String Matching

Thanks to
Prof. Piotr Indyk

String Matching

- Input: Two strings $T[1…n]$ and $P[1…m]$.
- Notation: $T[i..j]$ is the part of the string starting at $T[i]$ and ending at $T[j]$.
- Example $T[1…18] = "to be or not to be"$, then $T[3..5] = "be"$
- $T$ and $P$ containing symbols from alphabet $\Sigma$
- Goal: find all “shifts” $1 \leq s \leq n-m$ such that $T[s+1…s+m] = P$
- Example:
  - $\Sigma = \{a,b,…,z\}$
  - $T[1…18] = "to be or not to be"$
  - $P[1…2] = "be"$
  - Shifts: 3, 16

Simple Algorithm

\[ [t_{o}] \text{ be or not to be} \]
\[ s=0 \]

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

\[ [t_{o}] \text{ be or not to be} \]
\[ s=0 \]

- A very elegant use of an idea that we have encountered before, namely… HASHING!
- Idea:
  - Hash all substrings $T[1…m], T[2…m+1], T[3…m+2], \ldots$ etc.
  - Hash 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: $\Theta(mn)$
  - 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

\[ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ |
\]
\[ ^{8} \ 4 \ 2 \ 1 \]
\[ = 7. \]
\[ s=0 \]

- Think about each $T_s = T[s+1…s+m]$ as a number in binary representation, i.e., $t_s = T[s+1]2^0 + T[s+2]2^1 + \ldots + T[s+m]2^{m-1}$
- 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_s = T[s+1]2^0 + T[s+2]2^1 + ... + T[s+m]2^{m-1}
into
t_{s+1} = T[s+2]2^0 + T[s+3]2^1 + ... + T[s+m+1]2^{m-1}?

- Three steps:
  - Subtract T[s+1]2^0 (this is the least-significant bit. It is either 0 or 1)
  - Divide by 2 (i.e., shift the bits by one position)
  - Add T[s+m+1]2^{m-1}

- Therefore: 
t_{s+1} = (t_s - T[s+1]2^0)/2 + T[s+m+1]2^{m-1}

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!
- 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

010001110101000111

- We will instead compute
  \[ t'_s = h(T[s+1:s+m]) = \{ T[s+1]2^0 + T[s+2]2^1 + ... + T[s+m]2^{m-1} \} \mod q \]
- Where q is an "appropriate" prime number
- Compute \( 2^{-1} \mod q \) and \( 2^{m-1} \mod q \)
- To compute \( t'_{s+1} \) from \( t'_s \):
  \[ t'_{s+1} = \{ (t'_s - T[s+1]2^{-1} \mod q + T[s+m+1]2^{m-1} \mod q) \mod q \} \mod q \]
- 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 \( t'_s = P' \)
- Need to use a random q
- We will show that the probability of a false positive is small → randomized algorithm

False positives

- Consider any \( t \neq p \). We know that both numbers are in the range \( \{0...2^{m-1}\} \)
- How many primes \( q \) are there such that \( t \mod q = p \mod q = (t \mod p) \mod q \)?
- Such prime has to divide \( x = (t \mod p) \leq 2^m \)
- To understand how likely it is to stumble upon false positives, We need to understand the prime factorization of \( x \)
- Think about the representations of \( x \) as a product of primes:
  \[ x = p_1^{e_1}p_2^{e_2}...p_k^{e_k} \]
  \( p_i \) prime, \( e_i \geq 1 \)
  \( \text{exemplary: 12 = 2^2 \cdot 3^1, 900 = 2^2 \cdot 3^2 \cdot 5^2, 1024 = 2^{10}} \)
- Since \( 2 \leq p_i \), we have \( 2 \leq x \leq 2^m - k \leq m \)
- There are \( \leq m \) primes dividing \( x = (t \mod p) \)
**Algorithm**

- Let $[\pi]$ be a set of $2nm$ primes, each having $O(\log n)$ bits (we will replace this stage with a simpler one soon).
- Choose $q$ uniformly at random from $[\pi]$.
- Compute $t_q \ell \ldots$ (the hashed values) and $p'$.
- **Claim:** The probability of any false positive is at most $(n-m)/2n \leq 1/2$.
- **Proof:**
  - $t_0 \ell \ldots$ is the product of $\leq m$ primes.
  - $t_1 \ell \ldots$ is the product of $\leq m$ primes.
  - $t_2 \ell \ldots$ is the product of $\leq m$ primes.
  - $\ldots$
  - $t_m \ell \ldots$ is the product of $\leq m$ primes.
- Even if all these primes are distinct, there are only $\leq mn$ “bad” primes, that might yield false positive. If we pick $q$ at random from $\pi$, we have a probability of $\geq 1/2$ that we don’t pick a bad prime.

