Data Structures and Algorithms

String Algorithms

InverseHackermann

Definitions

Character: one letter of the alphabet. Often appears in single quotes.

String: sequence of characters. Often appears in double quotes. Length denoted as `|S|`

For `s=“onedragon”`, `S[0]=‘o’`, `S[1]=‘n’`, `S[2]=‘e’`, etc.

Substring: A range of characters in a string. `S[3 .. 6]=“drag”`.

Prefix: substring starting at the beginning of the string.

Suffix: substring ending at the end of the string.

Border: both a prefix and a suffix of the string. `“on”` is a border of `“onedragon”`.

Proper: not the entire string. A string is always a border of itself, but not a proper border.

String matching

Goal: look for occurrences of pattern P within S.

Directly comparing at each index can take $|S|*|P|$ comparisons.

Index:	0	1	2	3	4	5	6	7	8	9	10
S:	R	A	W	R	A	R	A	R	W	R	A
P:	R	A	R	W							
No match:				R	A	R	W				
Match:						R	A	R	W		

Knuth–Morris–Pratt (KMP) string matching

Solution: Combine string into `T = P#S` (‘#’ is a unique character).

Store **`LPB[i]` = length of longest proper border for each prefix `T[0 .. i]`**.

To compute `LPB[i]`, extend a border of `T[0 .. i]` by one character.

`LPB[i - 1]` is longest proper border of `T[0 .. i]`; how to get next shorter border?

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`T:`	R	A	R	W	#	R	A	W	R	A	R	A	R	W	R	A
`lpb[i]:`	0	0	1	0	0	1	2	0	1	2	3	2	3	4	1	2
`lpb[10]=3:`																
`lpb[2]=1:`																

Z-function

Again, combine string into `T = P#S` (‘#’ is a unique character). Store **`Z[i]` = length of longest matching prefix starting at index `i`**. (skip `i=0`)

To compute `Z[i]`, compute estimate for match length, then scan forwards.

Track `r` = furthest scanned index; When `r` increases, set `l` to match index, so `S[l .. r] = S[0 .. r - l + 1]`. Use `min(Z[i - l], r - l + 1)` as estimate for `Z[i]`.

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`T:`	R	A	R	W	#	R	A	W	R	A	R	A	R	W	R	A
`z[i]:`		0	1	0	0	2	0	0	3	0	4	0	1	0	2	0
`z[8]=3`																
`Estimates`										0	1					

Suffix Array

Array of sorted suffixes of a string. (Often last character is unique smallest char `$`)

Lexicographic order: compare first characters, break ties by next character.

Helps to think about **text** indices vs **lex** indices: regular string order is text index, lexicographic order is lex index. (I'll try keep `i` as text indices and `j` as lex indices.)

Suffix array: text indices sorted in lex order. Maps lex indices to text indices.

Similar substrings end up in nearby locations in suffix array.

Suffix Array

Text	S
0	R
1	A
2	W
3	R
4	A
5	R
6	A
7	R
8	W
9	R
10	A
11	$

Lex	Text												
0	11	$											
1	10	A	$										
2	4	A	R	A	R	W	R	A	$				
3	6	A	R	W	R	A	$						
4	1	A	W	R	A	R	A	R	W	R	A	$	
5	9	R	A	$									
6	3	R	A	R	A	R	W	R	A	$			
7	5	R	A	R	W	R	A	$					
8	0	R	A	W	R	A	R	A	R	W	R	A	$
9	7	R	W	R	A	$							
10	8	W	R	A	$								
11	2	W	R	A	R	A	R	W	R	A	$		

Burrows–Wheeler Transform (BWT)

Closely related to suffix array: start with **sorted cyclic shifts** of string.

Take last characters of each cyclic shift.

Due to sorted structure, similar substring appearances are grouped into ranges of equal characters, so run-length encoding after BWT can be used for compression.

“LF mapping”: `i`th occurrence of character `c` in first character of shifts is at the same text index as `i`th occurrence of character `c` in last character of shifts.

Use LF property to compute inverse BWT.

Burrows–Wheeler Transform (BWT)

Text	S
0	R
1	A
2	W
3	R
4	A
5	R
6	A
7	R
8	W
9	R
10	A
11	$

Lex	Text	F											L
0	11	$	R	A	W	R	A	R	A	R	W	R	A
1	10	A	$	R	A	W	R	A	R	A	R	W	R
2	4	A	R	A	R	W	R	A	$	R	A	W	R
3	6	A	R	W	R	A	$	R	A	W	R	A	R
4	1	A	W	R	A	R	A	R	W	R	A	$	R
5	9	R	A	$	R	A	W	R	A	R	A	R	W
6	3	R	A	R	A	R	W	R	A	$	R	A	W
7	5	R	A	R	W	R	A	$	R	A	W	R	A
8	0	R	A	W	R	A	R	A	R	W	R	A	$
9	7	R	W	R	A	$	R	A	W	R	A	R	A
10	8	W	R	A	$	R	A	W	R	A	R	A	R
11	2	W	R	A	R	A	R	W	R	A	$	R	A

Longest Common Prefix (LCP) Array and Longest Common Extension (LCE) Queries

Given suffix array `SA`, `LCP[j]` = longest common prefix of `SA[j]` and `SA[j - 1]`. (Skip `j = 0`).

Then `LCE(ia, ib)` = most matching characters starting at text indices `ia` and `ib` = `min(LCP[SA[ia] + 1 .. SA[ib]])`. (May need to swap so SA[ia] < SA[ib].)

Let `SAINV[SA[j]] = j`. (inverse permutation, maps text indices to lex indices.)

Value of `LCP[j] - 1` can estimate next text suffix `LCP[SAINV[1 + SA[j]]]`.

LCP Array and LCE Queries

Text	S
0	R
1	A
2	W
3	R
4	A
5	R
6	A
7	R
8	W
9	R
10	A
11	$

Lex	Text	LCP												
0	11	N/A	$											
1	10	0	A	$										
2	4	1	A	R	A	R	W	R	A	$				
3	6	2	A	R	W	R	A	$						
4	1	1	A	W	R	A	R	A	R	W	R	A	$	
5	9	0	R	A	$									
6	3	2	R	A	R	A	R	W	R	A	$			
7	5	3	R	A	R	W	R	A	$					
8	0	2	R	A	W	R	A	R	A	R	W	R	A	$
9	7	1	R	W	R	A	$							
10	8	0	W	R	A	$								
11	2	3	W	R	A	R	A	R	W	R	A	$		

Range Minimum Queries (RMQ)

Precompute power-of-two-length RMQs.

Break general length RMQ into min of two power-of-two-length RMQ.

Some elements are counted twice, but counting twice in a min is fine.

Index	0	1	2	3	4	5	6	7	8	9
Original RMQ										
Left power of two										
Right power of two										