IA167: String Algorithms **Spring 2014**

Lecture 1: February 17

Lecturer: Alex Popa *Scribes: Jiří Zárevúcky*

1.1 Notation

Σ	The alphabet. A set of characters that any string consists of.
$x[i]$	The i-th character of string x.
$t = t[1] \dots t[n]$	The *text*.
$p = p[1] \dots p[m]$	The *pattern*.
$x[i \dots j] = x[i]x[i+1] \dots x[j-1]x[j]$	A substring of x starting at i and ending at j.
	E.g. $t[1 \dots i]$ is a prefix of t, $p[i \dots m]$ is a suffix of p.
$\Pr[x]$	Probability that x is true, as a fraction ($0 \leq \Pr[x] \leq 1$).

1.2 The problem

Problem 1 (Exact pattern matching)**.** *Given two strings, text t and pattern p, find positions of every substring in t identical to p.*

We can formulate the problem more precisely: given $t = t[1] \dots t[n]$ and $p = p[1] \dots p[m]$, find every i for which it holds that $t[i \dots i+m-1] = p$.

From this formulation, we can make a naive algorithm simply by iterating over every possible i. It can range from 1 to $(n-m+1)$, and checking equality of two m-character strings takes $\mathcal{O}(m)$ operations, resulting in total complexity of $\mathcal{O}(n \cdot m)$.

Exercise: Implement this algorithm in your favorite language.

1.3 Rabin-Karp algorithm

The naive algorithm suffers from poor time complexity because it can potentially process the same portion of the text many times. Rabin-Karp algorithm tries to remove this problem by comparing small *fingerprints* in place of longer strings. We denote a fingerprint of p as $\phi(p)$.

Furthermore, this fingerprint is defined in such a way that $\phi(t[i+1 \dots i+m])$ can be computed from $\phi(t[i \dots i+m-1])$ in $\mathcal{O}(1)$ time, regardless of m. This is called a *rolling hash function.*

The fingerprint function is selected randomly from a family of similar functions, which makes the Rabin-Karp a randomized algorithm. If two strings have different fingerprints, we immediately know they are different, so we can avoid checking the equality character by character. However, since many strings can share the same fingerprint (this is always the case, as we want the fingerprint to be much shorter than the string), it is possible for some positive answers to be false positives. It is possible to naively check every positive result for correctness.

Omitting this check results in a *Monte Carlo algorithm* with $\mathcal{O}(n+m)$ worst-case complexity, but a small probability of erroneous result. Including the check results results in a *Las Vegas algorithm*, which has a complexity of $\mathcal{O}(n+m+k \cdot m)$, where k is the number of possible matches according to the fingerprint function, but zero probability of error. This can be up to $\mathcal{O}(n \cdot m)$ in the worst case.

1.3.1 The fingerprint function

The algorithm as originally presented selects ϕ at random as following:

- Have a fixed large integer T.
- Select uniformly at random a prime $p < T$.
- Select ϕ to be

$$\phi(s) = \sum_{i=1}^{|s|} s[i] \cdot 2^{|s|-i} \ \mathbf{mod}\ p$$

We are assuming that the characters of the alphabet have a numerical representation, so that $s[i]$ is in fact a natural number as well as a character.

Example:

$$s = 01001, p = 7$$
$$\phi(s) = (0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)\ \mathbf{mod}\ 7 = 9\ \mathbf{mod}\ 7 = 2$$

In the Rabin-Karp algorithm, we need to compute the fingerprint for every m-character substring of the text, in order to match it against the pattern. Fortunately, we only need to compute the full formula for the first substring. This is because the next substring only differs from the last in adding one character on one side, and removing one character on the other side.

