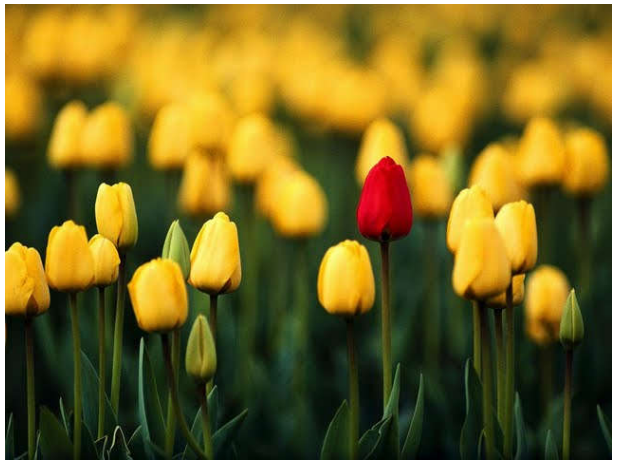
Randomized algorithms 2 Bloom filters, string matching

CS240 Spring 2020

Rui Fan

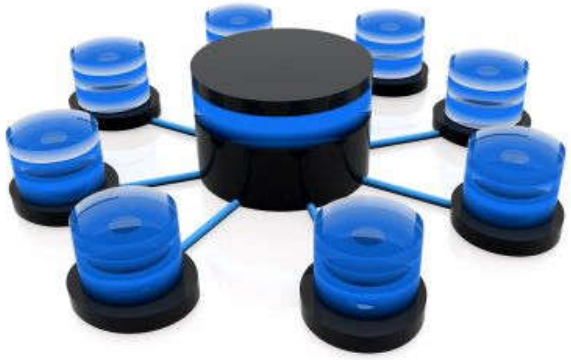
Approximate sets

- A Bloom filter is a data structure that can implement a set.
  - It only keeps track of which keys are present, not any values associated to keys.
  - It supports insert and find operations.
  - It doesn't support delete operations.
- Bloom filters use less memory than hash tables or other ways of implementing sets.
- However, Bloom filters are approximate.
  - It can produce false positives: it says an element is present even though it's not.
    - We can bound the probability of false positives.
  - But it doesn't produce false negatives: if it says an element isn't present, then it's not.



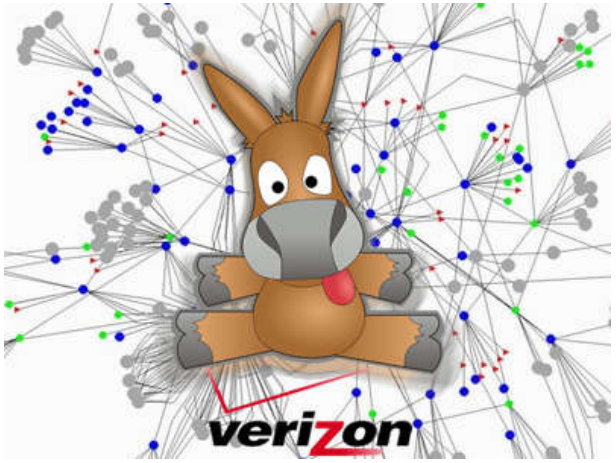
Bloom filter applications

- Suppose we have a big database, and querying it to check if an item is present is expensive.
- We store the set of items in the database using a Bloom filter.
  - This tells us whether an item is in database or not.
- If filter says an item's not present, it's definitely not in the database.
  - So no need to do an expensive query.
- If filter says an item is present, then either item is present, or there's false positive.
  - When we query the database, there's a small probability we waste time querying for a nonexistent item.
- Overall we save time by checking Bloom filter first before querying database.



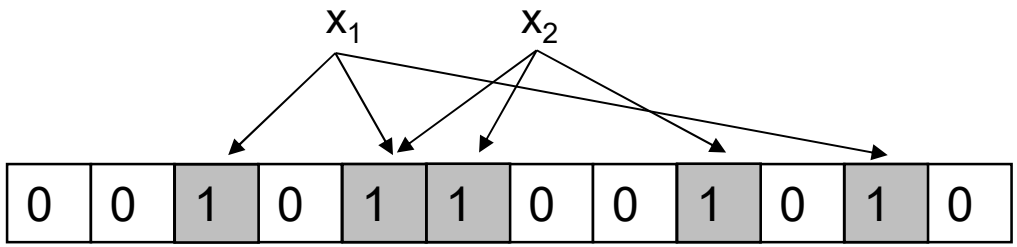
Bloom filter applications

- Consider a P2P network, where each node stores some files.
- If you want to get a file, you need to know which nodes have it.
- Keeping a list of all items stored at each node is too expensive.
- Instead, for every other node, keep a Bloom filter of its files.
- If filter says no for a node, it definitely doesn't have the file.
- If filter says yes, then either node has the file, or there's false positive and we make a useless request.
- Overall we save space, and also won't waste much communication because we rarely make useless requests.



