CSE 584A Class 2

Jeremy Buhler

January 25, 2016

1 First Task: Exact Matching

A simple, natural problem in stringology is the following:

> Given a *pattern string* P and a large text T, find all positions at which P occurs as a substring of T.

This is the *string matching problem*.

- Typically, T is extremely large. NCBI GenBank contains well over 10^{12} bases of DNA (including WGS data). Its sequence read archive, containing data from next-generation sequencing experiments, has passed 10^{15} bases!
- Typically, P is much smaller. Size may be anywhere from tens to thousands of bases. (Think single sequencing reads, mRNAs, short assembled bits of genome.)

Why might you want to solve this problem?

- If a sequence feature (e.g. a gene) is sufficiently well conserved, it may match some sequence in the database exactly over long stretches.
- You might obtain a sequence in an experiment and want to find out where in the genome it occurs (e.g. short read alignment for mRNASeq, ChIPSeq, or resequencing!).
- You might want to ensure that a sequence contains only one copy of a particular substring (marker or PCR primer design).

Naively, how would you solve this problem?

- Check $T[1 \ldots |P|]$. Is it a match to P?
- Check $T[2 \ldots |P|+1]$. Is it a match to P?
- and so on, and so on ...
- Check $T[|T|-|P|+1 \ldots |T|]$. Is it a match to P?

2 Cost of Exact Matching

What is the worst-case running time of this approach?

- We check $\Theta(|T|)$ starting points in the text.
- At each point, we check up to $|P|$ characters.
- Conclude that worst-case cost is $O(|T||P|)$.
- Can we actually perform $\Theta(|T||P|)$ character comparisons?
- Yes! If, say, $P = a^n$ and $T = a^m$, we read the entire pattern at each starting point.
- Cost is $\Theta(mn)$.

3 Speeding Up Exact Matching

Can we do better than the naive string matching algorithm in the worst case? YES!

- Because P is small, we will first *preprocess* it to uncover its structure.
- What structure? In particular, internal repetitions.
- Knowing when a pattern repeats itself will help us avoid unneeded character comparisons. Example:

We'll begin by talking about what Gusfield calls "fundamental preprocessing" and give a simple string matching algo based on it. First, some definitions.

- **Defn**: Let S be any string of length m. Define z_i to be the length of the longest *prefix match* between $S[1..m]$ and $S[i..m]$.
- Example:

- Call the substring $B_i = S[i..i + z_i - 1]$ the ith *Z-box*.

Z-boxes indicate where a string starts to repeat itself, and for how many chars. What good are they? Here's a simple string matching algorithm for pattern P and text T.

- Form the string $S = P \cdot T$ by concatenating P with T.
- Compute z_i for each offset i into S.
- If $z_i \geq |P|$, then by definition $S[1..|P|] = P = S[i..i + |P| - 1]$. Suppose $i > |P|$. Then $S[i..i + |P| - 1] = T[i - |P|..i - 1]$, a substring of T. Hence, we've found an occurrence of P starting at $T[i - |P|]$.
- Otherwise, there is a mismatch somewhere between $S[1..|P|]$ and $S[i..i + |P| - 1]$, and so no occurrence of P starts at $T[i - |P|]$.

This algorithm is surely correct (we just proved it), but is it fast? Naively, the only way to compute the z_i's is by applying the naive algorithm on S against itself. Can we do better?

4 Fast Computation of Z-Values

Claim: we can compute all Z-values for a string S of length m using at most $2m$ character comparisons.

- For any offset i into S, let ℓ_i and r_i be the left and right endpoints of the nonempty Z-box B_{ℓ_i}, $2 \leq \ell_i \leq i$, whose right endpoint lies *furthest to the right* in S.
- (If no nonempty Z-boxes lie between 2 and i, $\ell_i = r_i = 0$.)
- *Intuition*: what happens if some Z-box $B_\ell = S[\ell..r]$ starting before position i overlaps $S[i]$?

- We know that $S[\ell..r] = S[1..z_\ell]$.
- Therefore, $S[i]$, which lies between ℓ and r, is the same as $S[i - \ell + 1]$.
- So can we start a nonempty Z-box at $S[i]$? Yes iff we *were* able to start one at $S[i - \ell + 1]$!
- All the above reasoning can be accomplished without re-reading $S[i]$. It also holds for any characters past $S[i]$ that are known to be the same as the corresponding chars of $S[1..z_\ell]$.

Idea: extend the ith Z-box B_i as far as possible without re-reading any characters. Use any existing Z-box overlapping i to "look up" the work done before! (Afterwards, we might still have to keep extending B_i with explicit comparisons.)

```
ComputeZs(S)
  ℓ_1 ← 0
  r_1 ← 0
  i ← 2
  while i ≤ m do
    if r_{i-1} ≥ i                                  ▷ a prev Z-box covers i
      k ← i − ℓ_{i-1} + 1
      y_i ← min(z_k, r_{i-1} − i + 1)               ▷ extend as far as possible
    else
      y_i ← 0                                       ▷ no help from previous posns
    z_i ← y_i                                       ▷ extend new Z-box as far as possible explicitly
    while S[i + z_i] = S[1 + z_i] do
      z_i + +
    if z_i > 0 and i + z_i − 1 > r_{i-1}            ▷ if new Z-box extends past old one
      ℓ_i ← i
      r_i ← i + z_i − 1
    else
      ℓ_i ← ℓ_{i-1}
      r_i ← r_{i-1}
    i + +
```

Example:

Is this algo correct?

