CSE 584A Class 10

Jeremy Buhler

February 22, 2016

1 Sneaky LCP Facts, Part One

We've seen that LCP is an essential element of fast suffix array search.

- How do we compute LCP values?
- Naively, could take time $\Theta(n)$ for LCP of any two suffixes in string of length n.
- Fortunately, it's not actually that hard.
- First, we'll cover "adjacent LCP," the computation of LCP values for adjacent suffixes in a suffix array.
- Then, we'll extend this to "arbitrary LCP."
- *Goal*: compute arbitrary LCP in constant time and reasonable space.

First, let's consider adjacent LCP.

- Let A be our suffix array, and let R be its array of ranks (the inverse array).
- For each $2 \leq i \leq n$, we want to compute $\text{LCP}(A[i-1], A[i])$.
- (That is, LCP of each two adjacent suffixes in sorted order of A.)
- Naively, cost is $\Theta(n^2)$ (by explicit comparisons per suffix).
- Just as in sorting, want to transfer information between suffixes to avoid redundant comparisons.
- Suppose we compare suffixes $\sigma_{A[i-1]}$ and $\sigma_{A[i]}$, and that their LCP is $h > 0$.
- What can we say about suffix $\sigma_{A[i]+1}$?
- Let $k = R[A[i]+1]$, that is, the rank of suffix $\sigma_{A[i]+1}$ in A.
- **Claim**: $\text{LCP}(A[k-1], A[k]) \geq h - 1$.
- (That is, we have a lower bound on LCP of $\sigma_{A[i]+1}$ and its predecessor in A.)
- **Pf**: We know that $\text{LCP}(A[i-1]+1, A[i]+1) = h-1$, since these two suffixes are just $\sigma_{A[i-1]}$ and $\sigma_{A[i]}$ with their first chars removed.

- Now $\sigma_{A[i-1]} < \sigma_{A[i]}$, and their first characters agree, so it must be that $\sigma_{A[i-1]+1} < \sigma_{A[i]+1}$.
- But $\sigma_{A[k-1]}$, which *also* precedes $\sigma_{A[i]+1} = \sigma_{A[k]}$, is its closest predecessor, and so is at least as close in sorted order as $\sigma_{A[i-1]+1}$.
- Conclude that

$$\begin{aligned} \text{LCP}(A[k-1], A[k]) &= \text{LCP}(A[k-1], A[i]+1) \\ &\geq \text{LCP}(A[i-1]+1, A[i]+1) \\ &= h-1. \end{aligned}$$

- This proves the claim. QED

2 Computing Adjacent LCP Values

How does above argument help us compute LCPs?

- Following algorithm is due to Kasai, Lee, Arimura, Arikawa, and Park (CPM 2001).
- Computes adjacent LCP values for all suffixes of S with their predecessors in A.
- Fills in an array L, where $L[j] = \text{LCP}(A[j-1], A[j])$.
- ADJLCP(A, R)

```
h ← 0
for i in 1..n
    k ← R[i]
    if (k = 1)
      L[k] ← null
    else
      j ← A[k − 1]
      while (i + h ≤ n and j + h ≤ n and S[i + h] = S[j + h])
            h + +
      L[k] ← h

    if (h > 0) h − −

return L
```

- *Idea*: h gives us lower bound on each LCP value.
- We use this bound to avoid comparing known common prefixes.
- (When $h = 0$, we do a full comparison, so no harm.)

How fast is this algorithm?

- As usual, let's count character comparisons.
- Every mismatched comparison terminates some iteration of the loop.

- Hence, at most n of these (since i runs from 1 to n).
- To analyze matched comparisons, let's consider what happens to h.
- h is initially 0 and cannot exceed n (it's the length of an LCP).
- Each iteration decrements h by (at most) one.
- Total decrements over all iterations are at most n.
- Each successful comparison increments h by exactly one.
- Now total increments minus total decrements had better be $\leq n$; otherwise, h would take on an invalid value!
- Conclude that total increments is at most $2n$.
- Hence, total char comparisons is at most $3n$, which is $\Theta(n)$. QED

In conclusion, we can compute array L of all *adjacent* LCP values in linear time and space in n. Can look up any needed adj. LCP value in constant time.

3 Sneaky LCP Facts, Part Deux

Adjacent LCP values are not sufficient; we need *arbitrary* LCP values.

- How can we get from adjacent to arbitrary LCPs?
- Suppose we want to compute $\text{LCP}(A[i], A[j])$, for $j > i$.
- We can try to "propagate" adjacent LCP values from i to j.

- If all adjacent LCPs between i and j are long, so is $\text{LCP}(A[i], A[j])$.
- But, if any adjacent LCP in this interval is short, the chain is broken, and we cannot propagate.

Let's formalize this intuition.