Bloom filters

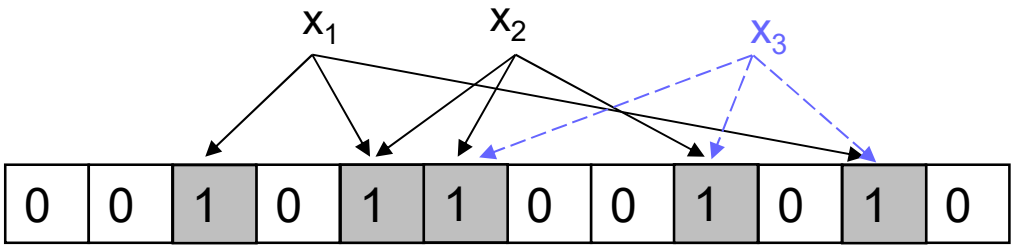
- A Bloom filter consists of
  - An array A of size m, initially all 0's.
  - k independent hash functions $h_1,...,h_k$, each mapping from keys to {1,...,m}.
- To store key x
  - Set $A[h_1(x)]$, $A[h_2(x)]$, ..., $A[h_k(x)]$ all to 1.
  - Some locations can get set to 1 multiple times; that's fine.
- To check if key x is in the set
  - Read array locations $A[h_1(x)]$, $A[h_2(x)]$, ..., $A[h_k(x)]$.
  - If all the values are 1, output "x is in set".
  - Otherwise output "x is not in set".



A Bloom filter with k=3 hash functions storing 2 items.

Correctness

- Let's look at the correctness of the search function.
- If search for x returns no, then at least one of $A[h_1(x)],...,A[h_k(x)]$ equals 0.
  - So x cannot be in the set, because if x had been inserted into the set, then we would have $A[h_1(x)]=...=A[h_k(x)]=1$.
  - So there are no false negatives.
- If search for x returns yes, then $A[h_1(x)]=...=A[h_k(x)]=1$.
  - So either x was inserted into the set.
  - Or we inserted some keys that hashed to the same k locations as x.
    - So it looks as if x was inserted, even though it wasn't.
    - This is a false positive. We'll bound the probability this happens.



False positive probability 1

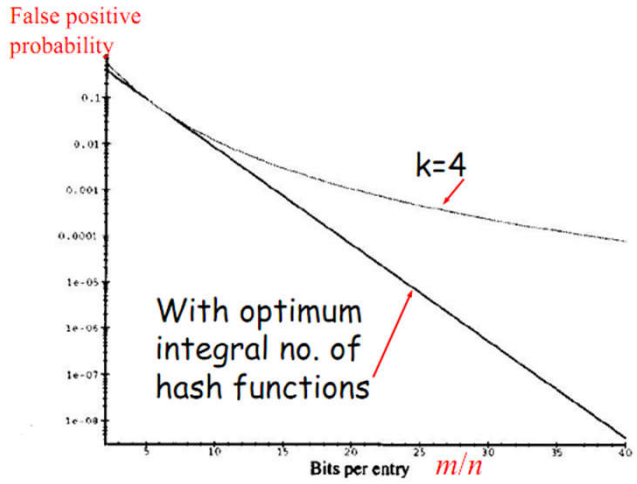
- False positive probability depends on k (number of hash functions), m (size of table) and n (number of keys inserted).
- Assume hash functions hash keys to random locations.
- When inserting one key, we set k random locations to 1.
- Fix any position i. Probability i is set to 1 by a hash function is 1/m, so probability i stays 0 is 1-1/m.
  - After k hashes, probability i still 0 is $(1 - 1/m)^k$.
  - To insert n items, we used nk hashes. So probability i still 0 after all these is $p = (1 - 1/m)^{nk}$.
- We now use an approximation $\left(1 - \frac{1}{m}\right)^{nk} \approx e^{-\frac{nk}{m}}$.

