CSE 203A
Advanced Algorithms

Winter 2015 - Prof. Ramamohan Paturi

Lecture Notes – Thursday, February 19th, 2015

Balls into Bins

Hashing / Poisson Distribution

Sergio Miguel Martin
A53070688

Hashing

Consider the application of a password checker, which prevents people from using common, easily cracked passwords by keeping a dictionary of unacceptable passwords. When a user tries to set up a password, the application would check if the requested password is a part of the unacceptable set. The elements of this problem are:

$S = \{s_1, s_2, ..., s_m\}$ *m*-sized list of unacceptable passwords

x - A candidate password to test

U - Set of all possible passwords such that:

$$x \in U \quad S \subseteq U$$

Supposing 8-bit characters, and at most 64 characters per password:

$$|U| = 2^{64 \times 8}$$

One first approach to implement such password checker is to keep S sorted in alphabetic and perform a binary search for every new password *x* requested. Disregarding the set-up cost of sorting the dictionary of passwords, traversing the list and comparing the input password with the forbidden passwords in the list would require $\Theta(\log m)$ time.

Although this approach achieves a logarithmic time, this may not be fast enough for some applications. Furthermore, since password checking may occur frequently, it would be useful to store the list in some fast access memory (cache) for quick access. This could not be feasible if the list of *m* passwords is large enough. Therefore, we need to consider a random approach to reduce both computational time and storage size.

To achieve such goals, we can use a *hashing* approach. We define a hash function *h* that maps every possible password to a single hash value:

$$h: U \rightarrow [0, n-1]$$

where:

n - Amount of hash values used

Note that, since *n* is expected to be much smaller than |U|, the hashing function *h* is not a 1:1 mapping from every possible password to a unique hash value, but rather a surjective function. That is, many different possible passwords can be mapped to the same hash value.

Then, we define A to be a data structure such that:

$A[i]$ - State of the i^{th} hash value, where:

$$A[i] = 1 \quad \text{if and only if} \quad \exists\ s_j \in S\ /\ h(s_j) = i$$

$$A[i] = 0 \quad \text{otherwise}$$

In other words, any hash value *i* will contain a 1 if any of the forbidden passwords has been mapped to *i*.

To evaluate if a new password *x* is acceptable, we need to apply the hashing function to it and look at the state of its corresponding hash value. The outcome could be either:

$$A[h(x)] = 0$$

In this case, no forbidden password has been mapped to the same hash value. Therefore, the password is acceptable.

$$A[h(x)] = 1$$

In this case, at least one forbidden password has been mapped to this value. This arises two possibilities: the password *x* could either match with one forbidden password mapped to the same value, or it could be just a false positive. That is, it does not match to any of the passwords.

To resolve whether *x* is a forbidden password in the latter case, we could keep a linked list of forbidden passwords for every hash value that contains at least one of them, and compare each of its items to determine if *x* matches any of them (this approach is called *chain hashing*). The question is now, how long could this take? Specifically, how large can any of this liked lists be?

To analyze this we define:

L_i Size of the list for the hash value *i*, for $0 \leq i \leq n-1$

Now let's consider a worst case scenario where a deterministic hash function is used. This function could map all the *m* forbidden passwords in S to a single hash value *i*, obtaining:

$$L_i = m, \quad L_j = 0 \quad \text{for all} \quad j \neq i$$

If the requested password is also mapped to *i*, then we will have to compare it with all *m* forbidden passwords. Even if using binary search, this will not offer any improvement. Ideally, we rather use a hashing function that distributes passwords evenly among the hash values such that:

$$L_i = \frac{m}{n} \quad \text{for all} \quad 0 \leq i \leq n-1$$

This would be a best case scenario. Note that choosing n = m, we would have:

$$L_i = 1$$

Requiring no comparisons, and providing an immediate result.

In order to try to approach that ideal mapping we could define a hash function *h* that satisfies the following requirements:

1. Tries to map passwords evenly among the hash values (best case).
2. It is simple to calculate. That is, it does not demand an excessive amount of computation to provide a $U \rightarrow [0, n-1]$ mapping.
3. *h* should should be selected independently from S. The same hashing function should be able to provide a valid mapping provided any set $S \subseteq U$. Furthermore, for each element in U, a hash value should be independently (pair-wise independence is sufficient) and uniformly assigned.

Note that these set of requirements could not be met by any deterministic algorithm. Therefore, we know that h should use a random approach to construct the hash map.

