max planck institut
informatik

Introduction and exact pattern matching I

Algorithms on Strings

Paweł Gawrychowski

April 24, 2013

Outline

Exam

- There **will** be an exam!
- ...but your overall grade will depend on the homeworks (40%) and programming assignments (20%), too.
- The exam will be oral. I will prepare a list of *x* questions, you will get 3 of them at random. The goal of the questions will be to check if you have some general understanding, so more like "explain what a suffix array is" than "reconstruct the details of this one technical lemma that was mentioned in one the lectures".

Problemsets

- Each problemset will contain at least one programming problem. You **will** be able to get a good grade without touching the keyboard, but my advice is that you should try to solve at least some of them. You might realize that it is surprisingly fun!
- For programming problems, we will use an automated judging system called SPOJ (`www.spoj.pl`).

Tutorials

- First of all, we need the tutor!
- ..and then we should choose the time.
- To get the credit for your solutions you should write them up. There will be at least one week between the time I publish the next problemset and its deadline.
- During the next tutorial after the deadline we will discuss the solutions. Please consider attending even if you have solved everything, say to present your solutions to the others.
- **Please** join the discussion group to get the news.

Lectures

- All remaining lectures will be from 14:15 to 16:00.
- I will publish the slides after each lecture.
- The content will be based on the following books:
  - "Jewels of Stringology" by Crochemore and Rytter
  - "Algorithms on Strings, Trees, and Sequences" by Gusfield
  - "Pattern Matching Algorithms" by Apostolico and Galil
- This is the first time I'm teaching this (well, any) course. So, do interrupt me, ask questions, and let me know if you think I should do something in a different way.

We will consider a number of problems concerning strings. A string is just a sequence of characters from some finite alphabet Σ, or in other words $\in \Sigma^*$.

Convention

- we use w, u, v to denote different strings.
- ϵ denotes the empty string.
- $|w|$ is the length of w.
- The alphabet Σ is usually fixed, say $\Sigma = \{0, 1\}$, or $\Sigma = \{a, b, \ldots, z\}$. Or $\Sigma = \{\circ, \triangle\}$!
- Sometimes we will consider larger alphabets. Then a string w is treated as a sequence of integers from $1, 2, \ldots, |w|$.

Words are just a different name for strings.

Convention, continued

- $w[i]$ is the i-th character of w. We start numbering from 1.
- $w[1..n]$ means that $w = w[1]w[2]\ldots w[n]$, where $n = |w|$.
- A prefix of w is any fragment $w[1..j]$.
- A suffix of w is any fragment $w[i..n]$.
- A subword (or infix) of w is any fragment $w[i..j]$.

Example

Say that $w = \texttt{algorithmsonstrings}$, and look at the whiteboard.

Exact pattern matching

Input: text $t[1..n]$ and pattern $p[1..m]$.
Output: does p occur in t?

In other words, we want to find i such that $w[i..i + |p| - 1] = p$, or detect that there is none.

Naive approach

Check all possible i one-by-one. For each of them verify if we have an occurrence there, i.e., if $p[1] = w[i], p[2] = w[i+1], \ldots$.

Example

Lets try this for $w = \texttt{aaaaaaaaab}$ and $p = \texttt{aaab}$.

So maybe we should rather check if $p[m] = w[i+m-1]$, $p[m-1] = w[i+m-2], \ldots$?

Example

Lets try this for $w = \texttt{aaaaaaaaaaaaaaaaaaaa}$ and $p = \texttt{aaabaaa}$.

Clearly, the naive method performs nm comparisons in the worst case. This is clearly very slow in the worst possible case.

We need some clever method to quickly check if $p = w[i..i+m-1]$. In other words, we want to quickly test equality of two BIG objects. We will develop a method which is:

- fast,
- simple,
- ...and incorrect, sometimes. But it won't happen very often.

Monte Carlo algorithm

A randomized algorithm is a false-biased Monte Carlo algorithm for a given decision problem if for any instance it:

- returns NO, and in such case we are sure that the answer to the problem is NO,
- returns YES, and in such case the probability that the answer to the problem is really YES is **big**, say $\frac{99}{100}$.

The bound on the running time should be worst-case.

We will construct a linear time Monte-Carlo algorithm for exact pattern matching.
Say that $\Sigma = \{a, b\}$ and treat the pattern (and the text) as two very long numbers:

aaaaaaaaaaaaaaaaaaaaaaaaa

aaaaabaaaaa

We would like to check if the green number is the same as the red number. Both of them potentially consist of **many** digits. For starters we can check if they have the same remainder modulo $q = 7$.

$$00000100000 = 4 \pmod 7$$
$$00000000000 = 0 \pmod 7$$

We will construct a linear time Monte-Carlo algorithm for exact pattern matching.
Say that $\Sigma = \{a, b\}$ and treat the pattern (and the text) as two very long numbers:

000000000**00000000000**000000000

00000100000

We would like to check if the green number is the same as the red number. Both of them potentially consist of **many** digits. For starters we can check if they have the same remainder modulo $q = 7$.

$$00000100000 = 4 \pmod 7$$
$$00000000000 = 0 \pmod 7$$

If the numbers have different remainders modulo q, they cannot be the same! Hence we don't have to check them digit-by-digit. What is they have the same remainder? We revert to the naive digit-by-digit verification.

Why this is useful?

If we take, say, $q = 10^9 + 7$, it seems reasonable that it won't happen very often that two numbers have the same remainders, but are different.

"Seems reasonable" doesn't sound very convincing. We will use a randomized approach. Checking if two numbers are equal is equivalent to checking if their fingerprints $\phi(w[i..i+m-1])$ and $\phi(p[1..m])$ are the same.

