CSE 584A Class 9

Jeremy Buhler

February 17, 2016

1 Basic Suffix Array Matching Algorithm

How do we compare a pattern string $P[1..m]$ to a suffix array A for a string $S[1..n]$?

- Previous answer was just "binary search."
- At each step of binary search, we compare P to some suffix of A.
- Requires $\Theta(\log n)$ comparisons, each of worst-case size $\Theta(m)$.
- Hence, time is $\Theta(m \log n)$.

Can we do better?

- Many character comparisons performed by naive algorithm are redundant.
- **Example**:

- How can we characterize redundancy?
- Binary search maintains a left bound index L and a right bound index R for interval in A that might contain a match to P.
- If both bounding suffixes $A[L]$ and $A[R]$ have a *common prefix*, and P lies between them, then P shares the *same* common prefix.
- Yet, we needlessly re-compare all the characters in this common prefix for every comparison.
- Let's explicitly track common prefixes to avoid redundant comparisons.

2 A Better Binary Search

The following algorithm is also from Manber & Myers.

- Before we get started, a bit of notation.
- For two suffixes σ_i and σ_j of a string S, let LCP(i, j) be the length of their longest common prefix.
- **Example**:

- We are going to assume for now that we can compute this value whenever needed; cost will be studied later.

OK, on with the algorithm.

- Basic structure of algorithm is binary search, like naive algo.
- Following version maintains invariant $\sigma_{A[L]} \leq P < \sigma_{A[R]}$.
- We'll assume there are sentinels at array positions 0, $n + 1$ whose 1st chars are $<$, $>$ first char of P respectively.
- A bit of code:

```
BSEARCH(A, P)
  L ← 0
  R ← n + 1
  while R > L + 1 do
    C ← ⌊(L + R)/2⌋
    if P < σ_{A[C]}
      R ← C
    else                                   ▷ P ≥ σ_{A[C]}
      L ← C
  return L
```

- Define lcpLeft and lcpRight to be lengths of longest common prefixes of query P with $\sigma_{A[L]}$ and $\sigma_{A[R]}$, respectively.
- Assume that, at entrance to while loop above, we know both lcpLeft and lcpRight.
- Suppose LCP$(A[C], A[R])$ > lcpRight.
- Then $\sigma_{A[C]}$ shares a longer prefix match with $\sigma_{A[R]}$ than does query P.
- Both $\sigma_{A[C]}$ and P are $< \sigma_{A[R]}$, so conclude that $P < \sigma_{A[C]}$.
- Hence, binary search may set $R \leftarrow C$ with no character comparisons!

- Similarly, suppose that $\text{LCP}(A[C], A[R]) < \text{lcpRight}$.
- Can argue as above that $P > \sigma_{A[C]}$.
- Hence, binary search may set $L \leftarrow C$ with no character comparisons!
- Finally, suppose $\text{LCP}(A[C], A[R]) = \text{lcpRight}$.
- P could be $>$, $<$, or $= \sigma_{A[C]}$; we can't tell.
- To decide, compare P to $\sigma_{A[C]}$ explicitly starting at position $\text{lcpRight} + 1$ until we find which way to jump.

Let's augment while loop body of BSEARCH to maintain and use lcpLeft, lcpRight.

- Here we go:

```
BSEARCH-LOOPBODY-R
  if lcpRight < LCP(A[C], A[R])
    R ← C                                  ▷ no change in lcpLeft or lcpRight
  else if lcpRight > LCP(A[C], A[R])
    L ← C
    lcpLeft ← LCP(A[C], A[R])              ▷ no change in lcpRight
  else
    i ← lcpRight + 1
    while i ≤ m and P[i] = σ_A[C][i] do
      i++
    if i ≤ m and P[i] < σ_A[C][i]
      R ← C
      lcpRight ← i − 1
    else
      L ← C
      lcpLeft ← i − 1
```

- Important observation: we could just as well have implemented the same hack using L, C, and lcpLeft for tests (exercise!).
- Call that version of the loop body BSEARCH-LOOPBODY-L.

- The full Manber-Myers search procedure is as follows.

```
BSEARCH-MM(A, P)
    L ← 0
    R ← n + 1
    lcpLeft ← 0
    lcpRight ← 0

    while R > L + 1 do
        C ← ⌊(L + R)/2⌋
        if lcpLeft > lcpRight
            do BSEARCH-LOOPBODY-L
        else
            do BSEARCH-LOOPBODY-R

    return max(lcpLeft, lcpRight)
```

- Note that at end of algorithm, returned value gives longest common prefix of P with any suffix of S.
- If this value is $m = |P|$, we found a match (to L or R as appropriate; code not shown.)
- Algo as shown finds leftmost match to P; modification would find rightmost match instead.