- **Claim**:

$$\text{LCP}(A[i], A[j]) = \min_{i<k\leq j} \text{LCP}(A[k-1], A[k]).$$

- **Pf**: by induction on $j - i$.
- **Bas**: if $j = i+1$, then statement is tautology: both sides are exactly LCP(A[i],A[i+1]) .
- **Ind**: let $j = i + m$, $m > 1$.
- By IH, we have that

$$\text{LCP}(A[i], A[j-1]) = \min_{i<k\leq j-1} \text{LCP}(A[k-1], A[k]).$$

- Let $h = \text{LCP}(A[i], A[j-1])$.
- If $\text{LCP}(A[j-1], A[j]) \geq h$, then 1st h chars of $A[j]$ agree with $A[j-1]$, which by assumption agrees with first h chars of $A[i]$.
- Hence $\text{LCP}(A[i], A[j]) = h$.
- If, however, $\text{LCP}(A[j-1], A[j]) = g < h$, then only 1st g chars of $A[j]$ agree with $A[j-1]$.
- Hence $\text{LCP}(A[i], A[j]) = g$.
- Conclude that

$$\text{LCP}(A[i], A[j-1]) = \min(\text{LCP}(A[i], A[j-1]), \text{LCP}(A[j-1], A[j]).$$

  which is what we want. QED

4 Computing Arbitrary LCPs Quickly

We now know how to compute arbitrary LCPs, but not how to compute them fast.

- Let L be array of adjacent LCP values computed above.
- Then for any $i < j$, $\text{LCP}(A[i], A[j]) = \min(L[i+1 \ldots j])$.
- Unfortunately, this still takes time $\Theta(j - i)$, which is $\Theta(n)$ in general.
- We need to accelerate the min over linearly many L values.

Solution: preprocessing!

- Following construction is by Bender and Farach-Colton (2000).
- It is a general accelerator for the *range-minimum query problem*: given an unordered array L of numbers, preprocess L so that in constant time, we can determine $\min(L[p \ldots q])$ for any $p \leq q$.
- Suppose we have a 2D table L', such that

$$L'[i, m] = \min(L[i \ldots i + 2^m - 1]).$$

  for $1 \leq i \leq n$ (ignoring undefined values at end).

- Such a table has $\Theta(n \log n)$ entries, since $m \leq \log n$.
- Now suppose we want to compute $\min(L[p \ldots q])$ for $p < q$.
- Let $\ell = q - p + 1$.
- Let $y = \lfloor \log \ell \rfloor$, and let $z = \ell - 2^y$.
- Observe that

$$\begin{aligned}
\min(L[p \ldots q]) &= \min(\min(L[p \ldots p + 2^y - 1]), \min(L[q - 2^y + 1 \ldots q])) \\
&= \min(\min(L[p \ldots p + 2^y - 1]), \min(L[p + z \ldots q])) \\
&= \min(\min(L[p \ldots p + 2^y - 1]), \min(L[p + z \ldots p + z + 2^y - 1])) \\
&= \min(L'[p, y], L'[p + z, y]).
\end{aligned}$$

- *First correctness observation*: $p + 2^y - 1 \leq q - 2^y + 1$, so the two range minima in the first step always include all of $p \ldots q$. (Check!)
- *Second correctness observation*: the second step is equivalent because

$$\begin{aligned}
p + z &= p + \ell - 2^y \\
&= p + q - p + 1 - 2^y \\
&= q - 2^y + 1.
\end{aligned}$$

  Similar algebra checks the validity of the third step.
- Hence, given L', can compute any LCP value in constant time! Takes only a floor-of-log (which can be done in integer arithmetic), a min, two table lookups, and a bit of arithmetic.
- How long does it take to fill in table L'?
- Do it bottom up starting from L.
- $L'[p, 0] = L[p]$ by definition ($L[p \ldots p + 2^0 - 1]$).
- For the general case, observe that

$$\begin{aligned}
L'[p, y] &= \min(L[p \ldots p + 2^y - 1]) \\
&= \min(\min(L[p \ldots p + 2^{y-1} - 1], \min(L[p + 2^{y-1} \ldots p + 2^y - 1])) \\
&= \min(L'[p, y - 1], L'[p + 2^{y-1}, y - 1]).
\end{aligned}$$

- Hence, each entry of L' can be computed in constant time from previous entries.
- Hence, filling L' takes time $\Theta(n \log n)$.

Conclude that, in time and space $\Theta(n \log n)$, we can build a data structure sufficient to let us compute arbitrary LCP values in constant time.

5 Linear-Space Preprocessing

What happened to our linearity?

- We can build the suffix array in linear space (and in fact linear time, though we have not seen such a construction yet)
- However, it seems to take $\Theta(n \log n)$ time and space to prepare for fast LCP, and hence fast search, using the above construction.
- However, above is only the easiest version of this construction.
- Two ways to improve bounds.
- *Approach 1* (Manber and Myers 1993): precompute only those LCP values needed to implement binary search.
- We don't use arbitrary LCP values in a binary search, only values for intervals of length 2^k for $k \geq 0$ starting at power-of-two boundaries.
- Only linearly many such intervals (think of a full binary tree), so we can enumerate them all in linear time and space.
- *Approach 2* (Bender and Farach-Colton 2000): use more efficient preprocessing for constant-time range-minimum queries on arrays of integers.
- B & F first reduce general problem to special case where successive entries in L have difference of $+/-1$.
- Then, they show that special case can be solved by applying algo of prev section to list of values of size $\Theta(n/\log n)$, plus extra space $\Theta(\sqrt{n} \log n \log \log n)$.
- *Conclusion*: $\Theta(n)$ time/space for setup, constant-time queries!
- (This algorithm is practical, though perhaps a bit hirsute.)