A family of Applications: Hashing and Load Balancing

In this lecture, we will see our first glimpse of a "killer app" of probability in EECS. Probability helps us understand quantitatively how much of a resource we need when demands are somewhat random. There is a wide range of applications, especially in networking where understanding randomness is the key to the idea of "stochastic multiplexing." The basic idea behind this (which are explored a lot more in EECS122 and EECS126) is that even though the universe of potential demand is vast, in the real world, the actual demands are random and not all potential "users" of the resource will show up at once. Consequently, we can "overbook" and share the resource while actually not provisioning for the worst-case total demand.

A naive interpretation of probability would completely neglect the random fluctuations associated with chance events and would assume perfectly regular demand. We shall see how that isn't wise by looking at two applications:

1. Suppose a hash function distributes keys evenly over a table of size n. How many (randomly chosen) keys can we hash before the probability of a collision exceeds (say) $\frac{1}{2}$?
2. Consider the following simple load balancing scenario. We are given m jobs and n machines; we allocate each job to a machine uniformly at random and independently of all other jobs. What is a likely value for the maximum load on any machine?

As we shall see, this question can be tackled by an analysis of the balls-and-bins probability space which we have already encountered.

Application 1: Hash tables

As you may recall, a hash table is a data structure that supports the storage of sets of keys from a (large) universe U (say, the names of all 313m people in the US). The operations supported are ADDing a key to the set, DELETEing a key from the set, and testing MEMBERship of a key in the set. The hash function h maps U to a table T of modest size. 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.

The question we are interested in here is the following: suppose our hash table T has size n, and that our hash function h distributes U evenly over T.[1] Assume that the keys we want to store are chosen uniformly at random and independently from the universe U. What is the largest number, m, of keys we can store before the probability of a collision reaches $\frac{1}{2}$?

[1] I.e., $|U| = \alpha n$ (the size of U is an integer multiple α of the size of T), and for each $y \in T$, the number of keys $x \in U$ for which $h(x) = y$ is exactly α.

Let's 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 keys are chosen uniformly and independently from U, and since the hash function distributes keys evenly over the table, we can see each key (ball) as choosing a hash table location (bin) uniformly and independently from T. 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 $\Pr[A]$ will decrease as m increases (with n fixed). Our goal is to find the largest value of m such that $\Pr[A]$ remains above $\frac{1}{2}$. [Note: Really we are looking at different sample spaces here, one for each value of m. So it would be more correct to write $\Pr_m$ rather than just Pr, to make clear which sample space we are talking about. However, we will omit this detail.]

Let's fix the value of m and try to compute $\Pr[A]$. 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 Note 10 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'll assume that $m \leq n$.

Now we can calculate the probability of no collision:

$$\begin{aligned} \Pr[A] &= \frac{n(n-1)(n-2)\dots(n-m+1)}{n^m} \\ &= \frac{n}{n} \times \frac{n-1}{n} \times \frac{n-2}{n} \times \cdots \times \frac{n-m+1}{n} \\ &= \left(1-\frac{1}{n}\right) \times \left(1-\frac{2}{n}\right) \times \cdots \times \left(1-\frac{m-1}{n}\right). \end{aligned} \tag{1}$$

Before going on, let's pause to observe that we could compute $\Pr[A]$ 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 $1, 2, \dots, i-1$. Then

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

Notice that we get the same answer. [You should make sure you see how we obtained the conditional probabilities in the second line above. For example, $\Pr[A_3|A_1 \cap A_2]$ 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 $m = 1, 2, 3, \dots$ into (1) until we find that $\Pr[A]$ drops below $\frac{1}{2}$. The corresponding value of m (minus 1) is what we want.

But this 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 $\Pr[A]$ for $m = 1, 2, 3, \dots$. 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 $\Pr[A]$ 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.[2] We get

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

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

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

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'll just write $\ln(1-x) \approx -x$, secure in the knowledge that we could make this approximation precise (in the sense of having upper and lower bounds) if necessary.

