Three Killer Applications: Hashing, Coupon Collecting and Load Balancing

In this note, we will see that the simple balls-and-bins probability space can be used to model a surprising range of phenomena. Recall that in this process we distribute m balls into n bins, where each ball is independently placed in a uniformly random bin. We can ask questions such as:

- How large can we choose m while ensuring that with probability at least $1/2$, no two balls land in the same bin?
- If $m = n$ (same number of balls as bins), what is the maximum number of balls that are likely to land in the same bin?

As we shall see, these two simple questions provide key insights into two important engineering applications: hashing and load balancing. These in turn arise in many contexts, such as the design of efficient data structures and resource allocation in networks, where demands can be assumed to be random (so-called "stochastic multiplexing.")

Our answers to these questions are based largely on the application of a simple technique called the union bound, discussed earlier. Given any two events A and B, it states that $\mathbb{P}[A \cup B] \leq \mathbb{P}[A] + \mathbb{P}[B]$. This observation plus some "back-of-the-envelope" calculations will get us a long way.

1 Hashing

One of the basic issues in hashing is the tradeoff between the size of the hash table and the number of collisions. Here is a simple question that quantifies the tradeoff:

- Suppose a hash function distributes keys evenly over a table of size n. How many keys can we hash before the probability of a collision exceeds (say) $\frac{1}{2}$?

Recall that a hash table is a data structure that supports the storage of a set of keys drawn from a large universe U (say, the names of all 325m people in the US). The set of keys to be stored changes over time, and so the data structure allows keys to be added and deleted quickly. Given a key, it also reports whether or not the key belongs to the currently stored set. The crucial question is: How large must the hash table be to allow these operations (addition, deletion and membership) to be implemented quickly?

Here is how the hashing works. The hash function h maps U to a table T of modest size (much smaller than U). To ADD a key x to our set, we evaluate $h(x)$ (i.e., apply the hash function to the key) and store x at the location $h(x)$ in the table T. All keys in our set that are mapped to the same table location are stored in a simple linked list. The operations DELETE and MEMBER are implemented in similar fashion, by evaluating $h(x)$ and searching the linked list at $h(x)$. Note that the efficiency of a hash function depends on having only few collisions — i.e., keys that map to the same location. This is because the search time for DELETE and MEMBER operations is proportional to the length of the corresponding linked list.

Of course, we could be unlucky and choose keys such that our hash function maps many of them to the same location in the table. But the whole idea behind hashing is that we select our hash function carefully, so that it scrambles up the input key and seems to map it to a random location in the table, making it unlikely that most of the keys we select are mapped to the same location. To quantitatively understand this phenomenon, we will model our hash function as a *random* function—i.e., one that maps each key to a uniformly random location in the table, independently of where all other keys are mapped. The question we will answer is the following: what is the largest number m of keys we can store before the probability of a collision reaches $\frac{1}{2}$? Note that there is nothing special about $\frac{1}{2}$. One can ask, and answer, the same question with different values of collision probability, and the largest number of keys m will change accordingly.

1.1 Balls and Bins

Let us begin by seeing how this problem can be put into the balls-and-bins framework. The balls will be the m keys to be stored, and the bins will be the n locations in the hash table T. Since the hash function maps each key to a random location in the table T, we can see each key (ball) as choosing a hash table location (bin) uniformly from T, independently of all other keys. Thus the probability space corresponding to this hashing experiment is exactly the same as the balls-and-bins space.

We are interested in the event A that there is no collision, or equivalently, that all m balls land in different bins. Clearly $\mathbb{P}[A]$ will decrease as m increases (with n fixed). Our goal is to find the largest value of m such that $\mathbb{P}[A] \geq 1-\varepsilon$, where $\varepsilon \in (0,1)$ is a specified tolerance level of collision probability. [Note: Strictly speaking, we are looking at different sample spaces here, one for each value of m. So it would be more correct to write $\mathbb{P}_m$ rather than just $\mathbb{P}$, to make clear which sample space we are talking about. However, we will omit this detail.]

