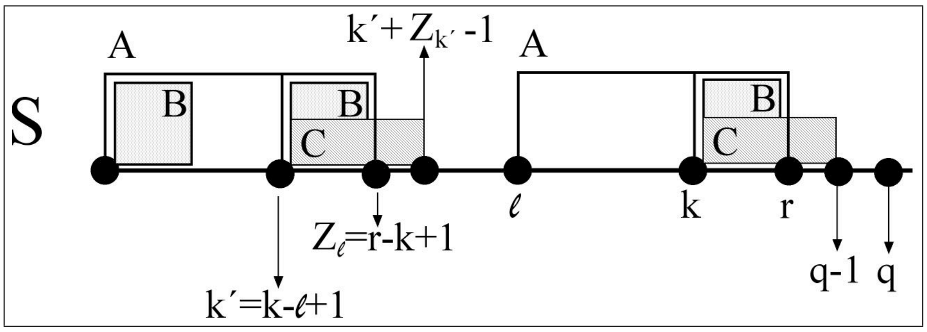


Figure 4.6: Case 2b of the Z algorithm.

Example. Let us trace the algorithm with string $S = \{$`aabcdaabcxyaabcdaabcdx`$\}$. The table below shows values taken by variables Z_i, l_i and r_i as well as which cases the algorithm runs through. Substrings A, B and C are also displayed for Case 2a and Case 2b.

Character	a	a	b	c	d	a	a	b	c	x	y
i	1	2	3	4	5	6	7	8	9	10	11
Z_i	–	1	0	0	0	4	1	0	0	0	0
l_i	–	2	2	2	2	6	6	6	6	6	6
r_i	–	2	2	2	2	9	9	9	9	9	9
Case	–	1	1	1	1	1	2a	2a	2a	1	1
Boxes A, B, C	–	–	–	–	–	–	$A = \{\texttt{aabc}\}$ $B = \{\texttt{abc}\}$ $C = \{\texttt{a}\}$	Same A, $B, C = \{\varepsilon\}$	Same A, B, C	–	–

Character	a	a	b	c	d
i	12	13	14	15	16
Z_i	9	1	0	0	0
l_i	12	12	12	12	
r_i	20	20	20	20	
Case	1	2a	2a	2a	2a
Boxes A, B, C	–	$A = \{\texttt{aabcdaabc}\}$ $B = \{\texttt{abcdaabc}\}$ $C = \{\texttt{a}\}$	Same A, $B, C = \{\varepsilon\}$	Same A, B, C	Same A, B, C

Character	a	a	b	c	d	x
i	17	18	19	20	21	22
Z_i	5	1	0	0	0	0
l_i	17	17	17	17	17	17
r_i	21	21	21	21	21	21
Case	2b	2a	2a	2a	2a	1
Boxes A, B, C	$A = \{\texttt{aabcdaabc}\}$ $B = \{\texttt{aabc}\}$ $C = \{\texttt{a}\}$, $q = 22$	$A = \{\texttt{aabcd}\}$, $B, C = \{\texttt{a}\}$	Same A, $B, C = \{\varepsilon\}$	Same A, B, C	–	–

Table 4.2: Tracing the Z algorithm.

Pseudocode for this algorithm is left to the reader as an exercise. Translating the main ideas to pseudocode should be straightforward. In the next two theorems we will prove correctness and complexity analysis of the Z algorithm.

Theorem 4.2.1 *The pre-processing algorithm correctly computes all the Z_i, r_i and l_i values.*

Proof: In Case 1 value Z_k is always correctly computed as it is obtained by explicit comparisons. None of the Z-boxes starting at a position j, $2 \leq j < k$, ends at or after position k. Therefore, the new value $r = k + Z_k - 1$ is the correct one for r. Setting $l = k$ is also correct.

As for Case 2a, we know that there is a substring $S[k..k + Z_{k'} - 1]$ matching a prefix of S. Such substring has length $Z_{k'} < |B|$. If that substring could be extended to the right, because of the presence of more matching characters, then that would contradict the property of $Z_{k'}$ being maximal. Indeed, if character $Z_{k'}+1$ matches character $k+Z_{k'}$, it follows that $Z_{k'}$ is not the maximal value such that $S[k'..Z_{k'}] \sqsubset S[1..Z_{k'}]$.

It remains to prove Case 2b. B is a prefix of S and the algorithm just extends B to the right by making explicit comparisons. Hence, Z_k is correctly computed. Also, the new r, which is set to $k+Z_k-1$ $(=q-1)$, is greater than any of the other Z-boxes starting below position k. Again, r is correctly updated. In both Cases 2a and 2b l is also correctly updated. ▮

Theorem 4.2.2 *The pre-processing algorithm runs in linear time on the size of S.*

Proof: The Z algorithm performs iterations and character-to-character comparisons. No more than $|S|$ comparisons are made (that corresponds to variable k). The number of comparisons is split between matches and mismatches. The number of mismatches is bounded by $|S|$, the worst case being when all comparisons are mismatches. Then number of matches is kept in variable r_k. Note that r_k is non-decreasing in all the three cases considered by the Z algorithm. If Case 1, r_k is always made bigger than r_{k-1}. If Case 2a, r_k remains unchanged. If Case 2b, r_k is changed to $q-1$, which by construction is greater than r_{k-1}. Therefore, since $r_{|S|} \leq |S|$, the number of matches is bounded by $|S|$.

