Competitive Programming

From Problem 2 Solution in O(1)

String Processing Algorithms

Suffix Arrays - LCP

Mostafa Saad Ibrahim
PhD Student @ Simon Fraser University

Longest Common Prefix array

- LCP[i] = longest common prefixes (suffix[i], suffix[i-1])
  - Generate suffixes, compute suffix array
  - For every pair compute the LCP
  - LCP(abcd, abckkk) = 3
  - LCP(abcd, hello_abcd) = 0
- To compute LCP, let's introduce Rank array
  - It represents the reverse of the suffix positions?
  - Recall we have suffix array idx, and its original position
  - Which suffix at position 6 in suffix array? Suffix at 8 (bra)
  - Reverse it
  - Where is suffix generated from pos 8 (bra) ? at position 6

LCP and Rank arrays

Idx	Suffix String	LCP array	Suffix array	Suffix (rev) rank
0		0	11	3
1	a	0	10	7
2	**abra**	1	7	11
3	**abracadabra**	**4**	0	4
4	acadabra	1	3	8
5	adabra	1	5	5
6	bra	0	8	9
7	bracadabra	3	1	2
8	cadabra	0	4	6
9	**dabra**	0	6	10
10	**ra**	**0**	9	1
11	racadabra	2	2	0

LCP(abra, abracadabra) = 4

For i = 6, suffix at 8 exists (bra)
Let's **reverse** that:

Where is suffix at 8? Position 6

```
FOR i
    rank[ suffix[i] ] = i
```

LCP

- Observation
  - Let LCP[i] = k = longest common prefix between ith sorted suffix and it previous one
  - suffix[i] = abcdZZ, suffix[i-1] = abcdXX => k = 4
  - Notice that, if we removed 1 letter from both suffixes
  - suffix[j] = bcdZZ, suffix[j-1] = bcdXX => k = 3
  - So lcp between 2 suffixes, tell us lcp about their internal suffixes
  - E.g. lcp(bcdZZ, bcdXX) = 3, lcp(cdZZ, cdXX) = 2
  - Specifically, if lcp = k, it tell us values of **k+1 consecutive pairs**

LCP

- Observation
  - To make of this, we need to iterate over the order that remove letter by letter from a suffix
  - This is the actual suffix generation order NOT suffix sorted array
  - That is why we needed the rank array, the reverse of it
  - So iterate over generation of suffixes order
  - For current position, get suffix position using rank
  - compute LCP = some K
  - this K will be the answer for K+1 iterations

LCP

0	abracadabra
1	bracadabra
2	racadabra
3	acadabra
4	cadabra
5	adabra
6	dabra
7	abra
8	bra
9	ra
10	a
11	

Idx	Suffix String	LCP array
0		0
1	a	0
2	**abra**	1
3	**abracadabra**	**4**
4	acadabra	1
5	adabra	1
6	bra	0
7	bracadabra	3
8	cadabra	0
9	**dabra**	0
10	**ra**	**0**
11	racadabra	2

- Start at i = 0, abracadabra
- rank[0] = 3
- Its previous one, 2
- Compute LCP(3, 2) = 4
- lcp[rank[0]] = lcp[3] = 4

- Move to i = 1
- so 1 letter from suffix 0 removed
- Let current LCP = 4-1=3

LCP

0	abracadabra
1	bracadabra
2	racadabra
3	acadabra
4	cadabra
5	adabra
6	dabra
7	abra
8	bra
9	ra
10	a
11	

Idx	Suffix String	LCP array
0		0
1	a	0
2	**abra**	1
3	**abracadabra**	**4**
4	acadabra	1
5	adabra	1
6	bra	0
7	bracadabra	3
8	cadabra	0
9	**dabra**	0
10	**ra**	**0**
11	racadabra	2

- Once we move 1 step, the LCP = K, decrease by 1, as 1 letter is removed from the 2 new suffixes
- **a**bracadabra -> bracadabra

- Move to i = 1
- LCP = LCP-1
- LCP = 3
- rank[1] = 8
- lcp[8] = 3

LCP

0	abracadabra
1	bracadabra
2	racadabra
3	acadabra
4	cadabra
5	adabra
6	dabra
7	abra
8	bra
9	ra
10	a
11	