1.2 Using the Union Bound

Let us see how to use the union bound to achieve this goal. We will fix the value of m and try to compute $\mathbb{P}[A]$. There are exactly $\binom{m}{2} = \frac{m(m-1)}{2}$ possible pairs among our m keys. Imagine these are numbered from 1 to $\binom{m}{2}$ (it does not matter how). Let C_i denote the event that pair i has a collision (i.e., both keys in the pair are hashed to the same location). Then the event $\overline{A}$ that *some* collision occurs can be written $\overline{A} = \bigcup_{i=1}^{\binom{m}{2}} C_i$. What is $\mathbb{P}[C_i]$? We claim it is just $\frac{1}{n}$ for every i; this is the probability that two particular balls land in the same bin when there are n bins. (Check that you understand this!)

So, using the union bound from earlier, we have

$$\mathbb{P}[\overline{A}] = \mathbb{P}\left[\bigcup_{i=1}^{\binom{m}{2}} C_i\right] \leq \sum_{i=1}^{\binom{m}{2}} \mathbb{P}[C_i] = \binom{m}{2} \times \frac{1}{n} = \frac{m(m-1)}{2n} \approx \frac{m^2}{2n}.$$

This means that the probability of having a collision is less than ε if $\frac{m^2}{2n} \leq \varepsilon$; that is, if $m \leq \sqrt{2\varepsilon n}$. (For $\varepsilon = \frac{1}{2}$, this just says $m \leq \sqrt{n}$.) Thus, if we wish to suffer no collisions with high probability, the size of the hash table must be about the *square* of the cardinality of the set we are trying to store. (In the later section on load balancing, we will see what happens to the number of collisions if we decrease the size of the hash table and make it similar to the size of the set we are trying to store.)

As detailed in Section 1.4, it turns out that we can derive a slightly more accurate bound for m using techniques that are less crude than the union bound. However, although that alternate bound is a little better, both bounds are the same in terms of their dependence on n (both are of the form $m = O(\sqrt{n})$).

1.3 The Birthday Paradox Revisited

Recall the Birthday Paradox from an earlier lecture: how many people need to be in a room in order to ensure that two of them have the same birthday, with probability at least $\frac{1}{2}$? This is exactly the hashing problem considered in the previous section, with $n = 365$. (Why? Actually there is a difference of 1 in the answers, since the hashing problem wants to avoid collisions with probability at least $\frac{1}{2}$ while the birthday problem wants to get at least one collision with probability at least $\frac{1}{2}$). We noted in the earlier lecture, by an exact calculation, that the answer to the birthday problem is 23. This is in line with our approximate calculation in the previous section, since 23 is of a similar order to $\sqrt{365} \approx 19$.

1.4 A More Accurate Bound

In this section, we derive a more accurate bound on m for the collision problem, which will show that the square-root dependence $m = O(\sqrt{n})$ is not only sufficient but also necessary.

Main Idea

Let's fix the value of m and try to compute $\mathbb{P}[A]$ exactly, rather than approximating it using the union bound as we did above. (This exact calculation is what we did for the birthday problem in the earlier lecture.) Since our probability space is uniform (each outcome has probability $\frac{1}{n^m}$), it's enough just to count the number of outcomes in A. In how many ways can we arrange m balls in n bins so that no bin contains more than one ball? Well, this is just the number of ways of choosing m things out of n *without* replacement, which as we saw in an earlier note is

$$n \times (n-1) \times (n-2) \times \cdots \times (n-m+2) \times (n-m+1).$$

This formula is valid as long as $m \leq n$: if $m > n$ then clearly the answer is zero. From now on, we will assume that $m \leq n$.

Now we can calculate the probability of no collision:

$$\begin{aligned}
\mathbb{P}[A] &= \frac{n(n-1)(n-2)\ldots(n-m+1)}{n^m} \\
&= \frac{n}{n} \times \frac{n-1}{n} \times \frac{n-2}{n} \times \cdots \times \frac{n-m+1}{n} \\
&= \left(1 - \frac{1}{n}\right) \times \left(1 - \frac{2}{n}\right) \times \cdots \times \left(1 - \frac{m-1}{n}\right). \qquad (1)
\end{aligned}$$

