Probability in Computing

LECTURE 6: BINS AND BALLS,
APPLICATIONS: HASHING & BLOOM FILTERS

Agenda

- Review: the problem of bins and balls
- Poisson distribution
- Hashing
- Bloom Filters

Balls into Bins

- We have m balls that are thrown into n bins, with the location of each ball chosen independently and uniformly at random from n possibilities.
- What does the distribution of the balls into the bins look like
  - "Birthday paradox" question: is there a bin with at least 2 balls
  - How many of the bins are empty?
  - How many balls are in the fullest bin?

Answers to these questions give solutions to many problems in the design and analysis of algorithms

The maximum load

- When n balls are thrown independently and uniformly at random into n bins, the probability that the maximum load is more than 3 ln*n*/lnln*n* is at most 1/*n* for *n* sufficiently large.
  - By Union bound, Pr [bin 1 receives ≥ M balls] ≤ $\binom{n}{M}\left(\frac{1}{n}\right)^M$.
  - Note that:

$$\binom{n}{M}\left(\frac{1}{n}\right)^M \le \frac{1}{M!} \le \left(\frac{e}{M}\right)^M.$$

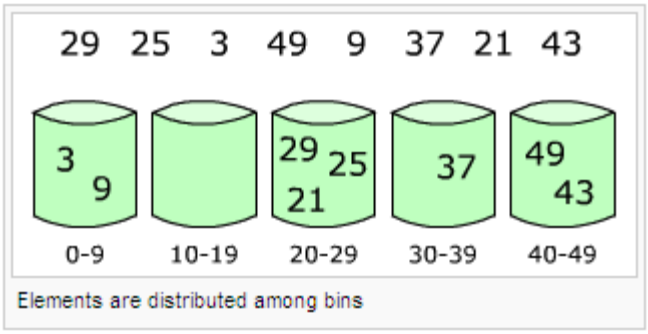
  - Now, using Union bound again, Pr [any ball receives ≥ M balls] is at most

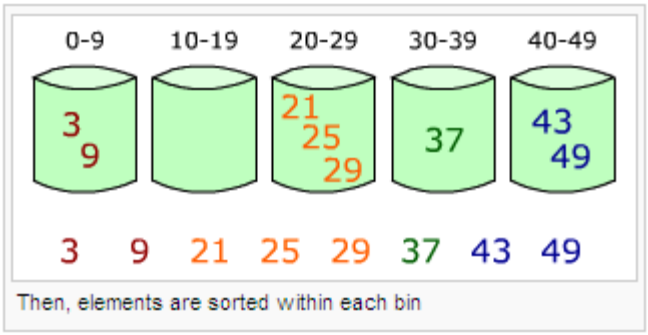
$$n\left(\frac{e}{M}\right)^M \le n\left(\frac{e \ln\ln n}{3 \ln n}\right)^{3 \ln n / \ln\ln n}$$

which is ≤ 1 / n

Application: Bucket Sort

- A sorting algorithm that breaks the $\Omega(n\log n)$ lower bound under certain input assumption
- Bucket sort works as follows:
  - Set up an array of initially empty "buckets."
  - Scatter: Go over the original array, putting each object in its bucket.
  - Sort each non-empty bucket.
  - Gather: Visit the buckets in order and put all elements back into the original array.





- A set of $n = 2^m$ integers, randomly chosen from $[0,2^k), k \geq m$, can be sorted in expected time $O(n)$
  - Why: will analyze later!

The Poisson Distribution

- Consider m balls, n bins
  - Pr [a given bin is empty] = $\left(1-\frac{1}{n}\right)^m \approx e^{-m/n}$;
  - Let X_j is a indicator r.v. that is 1 if bin j empty, 0 otherwise
  - Let X be a r.v. that represents # empty bins

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i] = n\left(1-\frac{1}{n}\right)^m \approx ne^{-m/n}$$

  - Generalizing this argument, Pr [a given bin has r balls] =

$$\binom{m}{r}\left(\frac{1}{n}\right)^r\left(1-\frac{1}{n}\right)^{m-r} = \frac{1}{r!}\frac{m(m-1)\cdots(m-r+1)}{n^r}\left(1-\frac{1}{n}\right)^{m-r}$$

  - Approximately, $p_r \approx \dfrac{e^{-m/n}(m/n)^r}{r!}$
  - So: **Definition 5.1:** *A discrete* Poisson random variable *X with parameter* μ *is given by the following probability distribution on* $j = 0, 1, 2, \ldots$:

$$\Pr(X = j) = \frac{e^{-\mu}\mu^j}{j!}.$$

Limit of the Binomial Distribution

We have shown that, when throwing m balls randomly into b bins, the probability p_r that a bin has r balls is approximately the Poisson distribution with mean m/b. In general, the Poisson distribution is the limit distribution of the binomial distribution with parameters n and p, when n is large and p is small. More precisely, we have the following limit result.