**“Details”**

- How do we know that such $[\pi]$ exists?
- How do we choose a random prime from $[\pi]$ in $O(n)$ time?

---

**Prime density**

- Primes are “dense”. I.e., if PRIMES(N) is the set of primes smaller than $N$, then asymptotically:
  \[
  \frac{|\text{PRIMES}(N)|}{N} \sim \frac{1}{\log N} 
  \]
- If $N$ large enough, then:
  \[
  |\text{PRIMES}(N)| \geq \frac{N}{(2\log N)}
  \]

**Prime density continued**

- If we set $N = 9mn \log n$, and $N$ large enough, then:
  \[
  |\text{PRIMES}(N)| \geq \frac{N}{(2\log N)} \geq 2mn
  \]
- All elements of PRIMES(N) are $\log N = O(\log n)$ bits long.

---

**Prime selection**

- Still need to find a random element of PRIMES(N).
- Solution:
  - Choose a random element from \{1 \ldots N\} (or pick at random from a database of primes).
  - Check if it is prime.
  - If not, repeat.

**Prime selection analysis**

- A random element $q$ from \{1 \ldots N\} is prime with probability $\sim 1/\log N$.
- We can check if $q$ is prime in time polynomial in $\log N$ (trust me☺).
- Therefore, we can generate random prime $q$ in $o(n)$ time.
- The rest of the algorithm takes $O(n)$ time.
Bloom Filter

As usual, need a data structure for a set \( S \). The data needs to support:

- \( \text{insert}(k, S) \)
- \( \text{find}(k, S) \)

In contrast to almost all other DS, here find only return a yes/no answer, but does not provide auxiliary data.

\[ \text{Delete}(S) \] Could also work, but is less efficient.

- For example, if \( S \) is the list of courses offered this semester, and we perform \( \text{find("cs545"}, S) \), the answer is that it is in \( S \) (offered), we to find any information about the course, queries in other databases are needed.
- It is possible that \( \text{find("cs545"}, S) \) will return "yes" even though cs545 is NOT in \( S \). (false positive). But the probability is small.
- If \( \text{find("cs545"}, S) \) returns NO, then "cs545" is not in \( S \) (with probability 100%).
- Bloom filter is used (da) for filtering the search. Only if Bloom returns 'yes' a more expensive search is performed.
- Very useful for distributed databases.

\[ \text{m} = 2n \text{ or } 3n \text{ for } \alpha \text{ = 0.05} \]

\[ \text{fix} \text{ k hash functions } h_1(x), h_2(x), \ldots, h_r(x) \text{ all returns values between } 1 \text{..} m. \]

Performing \( \text{find}(s) \)
\[ \text{return } A[h_i(x)] = 1 \text{ AND } \text{A}[h_2(x)] = 1 \text{ AND } \ldots \text{ AND } \text{A}[h_r(x)] = 1 \]

- Assume an insert(\( x \)) is performed. Fix \( \text{ k hash functions } h_1(x), h_2(x), \ldots, h_r(x) \) did not set this bit is \( (1 - 1/m)^k \approx e^{-k/m} \).
- If \( n \) words are inserted into \( S \), the prob that \( T[r] \) is still 0 is \( e^{-nm/k} \).
- The prob that \( T[i] = 1 \) is \( 1 - e^{-nm/k} \).
- Now, assume \( x \notin S \). The prob that we perform find(\( x \)) and the answer is erroneously YES is \( e = 1 - e^{-nm/k} \).
- This is the probability of a false positive.
- Next we need to optimize for \( n \text{ and } k \).

\[ \text{m} = 2n \text{ or } 3n \text{ for } e = 0.05 \text{. That is, } 2 \text{ bits per (comparing to maintaining the whole key - a string (multiple chars)) or at least, a word (64 bits)} \]

\[ k = 4 \text{ or } 5 \]

**CS 545**

**Bloom Filter**

Alon Efrat

Estimating number of items in a set

- Almost all operations could be performed in parallel, using bitwise arithmetic.
- Assume \( T \) and \( T' \) are the Bloom Filters arrays of sets \( S \) and \( S' \).