Example:
Suppose for simplicity that the string is a decimal number, and the fingerprint is the value of this number modulo a fixed prime.
(A value of a decimal string is a sum very similar to the above fingerprint function.)

$$s = \text{“415926”}, m = 5$$

$$s_1 = \text{“41592”}, s_2 = \text{“15926”}$$

$$\phi(s_1) = 41592 \ \mathbf{mod}\ 97 = \cdots = 76$$

$$\phi(s_2) = 15926 \ \mathbf{mod}\ 97 = ((41592 - 4 \cdot 10^m) \cdot 10 + 6) \ \mathbf{mod}\ 97 = (\phi(s_1) - s_1[1] \cdot 10^m) \cdot 10 + s_2[m] = 18$$

This simple trick makes it easy to compute each next fingerprint in constant time.

1.3.2 *Probability of error

It is somewhat more difficult to reason about uniformly random primes. In order to show that the chance of collision is small, we demonstrate the idea on a slightly different fingerprint function.

Alternative fingerprint:

1. fix a large prime p
2. choose uniformly randomly a number $r \in \{1, ..., p-1\}$

$$\phi_r(s) = \sum_{i=1}^{|s|} s[i] \cdot r^{|s|-i} \ \mathbf{mod}\ p$$

Theorem 1.1 (Lagrange's)**.** *A polynomial with degree d modulo a prime number p has at most d roots.*

Proof: See `http://en.wikipedia.org/wiki/Lagrange%27s_theorem_(number_theory)` ■

Theorem 1.2. *Suppose we have fixed s, s', $|s| = |s'|$, $s \neq s'$, and random r. Then*

$$\Pr[\phi_r(s) = \phi_r(s')] \leq \frac{|s|}{p-1}$$

Proof: The probability that s and s' have the same fingerprint is the same as the probability that r we chose is a root of the polynomial $(\phi_r(s) - \phi_r(s'))$ (where r is the variable). This polynomial has a degree of $|s| - 1$, which is easy to see from the definition of ϕ_r.

By the Lagrange's theorem, there are less than $|s|$ roots, but there are $p-1$ choices of r, so for any pair s, s', the chance cannot be more than $\frac{|s|}{p-1}$. ■

Consequently, for a big enough p, the probability of collision is very small. For the original fingerprint, the analysis is more complicated but the result is similar.

1.4 Knuth-Morris-Pratt algorithm

KMP algorithm improves on the naive algorithm using a different idea than Rabin-Karp algorithm. Let us illustrate the problem with an example.

Example:
Consider the following alignment of p = "aaaabaaab" against a text.

```
aaaaaaabaaaabaaabaaa
   aaaabaaab
```

Only the last character of the pattern is mismatched. However, in the naive algorithm, we shift by one and all those previously matched characters are read and checked again. This is the reason for the algorithm's quadratic complexity.

Ideally, we should not have to re-read characters that were once matched, because we have already seen them and we know they are a prefix of the pattern. We can achieve that by shifting the pattern such that the already matched characters match the pattern on its new position.

This means we are looking for a proper suffix of the matched text that is a prefix of p. In fact, we want to find the longest such suffix, as it corresponds to the shortest shift length. Since the characters are known to match the pattern, the shift for any given matched prefix of p is otherwise independent of the text.

Example:

```
aaaaaaabaaaabaaabaaa
   aaaabaaab
        aaaabaaab
```

In case there is no match before mismatch, we just shift by one:

```
amalimalamalimalo
malimalo
 malimalo
     malimalo
        malimalo
         malimalo
```

Definition 1.3. *A border of a word w is an index $i < |w|$ such that*

$$w[1 \dots i] = w[|w| - i + 1 \dots |w|]$$

Informally: It is the length of a prefix that is also a suffix at the same time. The suffixes we use when shifting the pattern are the longest borders of prefixes of p, and as such we can precompute them in a table for constant-time lookup.

Let us denote the longest border of $p[1 \dots i]$ to be $\Pi[i]$. The following lemma formalizes what we intuitively see from the above description.

Lemma 1.4. *The longest suffix of $p[1 \dots l]c$ that is also a prefix of p is:*

1. *$p[1 \dots l+1]$, if $c = p[l+1]$.*
2. *The longest suffix of $p[1 \dots \Pi[i]]c$ which is a prefix of p, otherwise.*

1.4.1 The search

Suppose that we already have the table Π ready. We keep index i_t in the text and index i_p in the pattern, both denoting the last matched character (or 0 if no character was matched yet).

