String Algorithms

Nick Haliday

2012-04-20

1 Intro

Strings are ubiquitous in computing, and one of the most basic problems, often used as a building block in other algorithms, is matching one string or pattern against another. The obvious naïve algorithm is $O(nm)$, which is slow. While Rabin-Karp does well in practice and is easy to code, it is also easy to construct test cases that bring out the same $O(nm)$ worst case performance. This lecture will focus on faster algorithms for the string matching problem.

2 Finite Automata

Finite automata are a useful abstraction for problems relating to sequences. They can be visualized as directed graphs were vertices correspond to states and edges to transitions between these states. There is a start state and a set of accepting states though these may be omitted for convenience. Letters from the input alphabet are associated with edges and a string is processed letter by letter. Deterministic finite automata (DFAs) associate at most one out-edge with any given letter.

If we can build an DFA that matches against a pattern string than we already have a linear time algorithm for string matching. The only problem is building the DFA.

Some notation can be useful, S_i denotes the i-long prefix of S. $S \sqsupset T$ means S is a suffix of T and $S \sqsubset T$ means the same for prefix. $\delta(q, a)$ is the transition function of our DFA, and $\phi(x)$ is the final state function. $\sigma(x)$ denotes for a given pattern P the longest prefix of P that is a suffix of x.

Suppose we have our pattern $P[1..m]$. Then our state set is $\{0, 1, \dots m\}$ and our transition function is $\delta(q, a) = \sigma(P_q a)$. This can be computed naïvely in $O(m^3|\Sigma|)$ where Σ is our alphabet. It is actually possible to compute the transition function in $O(m|\Sigma|)$, but it is easier to understand how to do that after looking at the Knuth-Morris-Pratt algorithm.

3 Knuth-Morris-Pratt

For KMP, we use a different function $\pi[q] = \max\{k : k < q \,\text{and}\, P_k \sqsupset P_q\}$, the longest prefix that is a proper suffix of P_q. Here's the code for the matching algorithm given $\pi[q]$.

```
q = 0
for i in range(1, n + 1):
  while q > 0 and P[q + 1] != T[i]:
    q = pi[q] # look for a suffix that works, or start over with the empty
              # string
  if q == m:
    yield i - m
    q = pi[q]
```

The code to compute $\pi[q]$ is very similar. Essentially it just runs the matching algorithm on the pattern itself.

```
pi[1] = 0
k = 0
for q in range(2, m + 1):
  while k > 0 and P[k + 1] != P[q]:
    k = pi[k]
  if P[k + 1] == P[q]:
    k = k + 1
  pi[q] = k
```

We can use amortized analysis to prove the $O(m)$ running time of these algorithms. First, note $\pi[q] < q$ by definition. So the `k = pi[k]` line always decreases `k`. Also, the total decrease is bounded above by the total increase, which is $O(m)$, so the line `k = pi[k]` can execute at most $O(m)$ times.

4 Z Algorithm

The Z algorithm computes a useful function of string in linear time. Suppose we have a string S, then Z_i is the greatest p such that $S_p = S[i..i + p - 1]$. How could this be useful? One application is string matching. Given the pattern P and the string to be matched S, define $T = P@S$, where @ is a character that does not appear in either string. Then matching positions are simply places where $Z_i = |P|$.

The real problem is finding Z_i. The algorithm attains linear running time by reusing previous values. We can visualize a string's "Z-boxes," all the intervals $[i, i + Z_i - 1]$. As we compute the new Z-boxes, we keep track of the rightmost Z box, $[L, R]$. To compute the new ith Z-box: if $i > R$, then we simply use the definition of Z_i and brute force the value, otherwise we can consider the value Z_k where $k = i - L + 1$. If that Z-box falls strictly within the range $[L, R]$, then we can use that as the new value. Otherwise we brute force again and get a new rightmost interval.

This algorithm is $O(|S|)$ because all operations move the L and R pointers right, and they can only move $O(|S|)$ times.

5 Problems

1. (DEC05G) Farmer John has lined up his N ($1 \leq N \leq 100,000$) cows, looking for a particular substring of K ($1 \leq K \leq 25,000$) cows. Each cow has some number of spots ($1..S$, $1 \leq S \leq 25$), but he does not remember the exact numbers of the K cows, only their relative ranking. For example, he might remember the sequence 1 4 4 3 2 1. This means cows 1 and 6 had the same numbers, as did 2 and 3, and 2 had more spots than 4, who had more spots than 5.

   Find the number substrings of the N cows consistent with the K-long ranking given.

2. (Codeforces, Beta Round 93) Asterix, Obelix, and their new friends Prefix and Suffix want to find a special substring t of larger string s ($1 \leq |s| \leq 10^6$). This must be the longest substring that appears as a prefix, as a suffix, and as neither. Print one such substring or say that none exists.

3. Given a string of length n find the number of palindromes in $O(n)$ time (try using a similar strategy to that of the Z algorithm).

4. Preprocess an m-long regular expression using $O(2^m)$ time and space to match in $O(n)$ time.

5. (CLRS) Give a linear time algorithm for determining whether one string is a cyclic rotation of another.