

## Prelim Notes

### Definitions

**Alphabets:** Finite set of symbols e.g., {A,C,T,G}, {A-Z, 0-9, “,”, “.”, ““,” “-”}

**Words, Strings:** n-tuples of symbols from an alphabet set; e.g., AAC, “This is a string”

$u = \text{“ACCT”}$  and  $t = \text{“CCT”}$ , then we write  $ut = \text{“ACCTCCT”}$

$u = \text{“A”}$ , then we write  $u_n = \text{“AAAA...A”}$

**Substring:** A contiguous subset of symbols

**Subsequence:** A co-linear but not necessarily contiguous subset of symbols

Given some alphabet set (e.g., A, C, G, T), string S from the alphabet, we wish to identify all occurrences of a string P (Pattern) in S.

### The “Z” algorithm (Gusfield, 1997)

Define: Given string S,  $Z_i(S)$  is the length of the longest substring of S starting at position i and matching a prefix of S

Example: S = aabcaabxaaz,  $Z_5(S) = 3$ ,  $Z_6(S) = 1$ ,  $Z_7(S) = 0$

Define: For any i where  $Z_i(S) > 0$  (for  $i > 1$ ) a **Z-box** at i is the segment  $(i, i+Z_i(S)-1)$

Z-box: indicates locations of the substrings that match the prefix of the string

AAACGTAACTTAACTTAA

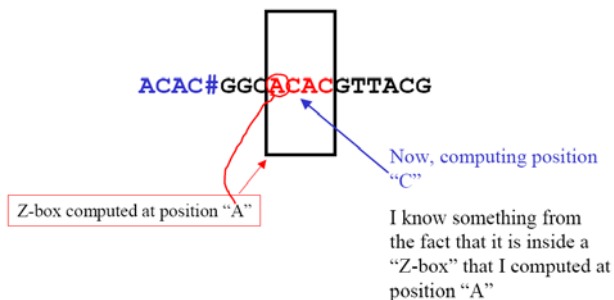
Z-boxes identifies all the substrings that partially match the beginning of the string

Algorithm: Want to find all P in S

1. Assume length of P is m and length of S is n
2. Construct a new string  $T = P\%S$ , where P is the pattern, % is some letter not in P or S, and S is the search string
3. Compute all the Z-boxes
4. Then P occurs where ever  $Z_i = m$

Want to compute  $Z_i$  as efficiently as possible:

**Idea**, before checking a position, see if we have any information from previous computations



### Possible Cases

1. We have no prior information on the current position -> try to match to prefix

↓

ACAC#GGCACACATTACG

2. We have prior information because we know that the current position is inside a previously computed "z-box" -> look back to the z-scores at the prefix

a. prefix z-score is less than remaining length of the search pattern

ACAC#GGCACACATTACG

0020

↓

GGCACACATTACG

This means that there is no way that a "ACAC" matching pattern can start from the 2<sup>nd</sup> position!

b. prefix z-score is identical to remaining length of the search pattern: Try to see if the match can be extended

ACAC#GGCACACATTACG

0020

↓

GGCACACATTACG

This means that there is a possibility that a match to "ACAC" could be started from the 3<sup>rd</sup> position!

### Computing Z-boxes

Variables  
 $Z_i(S)$ : As defined  
 $r$ : right most position of any Z-box starting left of current position  
 $l$ : left position of the Z-box marked by  $r$

1. Compute  $Z_2$  by comparing  $S[2\dots]$  to  $S[1\dots]$ , if  $Z_2 > 0$   $l = 2$ ,  $r = Z_2 + l - 1$ , else  $r = l = 0$
2. Suppose we are at position  $k$ , then assess whether we are inside of a Z-box by comparing  $k$  to  $r$ . If outside, do normal comparison to compute  $Z_k$
3. If inside, that means the current position matches beginning of the string at  $k' = k - l + 1$  position. Examine  $Z_k$ . If it is less than  $r - k + 1$ , then  $Z_k = Z_k$ , and  $r$  and  $l$  do not change. If it is greater, then extend by comparing  $S[r+1\dots]$  to  $S[r-k+2\dots]$ . Set  $Z_k$ ,  $l$  and  $r$  accordingly