Before going on, let's pause to observe that we could compute $\mathbb{P}[A]$ in a different way, as follows. View the probability space as a sequence of choices, one for each ball. For $1 \leq i \leq m$, let A_i be the event that the ith ball lands in a different bin from balls $1, 2, \ldots, i-1$. Then

$$\begin{aligned}
\mathbb{P}[A] = \mathbb{P}\left[\bigcap_{i=1}^{m} A_i\right] &= \mathbb{P}[A_1] \times \mathbb{P}[A_2 \mid A_1] \times \mathbb{P}[A_3 \mid A_1 \cap A_2] \times \cdots \times \mathbb{P}\left[A_m \mid \bigcap_{i=1}^{m-1} A_i\right] \\
&= 1 \times \frac{n-1}{n} \times \frac{n-2}{n} \times \cdots \times \frac{n-m+1}{n} \\
&= \left(1 - \frac{1}{n}\right) \times \left(1 - \frac{2}{n}\right) \times \cdots \times \left(1 - \frac{m-1}{n}\right).
\end{aligned}$$

Fortunately, we get the same answer as before! [You should make sure you see how we obtained the conditional probabilities in the second line above. For example, $\mathbb{P}[A_3 \mid A_1 \cap A_2]$ is the probability that the

third ball lands in a different bin from the first two balls, *given that* those two balls also landed in different bins. This means that the third ball has $n-2$ possible bin choices out of a total of n.]

Essentially, we are now done with our problem: Equation (1) gives an *exact* formula for the probability of no collision when m keys are hashed. All we need to do now is plug values $m = 1,2,3,\ldots$ into (1) until we find that $\mathbb{P}[A]$ drops below $1-\varepsilon$. The corresponding value of m (minus 1) is what we want. (Again, this is exactly what we did for the birthday problem in an earlier lecture.)

We can actually make this bound much more useful by turning it around, as we will do below. We will derive an equation which tells us the value of m at which $\mathbb{P}[A]$ drops below $1-\varepsilon$.

Further Simplification

The bound we gave above (for the largest number m of keys we can store before the probability of a collision reaches $\frac{1}{2}$) is not really satisfactory: it would be much more useful to have a formula that gives the "critical" value of m directly, rather than having to compute $\mathbb{P}[A]$ for $m = 1,2,3,\ldots$. Note that we would have to do this computation separately for each different value of n we are interested in: i.e., whenever we change the size of our hash table.

So what remains is to "turn Equation (1) around", so that it tells us the value of m at which $\mathbb{P}[A]$ drops below $\frac{1}{2}$. To do this, let's take logs: this is a good thing to do because it turns the product into a sum, which is easier to handle. We get

$$\ln(\mathbb{P}[A]) = \ln\left(1-\frac{1}{n}\right) + \ln\left(1-\frac{2}{n}\right) + \cdots + \ln\left(1-\frac{m-1}{n}\right), \tag{2}$$

where "ln" denotes natural (base e) logarithm. Now we can make use of a standard approximation for logarithms: namely, if x is small then $\ln(1-x) \approx -x$. This comes from the Taylor series expansion

$$\ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \cdots.$$

So by replacing $\ln(1-x)$ by $-x$ we are making an error of at most $(\frac{x^2}{2} + \frac{x^3}{3} + \cdots)$, which is at most $2x^2$ when $x \leq \frac{1}{2}$. In other words, we have

$$-x \geq \ln(1-x) \geq -x - 2x^2.$$

And if x is small then the error term $2x^2$ will be much smaller than the main term $-x$. Rather than carry around the error term $2x^2$ everywhere, in what follows we will just write $\ln(1-x) \approx -x$, secure in the knowledge that we could make this approximation precise if necessary.

Now let's plug this approximation into Equation (2):

$$\begin{aligned}
\ln(\mathbb{P}[A]) &\approx -\frac{1}{n} - \frac{2}{n} - \frac{3}{n} - \cdots - \frac{m-1}{n} \\
&= -\frac{1}{n}\sum_{i=1}^{m-1} i \\
&= -\frac{m(m-1)}{2n} \\
&\approx -\frac{m^2}{2n}.
\end{aligned} \tag{3}$$

Note that we have used the approximation for $\ln(1-x)$ with $x = \frac{1}{n}, \frac{2}{n}, \frac{3}{n}, \ldots, \frac{m-1}{n}$. So our approximation should be good provided all these are small, i.e., provided n is fairly big and m is quite a bit smaller than n (which it will be, since we'll only need to take m up to about $\sqrt{n}$). Once we are done, we will see that the approximation is actually pretty good even for modest sizes of n.

