Z Algorithm - Linear Time Exact String Matching

Thursday, September 14, 2017 2:45 PM

Looking for a pattern \( P \) in text \( T \):
- \( P \rightarrow \) length \( m \)
- \( T \rightarrow \) length \( n \)

\[ \text{Z ALGORITHM: } O(m+n) \]

- Compute \( Z \)-array
  from string \( S \) of length \( n \)

\( Z[i] \) holds the length of the longest substring of string starting from \( S[i] \) which is also a prefix of \( S \)

\[ \begin{array}{ccccc}
  0 & 1 & 2 & 3 & 4 & 5 \\
  S & a & a & b & c & a & a \\
  ~ & 1 & 0 & 0 & 2 & 1 \\
\end{array} \]

Substrings of \( aabca \)
- \( \{a, ab, ..., aabca\} \)
- Only \( a \) is a prefix of \( aabca \)

- Using \( Z \)-array for pattern finding

1. Concatenate \( S = P \$ T \). Thus, any prefix of \( S \) with length \( m \) must be exactly \( P \).

   The special character \( \$ \) ensures that you won’t find any match longer than \( m \) (because \( \$ \) never occurs in \( P \) or \( T \))

\[ \begin{array}{l}
  \text{eg.}\quad P = a a b \\
  \text{Try:}\quad P = a a b \\
  T = a a b a c a a b \\
  \text{solution on bottom of page} \\
\end{array} \]

\[ S = a a b \$ a a b a c a a b \\
Z = ~ 1 0 0 \ 3 \ 2 \ 0 \ 1 0 3 \ 1 0 \]

Given the \( Z \)-array, we can find occurrences in \( O(n) \) time - just search

\[ Z [m : n+m-1] \text{ for the value } 'm'. \]

But! How do we construct \( Z \) in linear time, also?

\[ \begin{array}{cccccccccccccccc}
  1 & 2 & 0 & 1 & 3 & 0 & 0 & 1 & \sim = Z \\
  a & a & b & a & b & \$ & a & b & a & b & a & b & S \\
\end{array} \]
Finding Z Array in Linear Time

Thursday, September 14, 2017 3:19 PM

Naively: Nested loops. Check every position \( \times \) operation) then expand (\( \approx n \) operations) to find longest substring which is also a prefix

\[ = \Theta(n^2) \]

Instead:

Maintain an interval \([l, r]\) which is the longest interval such that \( S[l..r] \) is also a prefix, aka it is a prefix substring

1. If \( i > R \), no prefix substring starts before \( i \) or ends after \( i \).
   Make a new interval \([l, r]\)

2. If \( i < R \), we already have some of the data for the length of the prefix substring starting at \( i \).

\[ Z \]

\[ S \]

\( i \) is in a previously discovered substring which starts at \( i-L \)

\( \text{Either } Z[i-L] < R-i+1 \implies Z[i] = Z[i-L] \)

\( \text{or } Z[i-L] \geq R-i+1 \implies \text{There’s a longer prefix substring starting at } i \)

Recompute \([l, r]\) & determine \( Z[i] = R-L+1 \)

Check out www.utdallas.edu/~besp/demo/John2010/z-algorithm.htm

Why is this linear? Intuition:

\( R \) always moves forward \( \rightarrow O(n) \) increments comparing one character per increment

\( i \) always moves forward, doing constant time work each step when looking at previous \( Z \) values. If it has to modify \( R \), we can consider that the work of the \( R \) variable – that work does not have to be repeated for each \( i \)

\[ L + O(n) + O(n) = O(n) \]