Theorem 5.5: *Let X_n be a binomial random variable with parameters n and p, where p is a function of n and $\lim_{n\to\infty} np = \lambda$ is a constant that is independent of n. Then, for any fixed k,*

$$\lim_{n\to\infty} \Pr(X_n = k) = \frac{e^{-\lambda}\lambda^k}{k!}.$$

This theorem directly applies to the balls-and-bins scenario. Consider the situation where there are m balls and b bins, where m is a function of b and $\lim_{n\to\infty} m/b = \lambda$. Let X_n be the number of balls in a specific bin. Then X_n is a binomial random variable with parameters m and $1/b$. Theorem 5.5 thus applies and says that

$$\lim_{n\to\infty} \Pr(X_n = r) = \frac{e^{-m/n}(m/n)^r}{r!},$$

Application: Hashing

- The balls-and-bins model is good to model hashing
- Example: password checker
  - Goal: prevent people from choosing common, easily cracked passwords
  - Keeping a dictionary of unacceptable passwords and check newly created password against this dictionary.
- Initial approach: Sorting this dictionary and do binary search on it when checking a password
  - Would require $\Omega(\log m)$ time for m words in the dictionary
- New approach: chain hashing
  - Place the words into bins and search appropriate bin for the word
  - The worlds in a bin: implemented as a linked list
  - The placement of words into bins is done by using a hash function

Chain hashing

- Hash table
  - A hash function f: U → [0,n-1] is a way of placing items from the universe U into n bins
  - Here, U consists of all possible password strings
  - The collection of bins called hash table
  - Chain hashing: items that fall into the same bin are chained together in a linked list
- Using a hash table turns the dictionary problem into a balls-and-bins problem
  - m words, hashing range [0..n-1] ➔ m balls, n bins
  - Making assumption: we can design perfect hash functions that map words into bins uniformly random
    - A given word could be mapped into any bin with the same probability

Search time in chain hashing

- To search for an item
  - First hash it to find the corresponding bin then find it in the bin: sequential search through the linked list
  - The expected # balls in a bin is about m/n ➔ expected time for the search is $\Theta(m/n)$
  - If we chose m=n then a search takes expectedly constant time
- Worst case
  - maximum # balls in a bin: $\Theta(\ln n / \ln\ln n)$ if choose m=n
  - Another disadvantage: wasting a lot of space in empty bins

Hashing: bit strings

- In chain hashing, n balls n bins, we waste a lot of empty bins ➔ should have m/n >>1
- Hashing using sort fingerprints will help
  - Suppose: passwords are 8-char, i.e. 64 bits
  - We use a hash function that maps each pwd into a 32-bit string, i.e. a fingerprint
  - We store the dictionary of fingerprints of the unacceptable passwords
  - When checking a password, compute its fingerprint then check it against the dictionary: if found then reject this password
- But it is possible that our password checker may not give the correct answer!

False positives

- This hashing scheme gives a false positive when it rejects a good password
  - The fingerprint of this password accidentally matches that of an unacceptable password
  - For our password checker application this over-conservative approach is, however, acceptable if the probability of making a false positive is not too high

False positive probability

- How many bits should we use to create fingerprints?
  - We want reasonably small probability of a false positive match
  - Prob [the fingerprint of a given good pwd ≠ any given unacceptable fingerprint] = $1- {}^{1}/_{2^b}$; here b # bits
  - Thus for m unacceptable pwd, prob [false positive occurs on a given good pwd] = $1- (1- {}^{1}/_{2^b})^m \geq 1- e^{-m/2^b}$
  - Easy to see that: to make this prob less than a given small constant, we need b= $\Omega(\log n)$
    - If use b=2log*n* bits ➔ Prob [a false positive]= $1-(1-{}^{1}/_{m^2})^m < {}^{1}/_{m}$
    - Dictionary of 2^{16} words using 32-bit fingerprint ➔ false prob ${}^{1}/_{65,536}$