Now we can undo the logs in (3) to get our expression for $\mathbb{P}[A]$:

$$\mathbb{P}[A] \approx e^{-\frac{m^2}{2n}}.$$

The final step is to figure out for what value of m this probability becomes $1-\varepsilon$. So we want the largest m such that $e^{-\frac{m^2}{2n}} \geq 1-\varepsilon$. This means we must have

$$-\frac{m^2}{2n} \geq \ln(1-\varepsilon) = -\ln\left(\frac{1}{1-\varepsilon}\right),$$

or equivalently

$$m \leq \sqrt{2\ln\left(\frac{1}{1-\varepsilon}\right)} \times \sqrt{n}. \tag{4}$$

For $\varepsilon = \frac{1}{2}$, the coefficient in front of $\sqrt{n}$ is $\sqrt{2\ln 2} \approx 1.177$. So the bottom line is that we can hash approximately $m = \lfloor 1.177\sqrt{n} \rfloor$ keys before the probability of a collision reaches $\frac{1}{2}$. Recall that our calculation was only approximate; so we should go back and get a feel for how much error we made. We can do this by using Equation (1) to compute the exact largest value $m = m_0$ for which $\mathbb{P}[A]$ remains above $\frac{1}{2}$, for a few sample values of n. Then we can compare these values with our estimate $m \approx 1.177\sqrt{n}$:

n	10	20	50	100	200	365	500	1000	10^4	10^5	10^6
$1.177\sqrt{n}$	3.7	5.3	8.3	11.8	16.6	22.5	26.3	37.3	118	372	1177
exact m_0	4	5	8	12	16	22	26	37	118	372	1177

From the table, we see that our approximation is very good even for small values of n. When n is large, the error in the approximation becomes negligible. (Note in particular that when $n = 365$ we get $m_0 = 22$. This corresponds to the fact that, when there are 22 people in a room, the probability that any two share a birthday is less than $\frac{1}{2}$, but adding one more person pushes this probability above $\frac{1}{2}$.)

If instead we were interested in keeping the collision probability below (say) $\varepsilon = \frac{1}{20}$ (i.e., 5%), Equation (4) implies that we could hash at most $m = \sqrt{(2\ln(20/19))n} \approx 0.32\sqrt{n}$ keys. Of course, this number is a bit smaller than $1.177\sqrt{n}$ because our collision probability is now smaller. But no matter what "confidence" probability we specify, our critical value of m will always be $c\sqrt{n}$ for some constant c (which depends on the confidence).

2 Coupon Collecting

In the *coupon collecting* problem, our goal is to collect a set of n different baseball cards. We get the cards by buying boxes of cereal: each box contains exactly one card, and it is equally likely to be any of the n cards. How many boxes do we need to buy until we have collected at least one copy of every card?

This problem too can be modelled with balls and bins, where the question now is: "How many balls should one throw in order to have at least one ball in each of the n bins?"

This question can again be addressed using the union bound method on the experiment of throwing m balls into n bins. First we calculate the probability that the *first* bin is empty. A ball misses the bin with probability

$(1-\frac{1}{n})$ and, using the product rule for independent events, we can conclude that all m balls miss the bin with probability $(1-\frac{1}{n})^m$. For large n, we can approximate this value by $e^{-\frac{m}{n}}$, using the standard exponential approximation $(1-\frac{1}{n})^n \approx e^{-1}$. (This approximation is actually very good, even for moderate values of n: try it out yourself!)

Now we can use the union bound to bound the probability that *any* of the n bins is empty. Writing A for the event that any of the bins is empty, and A_i for the event that the ith bin is empty, we have

$$\mathbb{P}[A] = \mathbb{P}\Big[\bigcup_{i=1}^{n} A_i\Big] \leq n \times e^{-\frac{m}{n}}.$$

Finally, by choosing $m = n\ln n + n$, we see that the probability that any bin is empty is at most $1/e$. Thus, with probability at least $1 - 1/e$ (which is a bit larger than $\frac{1}{2}$), we will have collected all n coupons after buying at most $n\ln n + n$ cereal boxes.