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

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

Note that we've 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. Once we're done, we'll 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 $\Pr[A]$:

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

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

$$-\frac{m^2}{2n} \geq \ln(\tfrac{1}{2}) = -\ln 2, \tag{4}$$

or equivalently

[2] This is a pretty standard trick and is safe because log is an invertible function. Invertibility is all we require when applying this sort of trick in the context of an exact equation. When approximating, it is often also desirable that the function we apply to both sides of an equation is continuous and monotonic — nice properties also possessed by log.

$$m \leq \sqrt{(2\ln 2)n} \approx 1.177\sqrt{n}. \tag{5}$$

So the bottom line is that we can hash approximately $\underline{m = \lfloor 1.177\sqrt{n} \rfloor \text{ keys}}$ 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 value $m = m_0$ at which $\Pr[A]$ drops below $\frac{1}{2}$, for a few sample values of n. Then we can compare these values with our estimate $m = 1.177\sqrt{n}$.

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.

Why $\frac{1}{2}$?

Our hashing question asked when the probability of a collision rises to $\frac{1}{2}$. Is there anything special about $\frac{1}{2}$? Not at all. What we did was to (approximately) compute $\Pr[A]$ (the probability of no collision) as a function of m, and then find the largest value of m for which our estimate is smaller than $\frac{1}{2}$. If instead we were interested in keeping the collision probability below (say) 0.05 (= 5%), we would just replace $\frac{1}{2}$ by 0.95 in equation (4). If you work through the last piece of algebra again, you'll see that this gives us the critical value $m = \sqrt{(2\ln(20/19))n} \approx 0.32\sqrt{n}$, which of course is a bit smaller than before 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).

The birthday paradox revisited

Recall from a previous lecture the birthday "paradox": what is the probability that, in a group of m people, no two people have the same birthday? The problem we have solved above is essentially just a generalization of the birthday problem: the bins are the birthdays and the balls are the people, and we want the probability that there is no collision. The above table at $n = 365$ tells us for what value of m this probability drops below $\frac{1}{2}$: namely, 23.

Using the union bound

Here's a cruder way to do the same calculation we did earlier. There are exactly $k = \binom{m}{2} = \frac{m(m-1)}{2}$ possible pairs among our m keys. Imagine these are numbered from 1 to $\binom{m}{2}$ (it doesn't matter how). Let A_i denote the event that pair i has a collision (i.e., both are hashed to the same location). Then the event $\overline{A}$ that *some* collision occurs can be written $\overline{A} = \bigcup_{i=1}^{k} A_i$. What is $\Pr[A_i]$? We claim it is just $\frac{1}{n}$, for every i. (Why?) So, using the union bound from the last lecture, we have

$$\Pr[\overline{A}] \leq \sum_{i=1}^{k} \Pr[A_i] = k \times \frac{1}{n} = \frac{m(m-1)}{2n} \approx \frac{m^2}{2n}.$$

This means that the probability of having a collision is less than $\frac{1}{2}$ provided $\frac{m^2}{2n} \leq \frac{1}{2}$, i.e., provided $m \leq \sqrt{n}$. This is a somewhat more restrictive condition than the one in equation (5) that we derived earlier, and it gives answers that are less accurate than those in our earlier table. However, in terms of the dependence on n both conditions are the same (both are of the form $m = O(\sqrt{n})$).

Application 2: Load balancing

One of the most pressing practical issues in distributed computing is how to spread the workload in a distributed system among its processors. This is a huge question, still very much unsolved in general. 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.

These kinds of load balancing problems are encountered in every major data center. So the Googles, Facebooks, Snapchats, Yahoos, Amazons, Microsofts, Pinterests, Netflixes, Tumblrs, etc. of the world deal with these sort of issues constantly.

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?

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 won't in general achieve an optimal balancing of the load. Let X be the maximum loading of any processor under our randomized scheme. Note that X isn't a fixed number: its value depends on the outcome of our balls and bins experiment.[3] 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

$$\Pr[X \geq k] \leq \tfrac{1}{2}.$$

If we have such a value k, then we'll know that, with good probability (at least $\frac{1}{2}$), the maximum load on any processor in our system won't exceed 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're just using this for illustration. As you can check later, essentially the same analysis can be used to find k such that $\Pr[X \geq k] \leq 0.05$ (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'll 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. (**You are encouraged to do this on your own to see how well you understand this material.**)

From Application 1 we know[4] that we get collisions already when $m \approx 1.177\sqrt{n}$. So when $m = n$ the maximum load will almost certainly be larger than 1 (with good probability). But how large will it be? If

[3] In fact, X is called a **random variable: we'll define this properly in the next lecture.**

[4] It is pretty clear that this problem of finding the heaviest load in load balancing is identical to asking what the length of the longest linked-list in a hash-table would be.