An approximate set membership problem

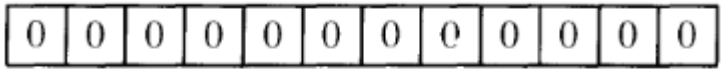
- Suppose we have a set $S = \{s_1, s_2, s_3, \ldots, s_m\}$ of m elements from a large universe set U. We would like to represent the elements of S in such a way so that
  - We can quickly answer the queries of form "Is x is an element of S?"
  - We want the representation take as little space as possible
- For saving space we can accept occasional mistakes in form of false positives
  - E.g. in our password checker application

Bloom filters

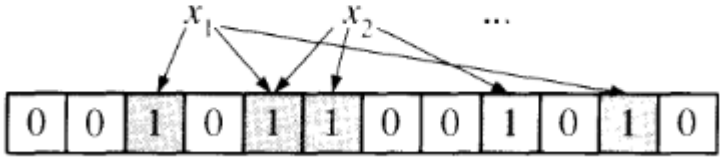
- A Bloom filter: a data structure for this approximate set membership problem
  - By generalizing these mentioned hashing ideas to achieve more interesting trade-off between required space and the false positive probability
  - Consists of an array of *n* bits, A[0] to A[n-1], initially set to 0
  - Uses *k* independent hash functions h_1, h_2, ..., h_k with range {0,...n-1}; all these are uniformly random
  - Represent an element $s \in S$ by setting $A[h_i(s)]$ to 1, i=1,..k

- Checking: For any value x, to see if $x \in S$ simply check if $A[h_i(x)] = 1$ for all $i=1,..k$
  - If not, clearly x is not a member of S
  - If right, we assume that x is in S but we could be wrong! ➔ false positive

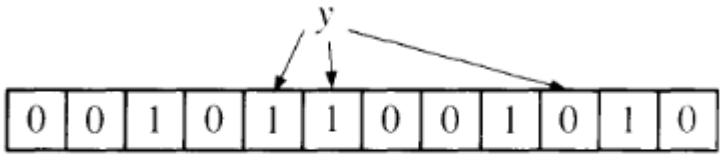
Start with an array of 0s.



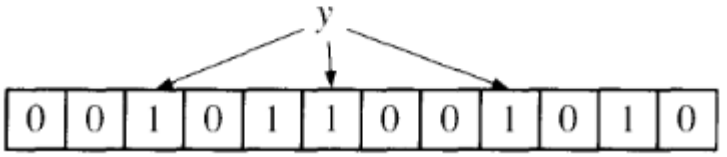
Each element of S is hashed k times; each hash gives an array location to set to 1.



To check if y is in S, check the k hash locations. If a 0 appears, y is not in S.



If only 1s appear, conclude that y is in S. This may yield false positives.



False positive probability

- The probability of a false positive for an element not in the set
  - After all m elements of S are hashed into Bloom filter, Prob[a give bit =0] = $(1-{}^{1}/_{n})^{km} \approx e^{-km/n}$. Let $p = e^{-km/n}$.
  - Prob [a false positive] = $(1-(1-{}^{1}/_{n})^{km})^{k} \approx (1-e^{-km/n})^{k} = (1-p)^{k}$. Let $f = (1-p)^{k}$.
  - Given m, n what is the optimum k to minimize f?
    - Note that a higher k gives us more chance to find a 0-bit for an element not in S, but using fewer h-functions increases the fraction of 0-bit in the array.
  - Optimal $k = \ln 2 . {}^{n}/_{m}$ which reaches minimum $f = ½^{k} \approx (0.6185)^{n/m}$
  - Thus Bloom filters allow a small probability of a false positive while keep the number of storage bit per item a constant
    - Note in previous consideration of fingerprints we need $\Omega(\log m)$ bits per items

Bloom filters: applications

- Discovering DoS attack attempt
  - Computing the difference between SYN and FIN packets
    - Matching between SYN and FIN packets by 4-tuples of addresses (source and destination ports)
- Many, many other applications

Application of hashing: breaking symmetry

- Suppose that n users want a unique resource (processes demand CPU time) how can we decide a permutation quickly and fairly?
  - Hashing the User ID into 2^b bits then sort the resulting numbers
    - That is, smallest hash will go first
    - How to avoid two users being hashed to the same value?
- If b large enough we can avoid such collisions as in birthday paradox analysis
  - Fix an user. Prob [another user has the same hash] = $1- (1-{}^{1}/_{2b})^{n-1} \leq {}^{(n-1)}/_{2b}$
  - By union bound, prob [two users have the same hash] = ${}^{(n-1)n}/_{2b}$
    - Thus, choosing b =3log*n* guarantees success with probability$\geq 1-{}^{1}/n$
  - Leader election