3 How Is This Better?

Let's count the number of character comparisons performed by BSEARCH-MM.

- **Claim**: BSEARCH-MM performs at most $m + \log n$ total character comparisons.
- As before, split into successful and failed comparisons.
- Every failed comparison terminates an iteration of the binary search loop.
- At most $\Theta(\log n)$ such iterations, so at most $\Theta(\log n)$ failed comparisons.
- What about successful comparisons?
- Let $z = \max(\text{lcpLeft}, \text{lcpRight})$.
- Explicit character comparisons always start after $P[z]$ (after lcpRight in BSEARCH-LOOPBODY-R, or after lcpLeft in BSEARCH-LOOPBODY-L).
- Each successful character comparison in BSEARCH-LOOPBODY-R increases i, which eventually becomes either lcpLeft or lcpRight.
- Whichever value is set is $>$ the original lcpRight, which is at least the original lcpLeft, so the value z increases.
- Similar argument for BSEARCH-LOOPBODY-L shows that z also increases in that case.

- But $z \leq m$, the length of P.
- Conclude that # successful char comparisons must be $\leq m$. QED

Great, but are we done? Not yet.

- We do $\Theta(m + \log n)$ character comparisons.
- What about the time spent computing LCP values between suffixes of S?
- We will show that, in $\Theta(n \log n)$ space with $\Theta(n \log n)$ preprocessing time for S, we can compute LCP of any two suffixes of S in *constant time*! (We can actually reduce this to $\Theta(n)$ space and preprocessing, but that's too hairy to present here.)
- Implies that BSEARCH-MM is truly $\Theta(m + \log n)$ time.

4 One Further Hack: Substring Matching

Let's talk about the substring matching problem.

- **Problem**: given pattern P and text S, find all *substrings* of P with length $\geq k$ that also appear in S.
- Naive algorithm: use BSEARCH (or BSEARCH-MM) to find longest prefix match between $P[i..m]$ and S for every $1 \leq i \leq m$.
- If this match is $\geq k$, we found a long substring of P in S.
- (Can specify where match occurs as in BSEARCH-MM.)
- Cost is $\Theta(m^2 + m \log n)$. Can we do better?

One "simple" solution: co-sort P and S.

- Create suffix array A from string $P\#S$, where # is a separator distinct from the alphabet of P and S.
- For each suffix $A[i]$ that begins in P, find the nearest suffixes $A[L]$, $A[R]$ with $L < i$, $R > i$ that begin in S.
- If $\text{LCP}(A[i], A[L])$ or $\text{LCP}(A[i], A[R])$ is $\geq k$, we found a k-substring match.
- Assuming constant-time LCP calculations, can return all such i (with locn of at least one match in S) in one linear-time pass over A; i.e., cost is $\Theta(m + n)$.
- (In fact, can also return *all k-matches* in S to each substring of P in time proportional to their number.)
- Probably solution of choice if P is about same size as S, and you can afford to build joint suffix array.

What if you don't want to build a joint array?

- Alternate plan B uses an extension of BSEARCH-MM.
- Suppose you've used BSEARCH-MM to localize P to between suffixes $\sigma_{A[i]}$ and $\sigma_{A[i+1]}$ of S.
- Suppose (say) lcpLeft > 1.
- Then $P[2..m]$ must be lexicographically $\geq \sigma_{A[i]+1}$.
- Moreover, the longest common prefix of these two strings is lcpLeft $- 1$ (since we've removed 1st char of each).
- Similar argument can be made when lcpRight > 1.
- Hence, we can actually start BSEARCH-MM for $P[2..m]$ with the following info already known:
  - nontrivial L and/or R values ($\sigma_{A[i]+1}$ and $\sigma_{A[i+1]+1}$)
  - non-zero lcpLeft and/or lcpRight values
  - (Note that you have to be able to locate suffix $\sigma A[i] + 1$ efficiently in A! Use inverse of A.)
- Note in particular that, from end of one search to beginning of next, max(lcpLeft, lcpRight) decreases by (at most) 1.
- Can extend running time argument for BSEARCH-MM to argue that total number of successful comparisons for running the algo at *all offsets into P successively* is $\Theta(m)$.
- Conclude that, if we run BSEARCH-MM successively for $i = 1..m$ while preserving state between runs, then total cost to find at least one substring match $\geq k$ to P in S whenever it exists is $\Theta(m + m \log n) = \Theta(m \log n)$.
- With constant-time LCP, can enumerate all such matches in time proportional to their number.

Next time – how to compute LCP values in constant time.