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'll do is analyze the load in any *one* bin, say bin 1; this will be fairly easy. Call the load in bin 1 X_1 (another random variable). What we'll do is find k such that

$$\Pr[X_1 \geq k] \leq \tfrac{1}{2n}. \tag{6}$$

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

$$\Pr[X_i \geq k] \leq \tfrac{1}{2n} \qquad \text{for } i = 1, 2, \ldots, n,$$

where X_i is the load in bin i. But now, since the event $X \geq k$ is exactly the union of the events $X_i \geq k$ (do you see why?), we can use the "Union Bound" from the previous lecture note:

$$\begin{aligned}
\Pr[X \geq k] &= \Pr[\textstyle\bigcup_{i=1}^{n} (X_i \geq k)] \\
&\leq \sum_{i=1}^{n} \Pr[X_i \geq k] \\
&\leq n \times \frac{1}{2n} \\
&= \frac{1}{2}.
\end{aligned}$$

It's worth standing back to notice what we did here: we wanted to conclude that $\Pr[A] \leq \frac{1}{2}$, where A is the event that $X \geq k$. We couldn't analyze A directly, but we knew that $A = \bigcup_{i=1}^{n} A_i$, for much simpler events A_i (namely, A_i is the event that $X_i \geq k$). Since there are n events A_i, and all have the same probability, it is enough for us to show that $\Pr[A_i] \leq \frac{1}{2n}$; the union bound then guarantees that $\Pr[A] \leq \frac{1}{2}$. This kind of reasoning[5] is very common in applications of probability in engineering contexts like Computer Science.

Now let's get back to our problem. Recall that we've reduced our task to finding k such that

$$\Pr[X_1 \geq k] \leq \tfrac{1}{2n},$$

where X_1 is the load in bin 1. It's not hard to write down an *exact* expression for $\Pr[X_1 = j]$, the probability that the load in bin 1 is precisely j:

$$\Pr[X_1 = j] = \binom{n}{j} \left(\tfrac{1}{n}\right)^j \left(1 - \tfrac{1}{n}\right)^{n-j}. \tag{7}$$

This can be seen by viewing[6] each ball as a biased coin toss: Heads corresponds to the ball landing in bin 1, Tails to all other outcomes. So the Heads probability is $\frac{1}{n}$; and all coin tosses are (mutually) independent. As we saw in earlier lectures, (7) gives the probability of exactly j Heads in n tosses. This will be formalized later in the notes when we introduce the binomial distribution.

[5] In fact, this kind of reasoning is why we require this course at all. It is very important for you to learn that often the response to a hard question is to simply ask an easier one. Then, after answering the easier one, you can hopefully connect it back to the original harder question. Where better to learn this aesthetic than in a mathematical area? After all, for most of you, math is your paradigmatic exemplar of an area that is black/white with no room for subjectivity and maneuvering around a question. By learning to appreciate changing the question here, the hope is that you will be able to do it effectively in the more complicated real-world scenarios you are likely to encounter in practice.

[6] Once again we see that one of the critical skills in creative mathematical thinking is the ability to find analogies. Finding analogies is something that practicing engineers do every day as they tackle hard problems.

