String Matching

Thanks to

Prof. Piotr Indyk and Carola Wenk

String Matching

- Input: Two strings T[1...n] (text) and P[1...m] (pattern), containing symbols from alphabet Σ
- Goal: find all "shifts" 1≤ s ≤n-m such that T[s+1...s+m]=P
 In other words: Finds all shifts of a window of length *m* inside the text *T*, so the context of the window is identical to the pattern *P*.
- Example:
 - $-\Sigma = \{,a,b,\ldots,z\}$
 - T[1...18]="to be or not to be"
 - P[1..2]="be"
 - Shifts: 3, 16

Simple Algorithm

for $s \leftarrow 0$ to $n \cdot m$ $Match \leftarrow 1$ for $j \leftarrow 1$ to mif $T[s+j] \neq P[j]$ then $Match \leftarrow 0$ exit loop if Match=1 then output s

Results

- Running time of the simple algorithm:
 - Worst-case: O(nm)
 - Average-case (random text): O(n)
- Is it possible to achieve O(n) for any input ?
 Knuth-Morris-Pratt'77: deterministic
 - Karp-Rabin'81: randomized

Karp-Rabin Algorithm

A very elegant use of an idea that we have encountered before, namely...
 HASHING !

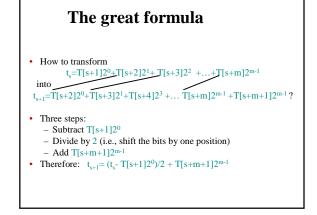
• Idea:

- Hash all substrings *T*[1...*m*], *T*[2...*m*+1], *T*[3...*m*+2], etc.
- Hash (details later) the pattern *P[1...m]*
- Report the substrings that hash to the same value as P
- Problem: how to hash n-m substrings, each of length m, in O(n) time ?

Implementation

• Attempt I:

- Assume $\Sigma = \{0,1\}$
- Think about each $T^s=T[s+1...s+m]$ as a number in binary representation, i.e., $t_s=T[s+1]2^0+T[s+2]2^1+...+T[s+m]2^{m-1}$
- Find a fast way of computing t_{s+1} given t_s
- Output all s such that t_s is equal to the
- number p represented by P



Algorithm

- Can compute t_{s+1} from t_s using 3 arithmetic operations
- Therefore, we can compute all $t_0, t_1, ..., t_{n-m}$ using O(*n*) arithmetic operations
- We can compute a number corresponding to P using O(m) arithmetic operations
- Are we done ?

Problem

- To get O(*n*) time, we would need to perform each arithmetic operation in O(1) time
- However, the arguments are m-bit long (and we have 32/64 bits machine) !
- It is unreasonable to assume that operations on such big numbers can be done in O(1) time
- We need to reduce the number range to something more manageable

Warm-up

- $((x \mod q) + (y \mod q)) \mod q = (x+y) \mod q$
- $((x \mod q) (y \mod q)) \mod q = (xy) \mod q$
- $(ax+b \mod q) = ((a \mod q) (x \mod q) + (b \mod q)) \mod q$
- Every integer x can be uniquely represented as $x=p_1^{e1}p_2^{e2}...p_k^{ek}$ where
 - 1. p_i is a prime, and
 - 2. e_i is an integer
 - 3. $k \le \log_2 x$ since each $p_i \ge 2$

Hashing

- We will instead compute
- $t'_s = T[s+1]2^0 + T[s+2]2^1 + \dots + T[s+m]2^{m-1} \mod q$ where q is an "appropriate" prime number
- One can still compute t'_{s+1} from t'_s :
- $t'_{s+1} = (t'_s T[s+1]2^0) * 2^{-1} + T[s+m+1]2^{m-1} \mod q$
- If q is not large, i.e., has $O(\log n)$ bits, we can compute all t'_s (and p') in O(n) time

Problem

- Unfortunately, we can have false positives,
 i.e., *T^s*≠*P* but *t*'_s=*p*'
 - (to discover a single false positive, we spend O(m) time)
- Need to use a random q
- We will show that the probability of a false positive is small → randomized algorithm

False positives

- Consider any *t_s≠p*. We know that both numbers are in the range {0...2^{*m*}-1}
- How many primes q are there such that $t_s \mod q = p \mod q \equiv (t_s \cdot p) = 0 \mod q$?
- Such prime has to divide $x = (t_s p) \le 2^m$
- Represent $x = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$, p_i prime, $e_i \ge l$
- Since $2 \le p_i$, we have $2^k \le x \le 2^m \rightarrow k \le m$
- There are $\leq m$ primes dividing *x*

Algorithm

- Let \prod be a set of 2nm primes, each having $O(\log n)$ bits (not generated explicitly)
- Choose q uniformly at random from \prod
- Compute *t*'₀, *t*'₁,, and *p*'
- For each shift *s*, the probability that $t'_s = p'$ while $T^s \neq P$ is at most $\log t_s / |\prod| = m/2nm = 1/2n$
- If $t'_s = p'$, we check if $t_s = p$ by checking each char. Takes time O(m). Altogether O(n)
- The probability of *any* false positive is at most $(n-m)/2n \le 1/2$

Geometric Hashing and other problems of shape matching

- This algorithm is an example of general idea:
 - Given a library of (many) shapes $T_{l'}, T_2...T_r$. Preprocess such that given a query pattern *P*, find the most similar shape.
 - Checking for given T_i if it is similar to P is expensive.
 Idea: Using hashing for <u>filtering</u> the shapes that need to
 - be checked: Compute hash values h(T) = h(T) and h(D) and
 - Compute hash values $h(T_1)...h(T_r)$, and h(P), and check if T_i matches P only if $h(P) = h(T_i)$.