False positive probability 2

- So probability any position i is 1 after n keys inserted is $1 - p \approx 1 - e^{-\frac{nk}{m}}$.
- Since there are m positions in the array, assume there are (1-p)m positions that are 1.
  - This isn't quite correct. The actual number of 1's in the array is a random variable, whose expectation is (1-p)m.
  - However, we can make the argument rigorous by showing that the actual number of 1's is $(1-p)m + \sqrt{m \log m}$ with high probability.
- We only get a false positive if when we check k random locations, they're all 1.
  - Probability is $f = (1-p)^k \approx \left(1 - e^{-\frac{nk}{m}}\right)^k$.

False positive probability 3

- Notice the false prob. $\left(1 - e^{-\frac{nk}{m}}\right)^k$ is a function of k, the number of hash functions we use.
- We find k to minimize the false positive prob. by differentiating f wrt k and solving.
- The optimum k is $\frac{m \ln(2)}{n}$, which leads to $f = \left(\frac{1}{2}\right)^k \approx 0.6185^{\frac{m}{n}}$.
  - Notice that m/n is the average number of bits per item. So error rate decreases exponentially in space usage.

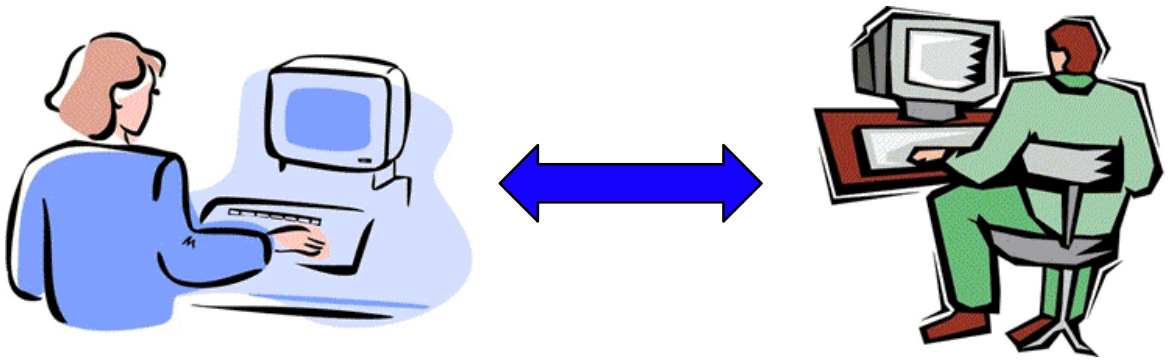


Improvements

- Right now Bloom filters can't handle deletes.
  - Say keys k_1, k_2 hash to two overlapping sets of locations. If you delete k_1 by setting some of its locations to 0, you could also delete k_2.
- Deletes can be done by storing a count of how many keys hashed to that location, and inc / dec the counts when inserting or deleting.
  - But this uses more memory.
  - Also, what if the counts overflow?
- Neat trick Given Bloom filters for sets S_1, S_2, we can create Bloom filter for $S_1 \cap S_2$ and $S_1 \cup S_2$ just by bitwise ANDing or ORing S_1 and S_2's filters.
  - ORing gives the exact Bloom filter for $S_1 \cup S_2$, while ANDing gives approximately the Bloom filter for $S_1 \cap S_2$.

String equality and fingerprinting

- Alice and Bob both have copies of a database.
- They want to keep the database consistent, so they want to check if their copies are the same.
  - If you think of the databases as strings, they want to check if their strings are equal.
- But transferring the entire database is expensive.
- Instead, they calculate a small value called a fingerprint of their databases.
  - If the fingerprints are the different, then their databases are definitely different.
  - If the fingerprints are the same, then the databases are probably the same; but there's a small probability they're actually different.
- Transferring the fingerprint is much cheaper than the database.



Fingerprinting

- Let Alice and Bob's databases be the bit sequences $(a_1,...,a_n)$ and $(b_1,...,b_n)$.
- View these as n-bit integers $a=\sum^{n}_{i=1} a_i*2^{i-1}$ and $b=\sum^{n}_{i=1} b_i*2^{i-1}$.
- The fingerprint $F(a) = a \bmod p$, for a specially chosen prime number p.
  - Alice transfers F(a) to Bob, and Bob compares it to his fingerprint F(b)=b mod p.
  - Since F(a) < p, transferring the fingerprint only takes O(log p) bits, instead of n.

