Suffix Arrays and Longest Common Prefix

Daniel Wisdom

19 January 2018

1 Suffix Arrays

A suffix of a string is a substring from a starting index to the end of the string. We can identify a suffix by the starting index, so A_k is the suffix starting at and including A[k]. A suffix array is an alphabetically sorted array of all suffixes of a string. We could use an $O(n \log n)$ sort with $O(n)$ string comparison, which computes the sorted suffix array in $O(n^2 \log n)$. All faster algorithms for suffix arrays use the fact that we are sorting related strings, not a list of n random strings.

There are algorithms that can compute a suffix array in $O(n)$ time, but they are more complicated than what we need. If you are interested look at SA-IS suffix array construction.

2 Prefix Doubling

In normal string comparison of ”ab” and ”ac”, first we would compare ’a’ and ’a’, and if they are equal ’b’ and ’c’. Now, if we wanted to compare ”abcdef” and ”abcbbb”, we could first compare ”abc” and ”abc”, and because they are equal, compare ”def” and ”bbb”. Using this strategy, if we precomute the comparisons between all substrings of length 3, we can easily compare all substrings of length 6. The prefix doubling algorithm uses this to build up a table of precomputed comparisons for substrings of length 1, 2, 4, 8, etc.

To precompute the comparisons between substrings we will give every substring of length 2^k a rank. The ranks should indicate the ordering of the suffixes and equal substrings must have equal ranks. For substrings of length 1, the rank is just the character value, usually $A[i] -' a'$. For longer substrings, sorting the substrings and then ranking by index is a good start. The problem is that in a sorted list equal elements appear next to each other, not at the same index. To make the ranks the same, we can compare every substring to the one before it in the sorted list, and set the ranks equal if they are equal. This gives us an $O(n \log n)$ way of doubling the length of known substrings. Using $\log n$ doubling steps we can create a sorted list of all suffixes. The overall algorithm takes $O(n \log^2 n)$.

3 Pseudocode

Algorithm 1 Prefix Doubling

```
rank = int[roundUp($\log n$)][$n$]
for $i$ from 0...$n-1$ do
    rank[0][$i$]=A[$i$]-'a'
for $k$ from 1...$m-1$ do
    len = $2^k$
    toSort=Substring[$n$]
    for $i$ from 0...$n-1$ do
        toSort[$i$].originialIndex = $i$
        toSort[$i$].firstHalf = rank[$k-1$][$i$]
        toSort[$i$].secondHalf = rank[$k-1$][$i+len$]
    Sort by firstHalf with secondHalf as a tiebreaker
    for $i$ from 0...$n-1$ do
        if $i=0$ or  toSort[$i$] does not match toSort[$i-1$] then
            rank[$k$][toSort[$i$].originalIndex] = $i$
        else
            rank[$k$][toSort[$i$].originalIndex] = rank[$k$][toSort[$i$].originalIndex-1]
```

We can only store the previous row of the rank matrix to save memory, but that information is useful for other problems.

If we use a radix sort instead of an $O(n \log n)$ sort we can get overall $O(n \log n)$ time.

4 Longest Common Prefix

The longest common prefix (LCP) between two suffixes is the first k characters of both suffixes that match. We can use the matrix from computing the suffix array to compute LCPs. Using the rank values we can compare substrings of length 2^k in O(1) time. If the length of LCP(a,b) is k, then for any x where $2^x \leq k$, rank[x][a]=rank[x][b]. If $2^x \geq k$ then rank[x][a] $\neq$ rank[x][b]. If we iterate from the largest x to 0, the first x that works means that k is at least 2^x. We can move up a and b by 2^x each and try the next lower x. By summing up all the moves we make we can find the length of LCP(a,b). This takes $O(\log n)$ steps per query.

Algorithm 2 LCP

```
if $a=b$ then return $n-a$
lcp=0
for $x$ from $\log n$...0 do
    if rank[$x$][$a$]=rank[$x$][$b$] then
        $a = a + 2^x$
        $b = b + 2^x$
        $lcp = lcp + 2^x$
return lcp
```

5 Problems

5.1 String Search

Given the suffix array of a long text and a search string, find an instance of the search string in the text. There is a way to do this in $O(|search|+\log|text|)$ if you are creative, but $O(|search|\log|text|)$

5.2 Standing Out From the Herd

USACO Platinum, December 2017

Given a lot of strings, find the number of unique substrings that only appear in one string.

5.3 Longest Shared Substring

Given 5 strings, find the longest substring shared by all five strings.