### Suffix Tree Definition

A suffix tree for a  $n$ -character string  $S$  is a rooted tree with  $n$  leaves numbered from 1 to  $n$ . Each internal node other than the root has at least two children and each edge is labeled with a substring of  $S$ . **No two edges out of a node can have edgelabels beginning with the same character.** The key feature is that for any leaf  $i$ , **the concatenations of the edge-labels on the path from the root to leaf  $i$  exactly spells out the suffix of  $S$  starting from  $i$ .**

### Building a suffix tree by hand

1. List all suffixes
2. Draw the root and an edge with the longest suffix
3. Consider the next longest suffix, examine to see if it can be attached to an existing branch, otherwise start a new edge from the root.
4. Repeat

### Suffix Tree Application: Find all redundancies

ACGACGCGAC

For every string concatenated along an edge, the number of leaves subtending from that point is the number of occurrences

**Important:** There exists an algorithm that can construct a suffix tree in operations proportional to length of  $S$

### What is a Finite State Automata?

- A machine with finite number of states
- The machine processes a finite alphabet (i.e., takes characters as input)
- Each input character changes the state of the machine

**Definition:** A finite automaton on some alphabet set,  $A$ , is a set of states  $Q = \{q_1, \dots, q_m\}$  and a transition function  $\delta: Q \times A \rightarrow Q$

One of the states is designated as the *initial state* and some subset of the states is designated as the *accepting states*

**Definition:** A *Suffix Function*  $\sigma(s)$  for pattern  $P$  is the length of the longest prefix of  $P$  that is a suffix of  $s$

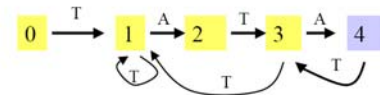
E.g.,  $P = AT$ ,  $\sigma(CCACA) = 1$ ,  $\sigma(CCAT) = 2$ ,  $\sigma(ATT) = 0$

Set transition function as  $\delta(q,c) = \sigma(P[1..q]c)$ .

$P = \text{"TATA"}$

	A	C	G	T
<b>0:</b> $P[1..q] = \phi$	$\sigma(A) = 0$	$\sigma(C) = 0$	$\sigma(G) = 0$	$\sigma(T) = 1$
<b>1:</b> $P[1..1] = T$	$\sigma(TA) = 2$	$\sigma(TC) = 0$	$\sigma(TG) = 0$	$\sigma(TT) = 1$
<b>2:</b> $P[1..2] = TA$	$\sigma(TAA) = 0$	$\sigma(TAC) = 0$	$\sigma(TAG) = 0$	$\sigma(TAT) = 3$
<b>3:</b> $P[1..3] = TAT$	$\sigma(TATA) = 4$	$\sigma(TATC) = 0$	$\sigma(TATG) = 0$	$\sigma(TATT) = 1$
<b>4:</b> $P[1..4] = TATA$	$\sigma(TATAA) = 0$	$\sigma(TATAC) = 0$	$\sigma(TATAG) = 0$	$\sigma(TATAT) = 3$

Algorithm for generating a finite automata to process a pattern  $P$  of length  $m$ : i.e., want to know how to produce the transition table



**Definition:** A *Suffix Function*  $\sigma(s)$  for pattern  $P$  is the length of the longest prefix of  $P$  that is a suffix of  $s$

E.g.,  $P = AT$ ,  $\sigma(CCACA) = 1$ ,  $\sigma(CCAT) = 2$ ,  $\sigma(ATT) = 0$

Set states =  $\{0, \dots, m\}$ , 0 is initial state,  $m$  is accepting state

Set transition function as  $\delta(q,c) = \sigma(P[1..q]c)$ .

Read: Given current state  $q$  and input character  $c$ , the next state is obtained by computing the suffix function on the concatenation of the target pattern  $P$  from 1 to  $q$ th position with the input character