CS125	Lecture 18	Fall 2014

18.1 The Birthday Paradox

How many people do there need to be in a room before with probability greater than $1/2$ some two of them have the same birthday? (Assume birthdays are distributed uniformly at random.)

Surprisingly, only 23. This is easily determined as follows: the probability the first two people have different birthdays is $(1-1/365)$. The probability that the third person in the room then has a birthday different from the first two, given the first two people have different birthdays, is $(1-2/365)$, and so on. So the probability that all of the first k people have different birthdays is the product of these terms, or

$$(1-\frac{1}{365})\cdot(1-\frac{2}{365})\cdot(1-\frac{3}{365})\ldots\cdot(1-\frac{k-1}{365}).$$

Determining the right value of k is now a simple exercise.

18.2 Balls into Bins

Mathematically, the birthday paradox is an example of a more general mathematical question, often formulated in terms of balls and bins. Some number of balls n are thrown into some number of bins m. What does the distribution of balls and bins look like?

The birthday paradox is focused on the first time a ball lands in a bin with another ball. One might also ask how many of the bins are empty, how many balls are in the most full bin, and other sorts of questions.

Let us consider the question of how many bins are empty. Look at the first bin. For it to be empty, it has to be missed by all n balls. Since each ball hits the first bin with probability $1/m$, the probability the first bin remains empty is

$$(1-\frac{1}{m})^n \approx \mathrm{e}^{-n/m}.$$

Since the same argument holds for all bins, on average a fraction $\mathrm{e}^{-n/m}$ of the bins will remain empty.

Exercise: How many bins have 1 ball? 2?

18.3 Hash functions

A *hash function* is a deterministic mapping from one set into another that appears random. For example, mapping people into their birthdays can be thought of as a hash function.

In general, a hash function is a mapping $f:\{0,\ldots,n-1\}\rightarrow\{0,\ldots,m-1\}$. Generally $n>>m$; for example, the number of people in the world in much bigger than the number of possible birthdays. There is a great deal of theory behind designing hash functions that "appear random." We will not go into that theory here, and instead assume that the hash functions we have available are in fact completely random. In other words, we assume that for each i ($0 \leq i \leq n-1$), the probability that $f(i) = j$ is $1/m$ (for ($0 \leq j \leq m-1$). Notice that this does mean that every time we look at $f(i)$, we get a different random answer! The value of $f(i)$ is fixed for all time; it is just equally likely to take on any value in the range.

While such completely random hash functions are unavailable in practice, they generally provide a good rough idea of how hashing schemes perform.

(An aside: in reality, birthdays are not completely random either. Seasonal distributions skew the calculation. How might this affect the birthday paradox?)

18.4 Applications: A Password-checker

We now consider a hashing application. Suppose you are administering a computer system, and you would like to make sure that nobody uses a common password. This protects against hackers, who can often determine if someone is using a common password (such as a name, or a common dictionary word) by gaining access to the encrypted password file and using an exhaustive search. When the user attempts to change their password, you would like to check their password against a dictionary of common passwords as quickly as possible.

One way to do this would be to use a standard search technique, such as binary search, on the string. This approach has two negative features. First, one must store the entire dictionary, which takes memory. Second, on a large dictionary, this approach might be slow. Instead we present a quick and space-efficient scheme based on hashing. The data structure we consider is commonly called a Bloom filter, after the originator. The downside of the Bloom filter is that it may sometimes give a wrong answer.

So far, we have really focused on giving solutions to problems that give the right answer. And that probably seems reasonable. But it's so limiting. Why are we so focused on getting the right answer?

In algorithms, we commonly think of the tradeoff between space and time. Space and time are resources that need to be traded off. But there are other "resources" that can be traded off, including correctness. We might be willing to take an answer that is incorrect – ideally, in some reasonably well understood way – to save on time and memory. (You do this in real life all the time.) Similarly, we might prefer simpler solutions to more complex solutions, even if the more complex solutions are somewhat better, in a number of situations.

18.4.1 Using a Single Hash Function

Choose a table size m. Create a table consisting of m bits, initially all set to 0. Use a hash function on each of the n words in the dictionary, where the range of the hash function is $[0,m)$. If the word hashes to value k, set the kth bit of the table to 1.

When a user attempts to change the password, hash the user's desired password and check the appropriate entry in the table. If there is a 1 there, reject the password; it could be a common one. Otherwise, accept it. A common password from the dictionary is always rejected. Assuming other strings are hashed to a random location, the probability of rejecting a password that should be accepted is $1 - \mathrm{e}^{-n/m}$.

18.4.2 Using a Bloom Filter

It would seem one would need to choose m to be fairly large in order to make the probability of rejecting a potentially good password small. Space can be used more efficiently by making multiple tables, using a different hash function to set the bits for each table. This approach of using multiple hash functions leads to a data structure that is known as a *Bloom filter*, named after its inventor, Burton Bloom. (For lots more on Bloom filters and variations, see the survey at `http://www.eecs.harvard.edu/~michaelm/NEWWORK/postscripts/BloomFilterSurvey.pdf`.)

To check a proposed password now requires more time, since the locations in our bit array given by several hash functions must be checked. However, as soon as a single 0 entry is found, the password can be accepted. (Why is this the case?) The probability of rejecting a password that should be accepted when using h tables, each of size m/h (for a total size of m), is then

$$\left(1-\mathrm{e}^{-n/(m/h)}\right)^h = \left(1-\mathrm{e}^{-nh/m}\right)^h.$$

Notice that the Bloom filter sometimes returns the wrong answer – we may reject a proposed password, even though it is not a common password. This sort of error is probably acceptable, as long as it doesn't happen so frequently as to bother users. Fortunately this error is one-sided; a common password is never accepted. One must set the parameters m and h appropriately to trade off this error probability against space and time requirements.

For example, consider a dictionary of 100,000 common passwords, each of which is on average 7 characters long. Uncompressed this would be 700,000 bytes. Compression might reduce it substantially, to around 300,000 bytes. Of course, then one has the problems of searching efficiently on a compressed list.

Instead, one could keep a 100,000 byte Bloom filter, consisting of 5 tables of 160,000 bits. The probability of rejecting a reasonable password is just over 2%. The cost for checking a password is at most 5 hashes and 5 lookups into the table.