This rough calculation actually gives us a pretty accurate answer: more sophisticated calculations can be used to show that, even for moderate values of n, the number of boxes required to collect all the coupons is close to $n\ln n$. To interpret this result, note that we certainly have to buy at least n boxes in order to collect all the coupons; but we would have to be incredibly lucky to get them all in only n boxes! We can view the $\ln n$ factor as the "overhead" in the number of boxes we have to buy in order to be pretty sure we've got all the coupons.

Despite its folksy description, coupon collecting arises frequently in the analysis of many randomized processes and algorithms: e.g., how long does it take for all n components in some system to fire, given that a random one fires each second?

3 Load Balancing

An important practical issue in distributed computing is how to spread the workload in a distributed system among its processors. Here we investigate an extremely simple scenario that is both fundamental in its own right and also establishes a baseline against which more sophisticated methods should be judged.

Suppose we have m identical jobs and n identical processors. Our task is to assign the jobs to the processors in such a way that no processor is too heavily loaded. Of course, there is a simple optimal solution here: just divide up the jobs as evenly as possible, so that each processor receives either $\lceil \frac{m}{n} \rceil$ or $\lfloor \frac{m}{n} \rfloor$ jobs. However, this solution requires a lot of centralized control, and/or a lot of communication: the workload has to be balanced evenly either by a powerful centralized scheduler that talks to all the processors, or by the exchange of many messages between jobs and processors. This kind of operation is very costly in most distributed systems. The question therefore is: What can we do with little or no overhead in scheduling and communication cost?

3.1 Back to Balls and Bins

The first idea that comes to mind here is... balls and bins! In other words, each job simply selects a processor uniformly at random and independently of all others, and goes to that processor. (Make sure you believe that the probability space for this experiment is the same as the one for balls and bins.) This scheme requires no communication. However, presumably it will not in general achieve an optimal balancing of the load. Let A_k be the event that the load of some processor is at least k. As designers or users of this load balancing scheme, here's one question we might care about:

Question: Find the smallest value k such that $\mathbb{P}[A_k] \leq \frac{1}{2}$.

If we have such a value k, then we will know that, with high probability (at least $\frac{1}{2}$), every processor in our system will have a load at most k. This will give us a good idea about the performance of the system. Of course, as with our hashing application, there's nothing special about the value $\frac{1}{2}$; we are just using this for illustration. Essentially the same analysis can be used to find k such that $\mathbb{P}[A_k] \leq 0.05$ (i.e., 95% confidence), or any other value we like. Indeed, we can even find the k's for several different confidence levels and thus build up a more detailed picture of the behavior of the scheme. To simplify our problem, we will also assume from now on that $m = n$ (i.e., the number of jobs is the same as the number of processors). With a bit more work, we could generalize our analysis to other values of m.

3.2 Applying the Union Bound

From the hashing application, we know[1] that we get collisions with high probability already when $m \approx 1.177\sqrt{n}$. So, when $m = n$, the maximum load will certainly be larger than 1 (with high probability). But how large will it be? If we try to analyze the maximum load directly, we run into the problem that it depends on the number of jobs at *every* processor (or equivalently, the number of balls in every bin). Since the load in one bin depends on those in the others, this becomes very tricky. Instead, what we will do is analyze the load in any *one* bin, say bin 1; this will be fairly easy. Let $A_k(1)$ be the event that the load in bin 1 is at least k. What we will do is to find the smallest k such that

$$\mathbb{P}[A_k(1)] \leq \frac{1}{2n}. \tag{5}$$

Since all the bins are identical, we will then know that, for the same k,

$$\mathbb{P}[A_k(i)] \leq \frac{1}{2n}, \qquad \text{for } i = 1, 2, \ldots, n,$$

where $A_k(i)$ is the event that the load in bin i is at least k. But now, since the event A_k is exactly the union of the events $A_k(i)$ (do you see why?), we can use the union bound:

$$\mathbb{P}[A_k] = \mathbb{P}\left[\bigcup_{i=1}^{n} A_k(i)\right] \leq \sum_{i=1}^{n} \mathbb{P}[A_k(i)] \leq n \times \frac{1}{2n} = \frac{1}{2}.$$