We now claim that such hash function *h* will always provide a mapping where the list size for any hash value *i* is at most:

$$L_i \leq \Theta\left(\frac{lnm}{lnlnn}\right) \text{ with a high probability.}$$

As a consequence, choosing n = m, will provide:

$$L_i \leq \Theta\left(\frac{lnm}{lnlnm}\right)$$

Which is a better bound than using just binary search.

The hashing approach analysis is one instance of the more general model called balls into bins. We will use that model to prove the claim above.

Balls into Bins Model

Consider the following scenario. We have *m* balls that are thrown into *n* bins, with the location of each ball chosen independently and uniformly at random from the *n* possibilities. Note that this problem is analogous of our hashing problem when using a random hash function.

- Balls represent the forbidden words

- Bins represent hash values

We define a simple random variable to indicate whether a specific ball *j* was thrown into a bin *i*.

$$L_{ij} = \begin{cases} 1 & \textit{iff } j \textit{ was thrown into } i \\ 0 & \textit{otherwise} \end{cases}$$

We then define L_i as the amount of balls that a specific bin *i* received by the sum (note that this is analogous of our definition of L for hashing):

$$L_i = \sum_{j=1}^{m} L_{ij}$$

Since the probability of a ball falling into a specific bin is uniform, we know that:

$$E[L_{ij}] = \frac{1}{n}$$

It follows:

$$E[L_i] = \sum_{j=1}^{m} \frac{1}{n} = \frac{m}{n}$$

It is expected that every bin should receive roughly the average amount of balls per bin. However, this information is not enough to determine its actual time complexity and spatial distribution.

Time Complexity

We would like to determine now, with a high probability, what is the *maximum* amount of balls that any bin will contain by the end of the process. This represents the worst case scenario in our hashing problem, in which a

binary search will take longer to determine if there is a matching forbidden password.

To determine this amount we can start by asking: what is the probability that one bin receives exactly certain number *M* of balls? Since each ball destination is determined independently, this comes from a binomial distribution:

$$P(L_i = M) = \binom{m}{M}\left(\frac{1}{n}\right)^M\left(1-\frac{1}{n}\right)^{m-M}$$

We can relax the previous statement by requiring the probability that a bin get *at least* M balls:

$$P(L_i \geq M) \leq \binom{m}{M}\left(\frac{1}{n}\right)^M \quad (*)$$

This probability is bounded by the right hand side since we do not take into account the outcome of the rest of the *m-M* balls.

To determine the maximum amount of balls a bin can receive, we would like to know what is the probability that there exists *at least* one bin receiving at least *M* balls:

$$P(max\, L_i \geq M) = P(\exists i / L_i \geq M)$$

To determine this probability, we apply union bound to the probability of just one bin (*), to all *n* bins:

$$P(\exists i / L_i \geq M) \underset{\text{union bound}}{=} n\,P(L_i \geq M)$$

Follows:

$$P(\exists i / L_i \geq M) \leq \binom{m}{M}\left(\frac{1}{n}\right)^M$$

We need to decompose the right hand side to gain more information about $P(\exists i / L_i \geq M)$:

$$\binom{m}{M}\left(\frac{1}{n}\right)^M = n\frac{m!}{(m-M)!\,M!}\left(\frac{1}{n}\right)^M \leq n\frac{m^M}{M!}\left(\frac{1}{n}\right)^M$$

$$n\frac{m!}{M!}\left(\frac{1}{n}\right)^M = \frac{n}{M!}\left(\frac{m}{n}\right)^M$$

We can simplify the factorial of the right hand side of the above equation by using Stirling's approximation (disregarding the lower order part under the square root):

$$n\frac{m!}{M!}\left(\frac{1}{n}\right)^M \approx \frac{n}{\left(\frac{M}{e}\right)}\left(\frac{m}{n}\right)^M = n\left(\frac{m\,e}{M\,n}\right)^M$$

Stirling's Approximation:

$$M! \approx \sqrt{2\pi M}\left(\frac{M}{e}\right)^M$$

More information about Stirling's approximation can be obtained in the book "Asymptopia" by Joel Spencer

Now let's evaluate the case in which we use the same amount of bins as balls. That is, m = n:

$$P(\exists i / L_i \geq M) \leq n\left(\frac{e}{M}\right)^M \quad (**)$$

Now, if we chose any M such that:

$$M \geq \frac{3\ln n}{3\ln\ln n}$$

We can bound the probability in (**) by:

$$\begin{aligned} P(\exists i / L_i \geq M) &\leq n\left(\frac{e\ln\ln n}{3\ln n}\right)^{\frac{3\ln n}{\ln\ln n}} \\ &\leq n\left(\frac{\ln\ln n}{\ln n}\right)^{\frac{3\ln n}{\ln\ln n}} \\ &= e^{\ln n}\left(e^{\ln\ln\ln n - \ln\ln n}\right)^{\frac{3\ln n}{\ln\ln n}} \\ &= e^{-2\ln n + 3\frac{(\ln n)(lnlnln n)}{\ln\ln n}} \\ &\leq \frac{1}{n} \quad \text{For } n \text{ sufficiently large, this probability tends to zero.} \end{aligned}$$