Thus we have

$$\Pr[X_1 \geq k] = \sum_{j=k}^{n} \Pr[X_1 = j] = \sum_{j=k}^{n} \binom{n}{j} \left(\tfrac{1}{n}\right)^j \left(1 - \tfrac{1}{n}\right)^{n-j}. \tag{8}$$

Now in some sense we are done: we could try plugging values $k = 1, 2, \ldots$ into (8) until the probability drops below $\frac{1}{2n}$. However, as in the hashing example, it will be much more useful if we can massage equation (8) into a cleaner form from which we can read off the value of k more directly. So once again we need to do a few calculations and approximations:

$$\begin{aligned} \Pr[X_1 \geq k] &= \sum_{j=k}^{n} \binom{n}{j} \left(\tfrac{1}{n}\right)^j \left(1 - \tfrac{1}{n}\right)^{n-j} \\ &\leq \sum_{j=k}^{n} \left(\frac{n\mathrm{e}}{j}\right)^j \left(\tfrac{1}{n}\right)^j \\ &= \sum_{j=k}^{n} \left(\frac{\mathrm{e}}{j}\right)^j \end{aligned} \tag{9}$$

In the second line here, we used the standard approximation[7] $(\frac{n}{j})^j \leq \binom{n}{j} \leq (\frac{n\mathrm{e}}{j})^j$. Also, we tossed away the $(1 - \frac{1}{n})^{n-j}$ term, which is permissible because doing so can only increase the value of the sum (i.e., since $(1 - \frac{1}{n})^{n-j} \leq 1$). It will turn out that we didn't lose too much by applying these bounds.

Now we're already down to a much cleaner form in (9). To finish up, we just need to sum the series. Again, we'll make an approximation to simplify our task, and hope that it doesn't cost us too much. (We'll verify this later[8].) The terms in the series in (9) go down at each step by a factor of at least $\frac{\mathrm{e}}{k}$. So we can bound the series by a *geometric* series, which is easy to sum:

$$\sum_{j=k}^{n} \left(\frac{\mathrm{e}}{j}\right)^j \leq \left(\tfrac{\mathrm{e}}{k}\right)^k \left(1 + \tfrac{\mathrm{e}}{k} + \left(\tfrac{\mathrm{e}}{k}\right)^2 + \cdots\right) \leq 2\left(\tfrac{\mathrm{e}}{k}\right)^k, \tag{10}$$

where the second inequality holds provided we assume that $k \geq 2\mathrm{e}$ (which it will be, as we shall see in a moment).

So what we have now shown is the following:

$$\Pr[X_1 \geq k] \leq 2\left(\tfrac{\mathrm{e}}{k}\right)^k \tag{11}$$

(provided $k \geq 2\mathrm{e}$). Note that, even though we have made a few approximations, inequality (11) is completely valid: all our approximations were "$\leq$", so we always have an *upper* bound on $\Pr[X_1 \geq k]$. [You should go back through all the steps and check this.]

Recall from (6) that our goal is to make the probability in (11) less than $\frac{1}{2n}$. We can ensure this by choosing k so that

$$\left(\tfrac{\mathrm{e}}{k}\right)^k \leq \tfrac{1}{4n}. \tag{12}$$

[7] Engineers and mathematicians carry around a bag of tricks for replacing complicated expressions like $\binom{n}{j}$ with simpler approximations. This is just one of these. It isn't too hard to prove the lower bound, i.e., that $\binom{n}{j} \geq (\frac{n}{j})^j$. The upper bound is a bit trickier, and makes use of Stirling's approximation $n! \approx \sqrt{2\pi n}(\frac{n}{\mathrm{e}})^n$, which implies that $j! \geq (\frac{j}{\mathrm{e}})^j$. You worked with Stirling's approximation in the homework and used it to compute bounds, and this is related to that. You should be building your own personal bag of tricks as you go through classes and as you do real-world projects later.

[8] Making approximations is challenging for students who are used to thinking about Math as a subject with absolutely no wiggle-room or subjectivity. In reality, the rigor and verifiability of math is what frees us to make approximations and simplifications all over the place. We can always come back and check our answers with a simulation or turn the approximation into a pair of inequalities using formal proofs. If your approximation ended up costing too much, we can just walk back through the steps of the derivation and see where we introduced the looseness.