Correctness

- No false positives (positive means "$a \neq b$").
  - If F(a)≠F(b), then a≠b.
- False negatives are possible.
  - If F(a)=F(b), then a mod p = b mod p.
  - So either a=b, or a≠b but p divides (a-b).
- We can't avoid false negatives. But we can minimize the probability it occurs.
- Pick a random p.
  - If a≠b, then probably p doesn't divide (a-b), so probably F(a)≠F(b) and we'll detect a and b are different.
  - Bigger p decreases false negative probability.
  - But we don't want to make p too big, since we have to transfer O(log p) bits.

Correctness

- To analyze the false negative probability, we use two facts from number theory.
- Lemma Any number t has at most $\log_2(t)$ distinct prime divisors.
- Proof Each divisor is ≥ 2, and their product is $\leq t$. If there were more than $\log_2(t)$ divisors, their product would be $> 2^{\log_2(t)} = t$, contradiction.
- Recall $a=\sum^{n}_{i=1} a_i*2^{i-1}$ and $b=\sum^{n}_{i=1} b_i*2^{i-1}$.
- So $a-b<2^n$, and so a-b has at most n distinct prime divisors.

Correctness

- Prime Number Theorem Given any number t, the number of primes smaller than t is ~ t / ln(t).
- The PNT allows us to efficiently generate a random prime.
  - Picking a number less than t at random, it has a 1/ln(t) probability of being prime.
  - We can check if a number is prime using the Rabin-Miller primality test.
    - If number is prime, it always passes the test.
    - If number is composite, there's small probability it's declared a prime.
    - Run the test few more times to exponentially decrease false positive probability.
  - So with high probability, we can tell if a number is prime.

Correctness

- Let $t = n^2 \ln(n)$. The number of primes less than t is $\approx \frac{t}{\ln(t)} = \frac{n^2 \ln(n)}{2 \ln(n) + \ln \ln(n)} = O(n^2)$.
- Pick a random prime p less than t.
- We get a false negative if a≠b but p divides (a-b).
  - We saw earlier that a-b has < n prime divisors, and p must be one of these.
  - But p is randomly chosen from $O(n^2)$ primes less than t.
  - So false negative probability $\leq n/O(n^2) = O(1/n)$.
- We transfer $\log(p) \leq \log(t) = O(\log n)$ bits.
- Transferring O(log n) bits gets O(1/n) probability of error. If we want perfect accuracy, we need to transfer the entire database, O(n) bits.

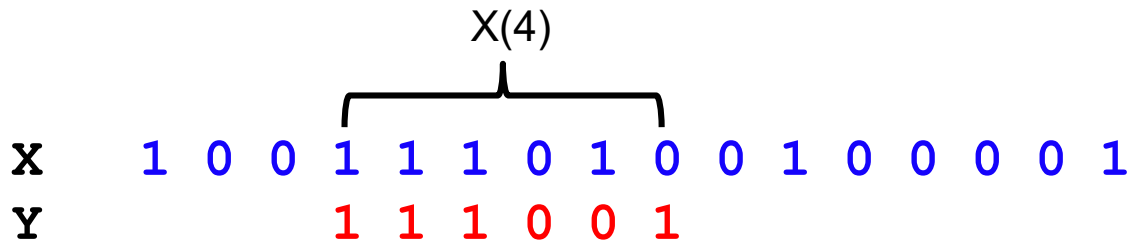
String matching

- In the previous problem, we checked whether two strings are equal.
- We now want to see if one string X contains the other Y. I.e. we want to match Y to a part of X.
- Let X and Y be binary strings with length n and m, resp., where $n \geq m$.
- We'll look at a simple randomized O(n+m) time string matching algorithm Rabin-Karp.

```
X   1 0 0 1 1 1 0 1 0 0 1 0 0 0 0 1
Y         1 1 1 0 0 1
```

String matching

- Let $X=x_1x_2...x_n$, and define $X(j)=x_jx_{j+1}...x_{j+m-1}$.
  - X(j) has the same length as Y.
  - It represents a potential match for Y starting at the j'th bit of X.
- Y matches X if there exists $1 \le j \le n-m+1$ s.t. X(j)=Y.
- We just saw a way to test string equality using fingerprints.
- We first give a O(n+m) time Monte Carlo string matching algorithm.
  - There's small probability it says Y matches X even when it doesn't.
- Then we convert it to an expected O(n+m) time Las Vegas algorithm, which always gives correct answer.



