Bloom Filter

As usual, need a data structure for a set S. The data needs to support
• insert(k, S)
• Find(k, S). In contrast to almost all other DS, here find only return a yes/no answer, but does not provide auxiliary data.
• Delete(S). Could also work, but is less efficient.

• For example, if S is the list of courses offered this semester, and we perform 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 find("cs545", S) will return "yes" even though cs545 is NOT in S. (false positive). But the probability is small.
• 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.

\[ h(x) \]

\[ \approx \]

\[ \text{Probability of false positive} \]
\[
\text{Performing find(x)} \\
\text{return \( A[h_1(x)] = 1 \) AND \( A[h_2(x)] \neq 1 \) AND \ldots AND \( A[h_k(x)] \neq 1 \)}
\]

• Assume an insert(x) is performed. Fix 1 \( \leq r \leq m \). Consider \( T[r] \). The probability that \( h_1(x) \neq r \) is \( 1/m \). This is the probability that this bit is not set to 1 by \( h_1(x) \).
• The prob that none of the \( k \) hash functions \( h_1(x), h_2(x), \ldots h_k(x) \) 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 \( e^{-nk/m} \).
• The probability that \( T[r] \) is 1 \( \approx e^{nk/m} \).
• Now, assume \( x \notin S \). The prob that we perform find(x) and the answer is erroneously YES is \( \varepsilon = (1 - e^{-nk/m})^k \). This is the probability of a false positive.
• Next we need to optimize for m and k.
• \( m=2n \) or \( 3n \) for \( \varepsilon = 0.05 \). That is, 2 bits per (comparing to maintaining the whole key - a string (multiple chars) or at least, a word (64 bits))
• \( k=4 \) or 5

Example (credit wikipedia)
\[
S = \{ x, y, z \}, \ w \notin S
\]

String Matching

• Need a guessestimate about \( n = |S| \) the number of keys to be inserted.
  Also given \( \varepsilon > 0 \), the error rate. Typically \( \varepsilon = 0.05 \).
• 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).
• Fix \( k \) hash functions \( h_1(x), h_2(x), \ldots h_k(x) \), all returns values between 1..m.

Performing insert(x)
For \( i = 1..k \)
Set \( A[h_i(x)] = 1 \) // independent of its previous value.

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

• Needs to work 20X faster or perform in parallel, using online arithmetic.
• To estimate the number of keys in \( A \), we could use
  • 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 \( S \cup S' \)?
  \[
  m(A \cup B) = -m \ln \left( 1 - \frac{|A|}{m} \right)
  \]
  • Answer: Just take the bitwise OR of the tables.
  \[
  n(A \cup B) = m \ln \left( 1 - \frac{|A \cup B|}{m} \right)
  \]
  • From knowing the number of 1’s in the filter, we could estimate n(S), the number of items in S
  • What about intersection ?
  • \( n(A \cap B) \approx n(A) + n(B) - n(A \cup B) \)

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 \( T[1…18] = "to be or not to be" \)
• \( P \) and \( T \) containing symbols from alphabet \( \Sigma \)
• Goal: find all "shifts" \( 1 \leq s \leq n-m \) such that \( T[s + 1..s + m] = P \)
• Example:
  - \( \Sigma = \{a, b, \ldots, z\} \)
  - \( T[1..18] = "to be or not to be" \)
  - \( P[1..2] = "be" \)
  - Shifts: 3, 16

Thanks to
Prof. Piotr Indyk
Simple Algorithm

\[
\begin{align*}
\text{for } s & \leftarrow 0 \text{ to } n-m \\
\text{Match} & \leftarrow 1 \\
\text{for } j & \leftarrow 1 \text{ to } m \\
\text{if } T[s+j] \neq P[j] \text{ then} \\
\text{Match} & \leftarrow 0 \\
\text{exit loop} \\
\text{if } \text{Match} = 1 \text{ then output } s
\end{align*}
\]

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

Karp-Rabin Algorithm

\[
\begin{align*}
\text{for } s & \leftarrow 0 \text{ to } n-m \\
\text{Match} & \leftarrow 1 \\
\text{for } j & \leftarrow 1 \text{ to } m \\
\text{if } T[s+j] \neq P[j] \text{ then} \\
\text{Match} & \leftarrow 0 \\
\text{exit loop} \\
\text{if } \text{Match} = 1 \text{ then output } s
\end{align*}
\]

Implementation

\[
\begin{align*}
0 & \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 [0 \ 1 \ 1 \ 1] \\
\text{for } s & \leftarrow 0 \\
\text{Think about each } T_t & = T[s+1\ldots s+m] \text{ as a number in binary representation, i.e.,} \\
t & = T[s+1]2^0 + T[s+2]2^1 + \ldots + T[s+m]2^{m-1} \\
\text{Find a fast way of computing } t_{s+1} \text{ given } t_s \\
\text{Output all } s \text{ such that } t_s \text{ is equal to the number } p \text{ represented by } P
\end{align*}
\]

The great formula

- How to transform
  \[
t_s = T[s+1]2^0 + T[s+2]2^1 + \ldots + T[s+m]2^{m-1}
\]
  into
  \[
t_{s+1} = T[s+2]2^0 + T[s+3]2^1 + \ldots + T[s+m+1]2^{m-1}
\]
- 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:
  \[
t_{s+1} = (t_s - T[s+1]2^0) + T[s+m+1]2^{m-1}
\]

Algorithm

- Can compute \( t_{s+1} \) from \( t_s \) using 3 arithmetic operations
- Therefore, we can compute all \( t_0, t_1, \ldots, t_{n-m} \) 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!
- 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 $t'_s = h(T[s+1:s+m]) = \sum_{i=s+1}^{s+m} T[i] \mod q$.

where $q$ is an “appropriate” prime number.

**Algorithm**

- Let $\prod$ 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 $\prod$.
- Compute $t'_0, t'_1, \ldots$ (the hashed values) and $p'$.

**False positives**

- Consider any $t \neq p$. We know that both numbers are in the range $\{0 \ldots 2m-1\}$.
- How many primes $q$ are there such that $t \mod q = p \mod q = 0$ ?
- Such prime has to divide $x = (t-p)$.
- 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: $x = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$.
- This has at most $m$ factors.

**“Details”**

- How do we know that such $\prod$ exists?
- How do we choose a random prime from $\prod$ in $O(n)$ time?
**Prime density**

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

**Prime density continued**

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

**Prime selection**

- Still need to find a random element of \( \text{PRIMES}(N) \)
- 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