Karp-Rabin-style fingerprints

$$\phi(S) = \sum_{k=1}^{|S|} S[k]2^{|S|-k} \bmod q$$

Lemma

If $r \in \{1, 2, \ldots, q-1\}$ is chosen uniformly at random, the probability that $\phi_r(S) = \phi_r(S')$ even though $S \neq S'$, is at most $\frac{|S|}{q-1}$.

Proof? See the whiteboard.

"Seems reasonable" doesn't sound very convincing. We will use a randomized approach. Checking if two numbers are equal is equivalent to checking if their fingerprints $\phi(w[i..i+m-1])$ and $\phi(p[1..m])$ are the same.

Karp-Rabin-style fingerprints

$$\phi_r(S) = \sum_{k=1}^{|S|} S[k] r^{|S|-k} \bmod q$$

Lemma

If $r \in \{1, 2, \ldots, q-1\}$ is chosen uniformly at random, the probability that $\phi_r(S) = \phi_r(S')$ even though $S \neq S'$, is at most $\frac{|S|}{q-1}$.

Proof? See the whiteboard.

Our algorithm selects a prime $q > 2mn$. Then it chooses a random $r \in \{1, 2, \ldots, q-1\}$ and compares all fingerprints $\phi_r(w[1..m])$, $\phi_r(w[2..m+1])$, $\phi_r(w[3..m+2])$ with $\phi_r(p)$. It returns YES if and only if two fingerprints are the same.

Lemma

The probability that there is **any** false positive, i.e., i such that $\phi_r(w[i..i+m-1) = \phi_r(p)$ even though $w[i..i+m-1] \neq p$, is at most $\frac{1}{2}$.

Proof? See the whiteboard.

By choosing a larger q we can decrease the error probability. Say, choose $q = 1000mn$.

OK, but how to compute all those fingerprints?

Horner's rule

Say that $q = 7$, then $\phi_2(101101) =?$

Hence we can compute each ϕ_r in $O(m)$ time. But this is way too slow to compute all $\phi_r(w[i..i+m-1])$! Well, it is fast enough to compute $\phi_r(p)$, though.

We can avoid recomputing every $\phi_r(w[i..i+m-1])$ from the scratch.

$$100010000\,11000000001\,100000110$$

$$11000000001 = 4 \pmod 7$$
$$10000000011 = 1 \pmod 7$$

$$10000000011 = (11000000001 - 10000000000) * 10 + 1 \pmod 7$$

$\phi_r(w[i+1..i+m])$ can be computed in $O(1)$ time using:

- $\phi_r(w[i..i+m-1])$,
- $r^{m-1} \mod q$.

We can avoid recomputing every $\phi_r(w[i..i+m-1])$ from the scratch.

$$1000100001\,10000000011\,00000110$$

$$11000000001 = 4 \pmod 7$$
$$10000000011 = 1 \pmod 7$$

$$10000000011 = (11000000001 - 10000000000) * 10 + 1 \pmod 7$$

$\phi_r(w[i+1..i+m])$ can be computed in $O(1)$ time using:

- $\phi_r(w[i..i+m-1])$,
- $r^{m-1} \mod q$.

This gives us a linear time Monte Carlo algorithm. But not being sure if the reported occurrences are really occurrences is somehow disturbing.

Las Vegas algorithm

A randomized algorithm is a Las Vegas algorithm for a given decision problem if it always returns the correct answer, but its running time is a random variable (with a finite expected value).

The following is a constant time Las Vegas algorithm: repeat drawing a number $r \in \{0, 1\}$ as long as $r = 0$.

We will construct an expected linear time Las Vegas algorithm for exact pattern matching.
We simply verify each of the reported matches, and stop as soon as one of them turns out to be a valid occurrence.

Lemma

The expected number of false positives, i.e., i such that $\phi_r(w[i..i+m-1) = \phi_r(p)$ even though $w[i..i+m-1] \neq p$, is at most $\frac{(n-m+1)m}{q-1}$.

Proof? See the whiteboard.

If $q > m^2$...

...then the expected number of false positives is at most $\frac{n}{m}$. Each of them can be verified in $O(m)$ time, hence the whole verification takes $O(n)$ time. Computing all fingerprints takes $O(n)$ time, too, so the whole running time is linear.

What if we want to report all occurrences?

Would the running time be still linear?

This was a Karp-Rabin-style algorithm, not **the** Karp-Rabin algorithm. In the original version they used a different hashing scheme:

$$\phi(S) = \sum_{k=1}^{|S|} S[k]2^{|S|-k} \bmod q$$

for a random prime q.

This is a reasonable scheme, too, but to analyze it you need to know something about the distribution of primes.

Other hashing scheme that people sometimes use is:

$$\phi(S) = \sum_{k=1}^{|S|} S[k]2^{|S|-k} \bmod 2^{\ell}$$

Please don't do that!

All this randomization was fun, but:

- we don't want to gamble, even if the chances are high,
- computing the fingerprints is not that quick, anyway (unless we calculate modulo a power of 2, but that's not the brightest idea).

We will show that the problem can be solved in deterministic linear time. The algorithm will be very short, but understanding why and how it works will take us a little bit.

```
1  int find(const string &t,const string &p) {
2    int n=t.size(),m=p.size();
3    int pi[p.size()+1];
4    pi[0]=pi[1]=0;
5    for (int i=2,j=0; i<=m; i++) {
6      while (j && p[i-1]!=p[j]) j=pi[j];
7      pi[i]=j+=p[i-1]==p[j];
8    }
9    for (int i=0,j=0; i<t.size(); i++) {
10     while (j && t[i]!=p[j]) j=pi[j];
11     j+=t[i]==p[j];
12     if (j==m) return i-m+1;
13   }
14   return -1;
15 }
```

Short, simple, and completely unreadable!