To estimate the number of keys in \( S \), we could use \( n(S) = m \ln \frac{1}{1 - |S|/m} \).

Here \( |S| \) is the number of bits which are 'on'. \( k \) is the number of hash functions and \( m \) is the length of the array.

- Assume \( T \) and \( T' \) are the Bloom Filters arrays of sets \( S \) and \( S' \). Both uses the same set of hash functions. How could we compute the array for the set \( S \cup S' \)?

\[ n(S \cup S') = \frac{m}{k} \ln \frac{1}{1 - \frac{|A \cup B|}{m}} \]

Answer: Just take the bitwise OR of the tables.

\[ n(S \cap S') = \frac{m}{k} \ln \frac{1}{1 - \frac{|A \cap B|}{m}} \]

From knowing the number of '1's in the filter, we could estimate \( n(S) \), the number of items in \( S \).

- What about intersection?
- \( n(A \cap B) = n(A) + n(B) - n(A \cup B) \)

Need a guesstimate about \( n = |S| \) the number of keys to be inserted. Also given \( e > 0 \), the error rate. Typically \( e = 0.05 \).

- We determine \( m \) - table size. In contrast to hash table, each cell in the table contains only one bit (not a key).
- Generate \( A[1..m] \). If \( S \) is empty, then all bits are set to 0.
- Determine a value \( k \) (later).

For \( i=1,k \text{, set } A[h(x)] = 1 \text{ independent of its previous value.} \)

Performing \( \text{find}(x) \)
\[ \text{return } A[h_1(x)] = 1 \text{ AND } A[h_2(x)] = 1 \text{ AND } \ldots \text{ AND } A[h_r(x)] = 1 \]

**Example** (credit: Wikipedia)

\[ S = \{x, y, z\}, w \notin S \]

Creating Filter For union and intersections of sets of false positive

Just perform bitwise AND or bitwise OR

**LSH - locality preserving hashing. Finding nearest nbrs in \( \mathbb{R}^d, d > 30 \)**

- A good example is the Locality Sensitive hashing (LSH).
- Big idea: Pick a random direction \( \vec{v} \).
- Create the line \( \{v \cdot \vec{x} = 1\} \), through the origin, and in the direction of \( \vec{v} \).
- Project (orthogonally) all the data points on the line \( \{v \cdot \vec{x} = 1\} \).
- Project means "orthogonal projection":
  - \( \text{Proj}(x) = (x, \vec{v}) \cdot \vec{v} \) (easier if \( \vec{v} = (1, 0, \ldots, 0) \)).
- Have some critical threshold \( r > 0 \) in mind. Points are close to each other if their distance is \( \leq r \) (approximately)
- Points that are close to each other in \( \mathbb{R}^d \), will be close to each other after the projection.
- Points that are far away (much larger than \( r \)) in \( \mathbb{R}^d \), their projection could be close or far from each other.
- Picking a random direction, the probability that their projection is far away from each other is much larger than probability that their projection is close to each other.

Due to false positive, we could use more than one line.

How could we find efficiently if two buckets share a point?
Grids

- Let \( S = \{p_1, p_2, \ldots, p_n\} \) data points.
- Let \( \varepsilon > 0 \) be a given value.
- Consider the grid \( G \) of edge-length \( \varepsilon \) (the corner fixed arbitrarily).
- We can use a hash table to support the following operations:
  - \( \text{Insert}(p) \) // add data point \( p \) - time \( O(1) \)
  - \( \text{Report}(q) \) - report all data points in the cell containing \( q \). Time \( O(1 + k) \) where \( k \) is their number.
- Note - the grid has \( \infty \) cells, but we store only the non-empty ones.

\[
\varepsilon = \begin{cases} 
(0.0, 0.0) & \text{(2x, 0)} \\
(2.0, 0.0) & \text{(6x, 0)}
\end{cases}
\]

**Note:** The two stars have same keys

This is a great DS to maintain, say who is in which block of the city.

**Note:** the grid has \( \infty \) cells, but we store only the non-empty ones.