- Prove that all z_i's are correct by induction on i.
- **Base**: z_2 is computed purely by explicit comparisons, since posn 2 cannot be overlapped by an earlier Z-box (posn 1 cannot be the start of a nontrivial Z-box).
- **Ind**: Suppose $z_{i'}$ correct for $i' < i$.
- Note that y_i is the initial lower bound for z_i. If y_i is always set $\leq$ the correct Z-box length, we are good; any further incrementing of z_i is by explicit comparisons, which are OK.
- Hence, will show that $S[i..i + y_i - 1]$ matches $S[1..y_i]$.

- If first if-test fails ($r_{i-1} < i$), then $y_i \leftarrow 0$. Hence, after test, $S[i..i-1]$ (empty string) matches $S[1..0]$ (empty string), which is surely true.
- Otherwise, let $z_{\ell_{i-1}} = r_{i-1} - \ell_{i-1} + 1$ be the length of the Z-box whose endpoints are ℓ_{i-1} and r_{i-1}.
- By definition of Z-box, $S[\ell_{i-1}..r_{i-1}] = S[1..z_{\ell_{i-1}}]$.
- Hence, $S[i..r_{i-1}] = S[i - \ell_{i-1} + 1..z_{\ell_{i-1}}] = S[k..z_{\ell_{i-1}}]$ (by defn of k in the code).
- Now consider the Z-box $B_k = S[k..k + z_k - 1]$ that starts at position k.
- Remaining proof of correctness considers two cases: either B_k extends at least as far as $S[z_{\ell_{i-1}}]$, or it does not.

- If B_k extends *at least* to $S[z_{\ell_{i-1}}]$, then $S[i..r_{i-1}] = S[k..z_{\ell_{i-1}}] = S[1..z_{\ell_{i-1}} - k + 1]$.
- Hence, we may extend the Z-box B_i at least as far as r_{i-1}, or equivalently, set $y_i = r_{i-1} - i + 1$.
- If, OTOH, B_k ends before $S[z_{\ell_{i-1}}]$, we still know that $S[i..i + z_k - 1] = S[k..k + z_k - 1] = S[1..z_k]$.
- Hence, we may safely set $y_i = z_k$.
- Actual code uses the min of these two values.
- Conclude that y_i is never set $> z_i$. If it is too small, explicit comparisons will extend z_i to the correct value. QED

Why is the algo fast? Let's bound number of char comparisons (we surely do constant work per position otherwise).

- Split accounting into comparisons that *match* and those that *mismatch*.
- We never compute more than one char mismatch per string position i, since any such mismatch ends the computation of z_i.
- Thus, at most m mismatched comparisons overall.
- To bound the number of matching comparisons, note first that $r_i \geq r_{i-1}$ (set equal initially, cannot decrement).

- Moreover, we explicitly match chars while computing the Z-box B_i only if we have already extended B_i past r_{i-1}.
- (Indeed, suppose to the contrary that endpoint of B_i is set strictly before r_{i-1} by first if-test; then it must have ended because of a mismatch (the same one that ended B_k). Hence, further matches are possible only if B_i is initially extended as far as r_{i-1}.)
- Hence, if we compute q character matches, r_i becomes at least $r_{i-1} + q$.
- Conclude that we cannot explicitly compute more than m char matches total without making some $r_i > m$, which is impossible.
- Hence, we compute at most m matches and at most m mismatches, or $2m$ character comparisons total in the worst case! QED

As an immediate corollary, using the Z computations to do string matching between pattern P and text T takes worst-case time $\Theta(|P| + |T|)$, and more specifically at most $2(|P| + |T|)$ comparisons.

5 An Optimistic Average-Case Cost Estimate for DNA

The worst case is important for many text matching applications but isn't terribly realistic for DNA. We don't see whole chromosomes made out of one letter, after all. How do you think the naive method performs in DNA *on average*?

- What does average DNA look like, anyway? We need a model.
- Here's a simple model that is easy to analyze and not totally ridiculous.
- *i.i.d. random DNA* (independent, identically distributed).
- Each base is chosen independently at random.
- For simplicity, assume *uniform* choice: each base appears with probability 1/4.

Now, we can ask: what is the *expected* performance of the naive string matching algorithm when the text T follows the i.i.d. model with equal base freqs?

- Suppose we check P against T starting at some character $T[j]$. What is the expected number of character comparisons?
- For any character of P, the chance that it matches a given character of T is 1/4, independent of all others.
- Hence, the chance that we have exactly k matches followed by a mismatch is

$$\Pr[\text{exactly } k \text{ matches}] = \left(\frac{1}{4}\right)^k \cdot \frac{3}{4}$$

- In other words, the number of matches to P at offset j into T is *geometrically distributed* with parameter $p = 3/4$.

- What is expected number of matches before the first mismatch? $E[k] = (1-p)/p = 1/3$.
- Hence, we compare an average of 4/3 characters (1/3 matches plus 1 mismatch) at each offset into T.
- By linearity of expectation, total *expected* cost of naive matching is about $4|T|/3$ char comparisons.

Limitations of this model?

- How might this cost change if the bases of T did not occur with equal frequencies?
- What if your sequence contains many repeats, and the pattern happens to match part of a repeat?