Now we are in good shape: given any value n for the number of jobs/processors, we just need to find the smallest value $k = k_0$ that satisfies inequality (12). We will then know that, with probability at least $\frac{1}{2}$, the maximum load on any processor is at most k_0. The table below shows the values of k_0 for some sample values of n. Remember that k_0 is itself an upper bound to the truly safe value. It is highly recommended that you perform the experiment and compare these values of k_0 with what happens in practice.

n	10	20	50	100	500	1000	10^4	10^5	10^6	10^7	10^8	10^{15}
exact k_0	6	6	7	7	8	8	9	10	11	12	13	19
$\ln(4n)$	3.7	4.4	5.3	6.0	7.6	8.3	10.6	13.9	15.2	17.5	19.8	36
$\frac{2\ln n}{\ln\ln n}$	5.6	5.4	5.8	6.0	6.8	7.2	8.2	9.4	10.6	11.6	12.6	20

Can we come up with a formula for k_0 as a function of n (as we did for the hashing problem)? Well, let's take logs in (12) and then multiply both sides by -1 to make all the expressions positive[9] by inspection.

$$k(\ln k - 1) \geq \ln(4n). \tag{13}$$

From this, we might guess that $k = \ln(4n)$ is a good value for k_0. Plugging in this value of k makes the left-hand side of (13) equal to $\ln(4n)(\ln\ln(4n) - 1)$, which is certainly bigger than $\ln(4n)$ provided $\ln\ln(4n) \geq 2$, i.e., $n \geq \frac{1}{4}e^{e^2} \approx 405$. So for $n \geq 405$ we can claim that the maximum load is (with probability at least $\frac{1}{2}$) no larger than $\ln(4n)$. The table above lists the values of $\ln(4n)$ for comparison with k_0. As might be expected (since we basically just neglected the $\ln k - 1$ term), the estimate is quite good for small n, but becomes a bit too conservative when n is very large.

For large n we can do better as follows. If we plug the value $k = \frac{\ln n}{\ln\ln n}$ into the left-hand side of (13), it becomes

$$\frac{\ln n}{\ln\ln n}(\ln\ln n - \ln\ln\ln n - 1) = \ln n\left(1 - \frac{\ln\ln\ln n + 1}{\ln\ln n}\right). \tag{14}$$

Now when n is large this is *just barely* smaller than the right-hand side, $\ln(4n)$. Why? Because the second term inside the parentheses goes to zero[10] as $n \to \infty$, and because $\ln(4n) = \ln n + \ln 4$, which is qualitatively close to $\ln n$ when n is large (since $\ln 4$ is a fixed small constant). So we can conclude that, for large values of n, the quantity $\frac{\ln n}{\ln\ln n}$ should be a pretty good estimate of k_0. Actually for this estimate to become good n has to be (literally) astronomically large. For more civilized values of n, we get a better estimate by taking $k = \frac{2\ln n}{\ln\ln n}$. The extra factor of 2 helps to wipe out the lower order terms (i.e., the second term in the parenthesis in (14) and the $\ln 4$) more quickly. The table above also shows the behavior of this estimate for various values of n.

Finally, here is one punchline from Application 2. Let's say the total US population is about 313 million. Suppose we mail 313 million items of junk mail, each one with a random US address. Then (see the above table) with probability at least $\frac{1}{2}$, no one person anywhere will receive more than about a (baker's) dozen items!

Further questions

After reading this far, you should be curious and have many more questions about the behavior of such totally random load-balancing schemes. For example, we have taken the perspective of the most loaded machine. What about the "typical" machine? What is the most common load? That's an interesting question, but

[9] This is generally a good idea. Your intuition is better about positive numbers than it is for negative numbers. So massage expressions so that the signs are immediately apparent.

[10] To see this, note that it is of the form $\frac{\ln z + 1}{z}$ where $z = \ln\ln n$, and of course $\frac{\ln z + 1}{z} \to 0$ as $z \to \infty$.

rather easy to answer based on the calculations we have already done. An even easier question concerns the least loaded machine: how loaded is it? That one is extremely easy for the case $m = n$. But it becomes more interesting if we let m get bigger than n.

What if we turned the perspective around. Instead of thinking about the jobs as a burden, what if the jobs are good? Then it is natural to wonder how many jobs we have to randomly distribute to make sure that it is likely that every server has at least one job. To answer that question using a straightforward calculation, we are going to build a little bit more machinery and formulate the language of random variables and expectations.