Idx	Suffix String	LCP array
0		0
1	a	0
2	**abra**	1
3	**abracadabra**	**4**
4	acadabra	1
5	adabra	1
6	bra	0
7	bracadabra	3
8	cadabra	0
9	**dabra**	0
10	**ra**	**0**
11	racadabra	2

- Move to i = 2
- LCP = LCP-1
- LCP = 2
- rank[2] = 11
- lcp[11] = 2

LCP

0	abracadabra
1	bracadabra
2	racadabra
3	acadabra
4	cadabra
5	adabra
6	dabra
7	abra
8	bra
9	ra
10	a
11	

Idx	Suffix String	LCP array
0		0
1	a	0
2	**abra**	1
3	**abracadabra**	**4**
4	acadabra	1
5	adabra	1
6	bra	0
7	bracadabra	3
8	cadabra	0
9	**dabra**	0
10	**ra**	**0**
11	racadabra	2

- Move to i = 3
- LCP = LCP-1
- LCP = 1
- rank[3] = 4
- lcp[4] = 1
-

LCP

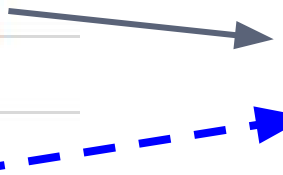
0	abracadabra
1	bracadabra
2	racadabra
3	acadabra
4	cadabra
5	adabra
6	dabra
7	abra
8	bra
9	ra
10	a
11	

Idx	Suffix String	LCP array
0		0
1	a	0
2	**abra**	1
3	**abracadabra**	**4**
4	acadabra	1
5	adabra	1
6	bra	0
7	bracadabra	3
8	cadabra	0
9	**dabra**	0
10	**ra**	**0**
11	racadabra	2

- Move to i = 4
- LCP = LCP-1
- LCP = 0
- rank[4] = 8
- lcp[8] = 0
-

LCP

0	abracadabra
1	bracadabra
2	racadabra
3	acadabra
4	cadabra
5	adabra
6	dabra
7	abra
8	bra
9	ra
10	a
11	



Idx	Suffix String	LCP array
0		0
1	a	0
2	**abra**	1
3	**abracadabra**	**4**
4	acadabra	1
5	adabra	1
6	bra	0
7	bracadabra	3
8	cadabra	0
9	**dabra**	0
10	**ra**	**0**
11	racadabra	2

- Move to i = 5
- LCP = LCP-1
- LCP = -1
- rank[5] = 5

- Now lcp = -1, means we covered the K+1 suffixes
- So recompute LCP from current step
- LCP(adabra, acadabra) = 1
- LCP = 1 (tell us 2 suffixes
- lcp[5] = 1

LCP

0	abracadabra
1	bracadabra
2	racadabra
3	acadabra
4	cadabra
5	adabra
6	dabra
7	abra
8	bra
9	ra
10	a
11	

Idx	Suffix String	LCP array
0		0
1	a	0
2	**abra**	1
3	**abracadabra**	**4**
4	acadabra	1
5	adabra	1
6	bra	0
7	bracadabra	3
8	cadabra	0
9	**dabra**	0
10	**ra**	**0**
11	racadabra	2

- Move to i = 6
- LCP = LCP-1
- LCP = 0
- rank[6] = 9
- lcp[9] = 0

And so on

- If LCP = K
- We know answer of K+1 LCPs
- Build them
- Compute LCP of new missing pair
- and so on
- Overall O(n) processing

LCP Algorithm

```
int Rank[MAXLENGTH + 1];
int lcp[MAXLENGTH + 1];

void buildLcp() {
  //compute the rank of each suffix
  for (int i = 0; i - 1 < 0 || str[i - 1]; i++)
    Rank[suf[i]] = i;

  int c = 0;  // the length of lcp between i and j
  for (int i = 0; i - 1 < 0 || str[i - 1]; i++) {
    if (Rank[i]) {  //if i is not the first suffix in the sorted array
      int j = suf[Rank[i] - 1];  //find the element before i and name it j
      while (str[i + c] == str[j + c])
        c++;  //count the number of shared chars
    }
    lcp[Rank[i]] = c;  //store the result in lcp array
    if (c)
      c--;  //Decrement c by one because length of lcp of i+1 is c-1
  }
```

تم بحمد الله

علمكم الله ما ينفعكم

ونفعكم بما تعلمتم

وزادكم علماً