Monte Carlo string matching

- Choose a random prime p.
  - We'll determine how big p should be later.
- For every $1 \le j \le n-m+1$.
  - View X(j) and Y as m-bit numbers.
  - Compute $F(X(j))=X(j) \bmod p$ and $F(Y)=Y \bmod p$.
  - If $F(X(j)) \neq F(Y)$, go to next j.
  - If $F(X(j))=F(Y)$, output "Y matches X at bit j".
- Output "Y doesn't match X".

Correctness

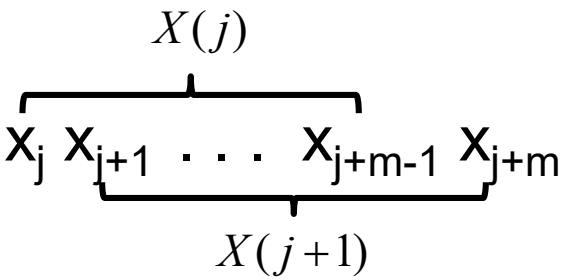
- Consider X(j) and Y, for some j.
- If $F(X(j)) \neq F(Y)$, then $X(j) \neq Y$ and there's no match at X's j'th bit.
  - So no false negatives.
- If $F(X(j)) = F(Y)$, then either $X(j) = Y$, or $X(j) \neq Y$ but p divides X(j)-Y.
  - So there could be false positives.
  - Let's bound this probability.
- X(j)-Y is an m-bit number, so by the lemma, it has $\leq$ m distinct prime factors.
- Let $t = n^2 m \log(n^2 m)$, and let p be a random prime $\leq$ t.
  - There are ~ $n^2 m$ primes $\leq$ t, by the PNT.

Correctness

- We only get a false positive if p is one of the $\leq m$ factors of X(j)-Y.
- So we get a false match between X(j) and Y with probability $\leq m/n^2m = 1/n^2$.
- Now, we falsely match X to Y if there is any $1 \leq j \leq n-m+1$ s.t. $X(j) \neq Y$ but $F(X(j))=F(Y)$.
  - For each j, this event has probability $\leq 1/n^2$.
  - So the probability this happens at any of the n-m+1 j's is $\leq (n-m+1)/n^2 = O(1/n)$.
  - This is called the union bound.
    - If $Pr[E_i]=p_i$ for $1 \leq i \leq k$, then $Pr[E_1 \cup ... \cup E_k] \leq p_1 + ... + p_n$.
- Putting it together, if Y matches X somewhere, then this algorithm finds the match. If Y doesn't match X, this algorithm says Y matches X with probability O(1/n).

Running time

- We said this algorithm runs in O(n+m) time. But computing $F(X(j)) = \left(\sum_{i=j}^{j+m-1} x_i 2^{i-1}\right) \bmod p$ takes O(m) time, and we have to do this for n-m+1 j's, which makes the running time O(mn)!
- However, there's a trick to computing F(X(j+1)) fast once you've computed F(X(j)).
  - X(j+1) and X(j) differ only in bits x_j and x_{j+m}.
  - So $F(X(j+1))=(2*(F(X(j)) - 2^{m-1}x_j) + x_{j+m}) \bmod p$.
  - So computing F(X(j+1)) given F(X(j)) takes O(1) time.
- So computing F(X(1)),...,F(X(n-m+1)) takes O(n) time.
- Computing F(Y) takes O(m) time.
- So altogether the algorithm takes O(n+m) time.



A Las Vegas algorithm

- We now modify the previous algorithm so it always produces the right answer.

- Run the Monte Carlo algorithm.
- If it never produces a match, output “Y doesn’t match X”.
- If it ever outputs a match, say between X(j) and Y, then check X(j)=Y in O(m) time.
- If X(j)=Y, output “Y matches X at bit j”.
- If X(j)≠Y, start over and run the brute force O(mn) time to match Y to X.

Correctness

- Since the Monte Carlo algorithm only produced false positives and this algorithm checks them using brute force, it always produces the right answer.
- The expected running time is Pr[no false positives]*(running time if no false positives)+Pr[there is false positive]*(running time when there's false positive).
  - The first term is $\leq (1-1/n)(n+m)$.
  - The second term is $\leq 1/n*O(mn)=O(m)$.
  - So the total expected running time is $O(n+m)$.