- If $t[i_t + 1] = p[i_p + 1]$, then we simply advance both by one.
- Otherwise, if $i_p > 0$, we assign to i_p the value $\Pi[i_p]$, "shifting" the pattern.
- Finally, if neither of the two conditions hold, we just advance i_t by one.

This is repeated until either i_t exceeds n, in which case the search failed, or i_p reaches m, in which case the match is reported and i_p is possibly set to $\Pi[m]$ to look for the next match. Note that advancing both indices together corresponds to the pattern staying in place, while just decreasing i_p or just increasing i_t corresponds to shifting the pattern forward.

Theorem 1.5. *The search takes $\mathcal{O}(n)$ operations.*

Proof: i_t never decreases and is bounded by n. The only case in which i_t is not increased is a case in which i_p is decreased, but i_p only ever increases together with i_t. That means the number of i_p decreases may never be larger than the number of i_t increases, which is at most n. ■

1.4.2 Precomputing borders

That leaves us the problem of computing Π. Fortunately, that is quite easily achievable in $\mathcal{O}(m)$ time:

For the prefix $p[1 \dots 1]$, $\Pi[1] = 0$ (trivial).

Suppose we know $\Pi[1], \dots, \Pi[i]$. The longest border $\Pi[i+1]$ can be 0, or it can be some border of $p[1 \dots i]$ extended by one character. Borders naturally nest – every border of a string except the longest one is also a border of the longest border substring. Thus Π already contains all borders of $\Pi[i]$, not just the longest one. We can simply iterate over them by repeatedly applying Π to i.

The proof that iterating over borders does not break the overall linear complexity is similar to the analysis for search. The border can only increase by one per step ($\Pi[i+1] \leq \Pi[i] + 1$), so it cannot decrease more than m times during the entire computation. In fact, if you look carefully, you can see that precomputing is very similar to the search algorithm, the difference being only in information stored in every step.

1.5 Boyer-Moore Algorithm

Exercise: Assume that the pattern does not contain the same character twice. Modify the naive algorithm to have $\mathcal{O}(n)$ time complexity.

Exercise: Modify the algorithm from the previous exercise to have $\mathcal{O}\left(\frac{n}{m}\right)$ complexity for an infinite class of inputs, while still being correct for all patterns that do not contain the same character twice.

In the exercise, we see that for some patterns, the search can take much less than n character comparisons. The Boyer-Moore Algorithm takes advantage of this, by matching the pattern against the text from right to left. The result is an algorithm that can run as fast as $\mathcal{O}\left(\frac{n}{m}+m\right)$ for some inputs, while still being $\mathcal{O}(n+m)$ in the worst case. It uses a combination of two (sometimes three) independent rules.

1.5.1 Bad character rule

The first rule is that every time a mismatch occurs, the pattern is shifted such that the mismatched character is aligned to the matching character in the pattern. If there is no such character in the pattern, the pattern is shifted beyond the mismatched character.

Example:

```
aaaaabaaabaaaaaaaaa
aabaaaaa
   aabaaaaa
       aabaaaaa


aaaaaacabababaaaaaaa
   ababab
       ababab
```

1.5.2 Good suffix rule

The idea behind the *good suffix* rule is that when a mismatch occurs, the pattern is shifted such that the matched portion of the text matches the shifted pattern. I.e. when a suffix $s = p[i \ldots m]$ matches the text, and $p[i-1]$ mismatches, we shift to the substring $p[j \ldots k] = s$, $k < m$, where k is the largest possible. If there is no such string, we shift p to the longest suffix of the matched portion that is a prefix of p.

Example:

```
aaaaaabaaacaa
aabaaacaa
   aabaaacaa
    aabaaacaa
```

1.5.3 *Galil rule

When using just the first two rules, the algorithm has worst-case linear complexity only for cases when the pattern does not appear in the text. To achieve linear complexity for all cases, we need one more rule.

See `http://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string_search_algorithm#The_Galil_Rule`.