Therefore, the with high probability, there will no bin with more than $\Theta\left(\frac{lnm}{lnlnm}\right)$ balls

Spatial Distribution

Determining the spatial distribution of this random approach can be relevant to its applications. For example, having an excessive amount of empty hash list can waste space in a small cache memory. The analogy with the balls into bins problem is when bins end up empty after throwing all balls randomly. What is, then, the probability that a bin will end up empty? and how many empty bins will there be after the process?

For the analysis of empty bins we can define, for every bin i:

$$T_i = \begin{Bmatrix} 1 & iff\ L_i = 0 \\ 0 & otherwise \end{Bmatrix} \text{ with } P(T_i = 1) = P(L_i = 0)$$

With: $P(L_i = 0) = \left(1 - \frac{1}{n}\right)^m \approx e^{\frac{-m}{n}}$

In this case, we cannot apply a Chernoff bound for the expectancy of T because the events T_i are not independent: if one bin is empty, it is less likely for the rest to be empty as well. However, we can still use the union bound, since it does not require independence, obtaining:

$$E[T] \approx ne^{\frac{-m}{n}}$$

In the case of the chain hashing approach, since we use $n=m$, several of the bins may end up empty, potentially leading to a wasted space problem (one of the problems we originally needed to address). In order to solve this, section 5.5.2 of the textbook introduces a *fingerprint* approach to hashing that addresses the space problem.

If we wanted to bound probability for T, we will not be able to use a Bernoulli distribution since, as explained previously, the events T_i are not independent. However, we can still use a Poisson distribution to obtain the answer.

Poisson Distribution (an introduction)

We have seen that the probability that a specific bin ends up empty after throwing all the balls randomly can be approximated to:

$$P(L_i=0) \approx e^{\frac{-m}{n}}$$

And the expected amount of empty bins will be:

$$E[T] \approx ne^{\frac{-m}{n}}$$

We can now generalize the argument to find the expected amount of bins with *r* balls for any constant *r*. The probability that a given bin has *r* balls is:

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

When *m* and *n* are large compared to *r*, the second factor on the right hand side is approximately $\left(\frac{m}{r}\right)^r$, and the third factor is approximately $e^{\frac{-m}{n}}$. Hence the probability of that a given bin has *r* balls is approximately:

$$P(L_i=r) \approx \frac{e^{\frac{-m}{n}}\left(\frac{m}{r}\right)^r}{r!}$$

This calculation leads into the definition of a discrete Poisson random variable:

A *discrete* Poisson random variable *X* with parameter μ is given by the following probability distribution on j = 0, 1, 2...:

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

We now verify that all probabilities in this distribution sum up to 1:

$$\begin{aligned}\sum_{j=0}^{\infty} P[X=j] &= \sum_{j=0}^{\infty}\frac{e^{-\mu}\mu^j}{j!}\\ &= e^{-\mu}\sum_{j=0}^{\infty}\frac{\mu^j}{j!}\\ &= \sum_{j=0}^{\infty}\frac{j!}{\mu^j}\sum_{j=0}^{\infty}\frac{\mu^j}{j!}\\ &= 1\end{aligned}$$

Then, we show that the expectation of this random variable is μ:

$$
\begin{aligned}
E[X] &= \sum_{j=0}^{\infty} j\,P(X=j) \\
&= \sum_{j=1}^{\infty} j\frac{e^{-\mu}\mu^{j}}{j!} \\
&= \mu\sum_{j=1}^{\infty} j\frac{e^{-\mu}\mu^{j-1}}{j-1!} \\
&= \mu\sum_{j=0}^{\infty} j\frac{e^{-\mu}\mu^{j}}{j!} \\
&= \mu
\end{aligned}
$$

This is expected from our conclusions for the balls into bins problem with *m* balls and *n* bins. The expectancy of balls in a bin is approximately Poisson with $\mu = \frac{m}{n}$ which is exactly the average number of balls per bin.

The importance of being able to describe the probability of a bin receiving a certain number of balls as a Poisson variable with mean *m/n* is that this allows us to treat each bin as an independent variable. Section 5.4 of the textbook shows how can this fact be used to obtain a more precise bound ($e\sqrt{(m)}$) for the probability of the event when *m* balls are thrown into *r* balls.