Thus, we conclude the whole algorithm runs in $O(|S|)$ time. ▮

Exercises

$\boxed{1}$ Write a pseudocode for the Z algorithm.

$\boxed{2}$ Consider the Z algorithm and box A appearing in Case 2a and 2b. Can that box and its prefix overlap? Justify your answer.

$\boxed{3}$ Consider the Z algorithm. If $Z_2 = a > 0$, is it true that all the values $Z_3, \ldots, Z_{a+2}$ can be obtained without further comparisons? Justify your answer.

$\boxed{4}$ Analyse the complexity of the Z algorithm including the constant hidden behind the asymptotic notation. It should be similar to the analysis of *INSERTION-SORT* presented in Chapter 2. Use the pseudocode you were asked to write in the previous section.

4.3 The Linear-Time Algorithm

The Z algorithm can be turned into a string matching algorithm that runs in linear time $O(n+m)$ and linear space $O(n+m)$. The idea is witty and simple. Gusfield, who came up with the idea, claims that "it is the simplest algorithm we know of" ([Gus96], page 10).

Let \$ be a character found neither in P nor in T. Build the string $S = P\$T$. String S has length $n+m+1$, where $m \leq n$. Run the Z algorithm on S. The resulting values Z_i hold the property that $Z_i \leq m$. This is due to the presence of character \$ in S. More important, any value $Z_i = m$ identifies an occurrence of P in T at position $i - (m+1)$. By definition of Z_i,

$$S[i..i + Z_i - 1] = S[i..i + m - 1] = T[i - m - 1..i - 1] = S[1..m] = P[1..m].$$

The converse is also true. When P is found in T at position i, the Z algorithm will output $Z_{m+1+i} = m$. We finish by stating the following theorem.

Theorem 4.3.1 *The string matching computation problem can be solved in linear time and space by using the Z algorithm.*

We will refer to the Z algorithm both when it is used to compute the longest substring that is a prefix of S and when it is used to solve a string matching problem. The context will make clear the meaning in each case.

Example. Let $P = \{$`aab`$\}$ and $T = \{$`abcaaabxy`$\}$. Their respective lengths are $m = 3$ and $n = 9$. Form S by setting $S = P\$T = \{$`aab$abcaabxy`$\}$. Table 4.3 shows the Z_i values for S.

i	1	2	3	4	5	6	7	8	9	10	11	12
$Z_i(S)$	–	1	0	0	1	0	0	3	1	0	0	0

Table 4.3: Solving the SMC problem by using the Z algorithm.

So, at position $8 - (3 + 1) = 4$ we find pattern P in T.

Exercises

$\boxed{1}$ Let $P = \{$`abca`$\}$ and $T = \{$`abcbabcaay`$\}$. Trace the Z algorithm to find all the occurrences of P in T.

$\boxed{2}$ Can the Z algorithm be randomized so that its average-case is sublinear? How is the behaviour of the Z algorithm like when characters of P and T are randomly drawn from the alphabet?

$\boxed{3}$ **Match-Count**. Consider two n-length strings S_1, S_2 and a parameter k, where $1 \leq k \leq n$. There are $n - k + 1$ substrings of length k in each string. Take a substring from each string, align them and count the number of matches. The total number of matches is called the **match-count** of strings S_1 and S_2. A naive algorithm to compute the match-count would take $O(kn^2)$ time. Improve that complexity and produce an $O(n^2)$-time algorithm.

$\boxed{4}$ Can the Z algorithm be easily generalized to solve the 2D string matching problem? Justify your answer.

4.4 More String Matching Problems

An **on-line algorithm** is one that does not have the whole input available from the beginning, but receives it piece by piece. In particular, the algorithm outputs the solution at step i before reading the $(i+1)$-th input symbol. For instance, selection sort requires that the entire list be given before sorting, while insertion sort does not. Selection sort is an **off-line algorithm**. A **real-time algorithm** is an on-line algorithm with the extra condition that a constant amount of work is done between consecutive readings of the input. Intrinsically, off-line and on-line algorithms are quite different. An on-line algorithm, since it cannot examine the whole input, may make decisions that later will turn out not to be optimal. Although on-line and real-time algorithms does not appear often in practice, they still have theoretical interest. See [AHU83, Zvi76] for more information.

The real-time string matching problem is defined by assuming that the text is not entirely available and the pattern is. The text is given to the algorithm character by character in its order. The algorithm has to solve the problem with the information received so far by doing a constant amount of work. Formally, the problem is stated as follows.

The real-time string matching problem (RTSM problem): Find all occurrences of pattern P in text T, where T is input one character at a time.

Notice that trying to solve this problem by just checking whether the last m characters received actually form pattern P is equivalent to the off-line naive algorithm. Its time complexity would be $O(nm)$.

Exercises

$\boxed{1}$ Give an $O(n+m)$ algorithm to solve the real-time string matching problem. Prove the correctness of your algorithm and analyse its time complexity.

4.5 Chapter Notes

Since the Z algorithm solves the SMC problem in linear time, it is simple and efficient, it would seem that it is not worth examining other more complex algorithms. That is not the case. In the next few chapters we will examine several algorithms that are relevant because historical reasons, good performance in practice, good average-case complexity and ease of generalization to other string matching problems.