It is worth standing back to notice what we did here: we wanted to conclude that $\mathbb{P}[A_k] \leq \frac{1}{2}$. We could not analyze A_k directly, but we knew that $A_k = \bigcup_{i=1}^{n} A_k(i)$, for much simpler events $A_k(i)$. Since there are n events $A_k(1), \ldots, A_k(n)$, and all have the same probability, it is enough for us to show that $\mathbb{P}[A_k(i)] \leq \frac{1}{2n}$; the union bound then guarantees that $\mathbb{P}[A_k] \leq \frac{1}{2}$. This kind of reasoning is very common in applications of probability in engineering contexts like Computer Science.

Now all that is left to do is to find the smallest k that satisfies (5). That is, we wish to bound the probability that bin 1 has at least k balls (and find the smallest value of k so that this probability is smaller than $\frac{1}{2n}$). We start by observing that for the event $A_k(1)$ to occur (that bin 1 has at least k balls), there must be some subset S of exactly k balls such that all balls in S ended up in bin 1. We can say this more formally as follows: for a subset S (where $|S| = k$), let B_S be the event that all balls in S land in bin 1. Then the event $A_k(1)$ is equal to the union of the events B_S over all sets S of cardinality k. Thus we have:

$$\mathbb{P}[A_k(1)] = \mathbb{P}[\bigcup_{S} B_S].$$

[1] Note that this problem of finding the heaviest load in load balancing is identical to asking what the length of the longest linked-list in the hash-table would be, when we are storing n items in a hash table of size n.

We can use the union bound on $\mathbb{P}[\bigcup_S B_S]$:

$$\mathbb{P}\left[\bigcup_S B_S\right] \leq \sum_S \mathbb{P}[B_S].$$

There are $\binom{n}{k}$ sets we are summing over, and for each set S, $\mathbb{P}[B_S]$ is simple: it is just the probability that all k balls in S land in bin 1, which is $\frac{1}{n^k}$. Using these observations and the above equations, we can compute an upper bound on $\mathbb{P}[A_k(1)]$:

$$\mathbb{P}[A_k(1)] \leq \binom{n}{k}\frac{1}{n^k}.$$

Hence, to achieve our original goal of satisfying (5), we need to find the smallest k so that $\binom{n}{k}\frac{1}{n^k} \leq \frac{1}{2n}$. First note that

$$\binom{n}{k}\frac{1}{n^k} = \frac{n(n-1)\cdots(n-k+1)}{k!}\frac{1}{n^k} \leq \frac{1}{k!},$$

so $\binom{n}{k}\frac{1}{n^k} \leq \frac{1}{2n}$ if $\frac{1}{k!} \leq \frac{1}{2n}$. Taking logs, this is equivalent to $\ln k! \geq \ln(2n)$. Using a coarse form of Stirling's approximation $\ln k! \approx k\ln k - k$ for large k, we conclude that $\ln k! \approx \ln(2n)$ if k is chosen to be

$$k \approx \frac{\ln n}{\ln\ln n},$$

for large n. This approximation is actually in line with the value obtained by more detailed calculations, and for large n is a good estimate of the maximum load. In fact, a more careful analysis suggests that the value $k = \frac{2\ln n}{\ln\ln n}$ is a good upper bound on the maximum load for reasonable values of n. (The extra factor of 2 wipes out some lower order terms.) The following table shows the growth of this bound, giving a good idea of how the maximum load behaves as a function of n.

n	10	20	50	100	500	1000	10^4	10^5	10^6	10^7	10^8	10^{15}
$\frac{2\ln n}{\ln\ln n}$	5.5	5.5	5.7	6.0	6.8	7.2	8.2	9.4	10.6	11.6	12.6	20

Finally, here is one punchline from the above. Let's say the US population is about 350 million. Suppose we mail 350 million items of junk mail, each one with a random US address. Then with probability at least $\frac{1}{2}$, no one person anywhere will receive more than about a dozen items!