String Matching

Thanks to
Prof. Piotr Indyk and Carola Wenk

Simple Algorithm

for \( s \leftarrow 0 \) to \( n-m \)
   \( Match \leftarrow 1 \)
for \( j \leftarrow 1 \) to \( m \)
   if \( T[s+j] \neq P[j] \) then
      \( Match \leftarrow 0 \)
      exit loop
if \( Match = 1 \) then output \( s \)

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?

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

Implementation

- Attempt I:
  - Assume \( \Sigma = \{0,1\} \)
  - Think about each \( T=T[s+1...s+m] \) 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 \( P \)
The great formula

- How to transform $t = T[s] + T[s+1] + T[s+2] + \ldots + T[s+m]$ into $t = T[s+1] + T[s+2] + \ldots + T[s+m+1]$?

  Three steps:
  - Subtract $T[s+1]$
  - Divide by 2 (i.e., shift the bits by one position)
  - Add $T[s+m+1]$

  Therefore: $t = (t - T[s+1]) / 2 + T[s+m+1]$

Algorithm

- Can compute $t_{s+1}$ from $t_s$ using 3 arithmetic operations
- Therefore, we can compute all $t_0, t_1, \ldots, t_{n-m}$ 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

Hashing

- We will instead compute $t'_{s+1} = T[s+1] + T[s+2] + \ldots + T[s+m] \mod q$ where $q$ is an “appropriate” prime number
- One can still compute $t'_{s+1}$ from $t'_s$:
  $t'_{s+1} = (t'_s \cdot T[s+1]) / 2 + T[s+m+1] \mod q$
- 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., $T \neq P$ 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

Warm-up

- $((x \mod q) + (y \mod q)) \mod q = (x+y) \mod q$
- $((x \mod q) \cdot (y \mod q)) \mod q = (x \cdot y) \mod q$
- $(x \cdot y \mod q) = (a \mod q) \cdot (x \mod q) + (b \mod q) \mod q$

Every integer $x$ can be uniquely represented as $x = p_1^{e_1} p_2^{e_2} \ldots p_k^{e_k}$ where
1. $p_i$ is a prime, and
2. $e_i$ is an integer
3. $k \leq \log_2 x$ since each $p_i \geq 2$
**False positives**

- Consider any $t \neq p$. We know that both numbers are in the range $\{0, \ldots, 2^m-1\}$.
- How many primes $q$ are there such that $t \mod q = p \mod q = (t-p) \mod q$?
- Such primes have to divide $x = (t-p) \leq 2^m$.
- Represent $x = p_1^{e_1}p_2^{e_2} \cdots p_k^{e_k}$, $p_i$ prime, $e_i \geq 1$.
- Since $2 \leq p_i$, we have $2^k \leq x \leq 2^m \Rightarrow k \leq m$.
- There are $\leq m$ primes dividing $x$.

**Algorithm**

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

**Geometric Hashing and other problems of shape matching**

- This algorithm is an example of general idea:
  - Given a library of (many) shapes $T_1, T_2, \ldots, T_r$.
  - Preprocess such that given a query pattern $P$, find the most similar shape.
  - Checking for given $T$ if it is similar to $P$ is expensive.
  - Idea: Using hashing for filtering the shapes that need to be checked.
  - Compute hash values $h(T_1), \ldots, h(T_r)$, and $h(P)$, and check if $T_i$ matches $P$ only if $h(P) = h(T_i)$.