In this lecture we give two applications of randomized algorithms. The first one is load balancing, where the goal is to spread tasks evenly among resources. We model this as a balls and bins problem. The second is hashing, where we tradeoff storage space with lookup time.

7.1 Randomized Load Balancing and Balls and Bins Problem

In the last lecture, we talked about the load balancing problem as a motivation for randomized algorithms. Here we examine this problem more closely. In general, load balancing is a problem to distribute tasks among multiple resources. One example is assigning students to advisors. On any given day, each of m students will visit one of n advisors. We want to make an assignment of students to advisors so that regardless of the subset of m students under consideration, every advisor gets more or less an even load of students. An easy randomized approach is to assign each student uniformly at random to any advisor. Let us analyze this approach. We can model this as throwing balls into bins. Here, balls can be considered as tasks and bins as resources.

Balls and Bins

Consider the process of throwing m balls into n bins. Each ball is thrown into a uniformly random bin, independent of other balls, which implies that the probability that a ball falls into any given bin is $1/n$. Based on this process, we can ask a variety of questions. But first we define some notation: Fall_i^b denotes the event of ball i falling into bin b and Col_{ij} be the event that ball i and ball j collide.

Question 1: *What is the probability of any two balls "colliding", or falling into one bin?*

We can determine this using Bayes rule:

$$\begin{aligned}\mathbf{Pr}[\text{Col}_{12}] &= \sum_{b=1}^{n} \mathbf{Pr}\left[\text{Fall}_2^b | \text{Fall}_1^b\right] \mathbf{Pr}\left[\text{Fall}_1^b\right] \\ &= \sum_{b=1}^{n} \frac{1}{n} \mathbf{Pr}\left[\text{Fall}_1^b\right] = \frac{1}{n} \end{aligned} \tag{7.1.1}$$

Question 2: *What is the expected number of collisions when we throw m balls?*

Note that we count collisions over distinct and unordered pairs, that is, if three balls fall into one bin, three collisions are counted.

To answer the second question, we use an indicator random variable X_{ij} to indicate whether the event Col_{ij} happens. $X_{ij} = 1$ if Col_{ij} happens, $X_{ij} = 0$ if Col_{ij} doesn't happen. As an aside, indiator random variables are especially important if we are trying to count the number of occurences of some events in any setting. Note that indicator variables are Bernoulli variables since they take on values 0 or 1. We also define X to be the number of collisions, i.e. $X = \sum_{i \neq j} X_{ij}$. Then we can

calculate $\mathbf{E}[X]$ using the linearity of expectation, as follows:

$$\begin{aligned}\mathbf{E}[X] &= \sum_{i \neq j} \mathbf{E}[X_{ij}] \\ &= \sum_{i \neq j} \mathbf{Pr}[X_{ij} = 1] = \frac{1}{n}\binom{m}{2}\end{aligned} \tag{7.1.2}$$

The answer to this question demonstrates an interesting phenomenon, the **Birthday Paradox**. The birthday problem asks, how many people must there be in a room for there to be at least a 50% chance that two of them were born on the same day of the year (assuming birthdays are distributed evenly)? Our calculation above answers a related question: how many people must there be to expect at least one birthday collision? The answer is surprisingly few: we can view people as balls and days as bins; Then solving $\mathbf{E}[X] = \frac{1}{365}\binom{m}{2} = 1$ we get $m \geq 23$.

For the remainder of our discussion, we assume $m = n$ for simplicity. All of the analysis can be generalized to $m \neq n$.

Question 3: *What is the probability of a particular bin being empty? What is the expected number of empty bins?*

The probability of a particular ball not falling into a particular bin is $1 - \frac{1}{n}$. Thus, we have

$$\mathbf{Pr}[\text{bin } i \text{ is empty}] = \left(1 - \frac{1}{n}\right)^n \to \frac{1}{e} \tag{7.1.3}$$

Using indicator random variables, as before, we can show that the expected number of empty bins is $(1 - \frac{1}{n})^n \approx \frac{n}{e}$.

Question 4: *What is the probability of a particular bin having exactly k balls?*

$$\begin{aligned}\mathbf{Pr}[\text{bin } i \text{ has } k \text{ balls}] &= \binom{n}{k}\left(\frac{1}{n}\right)^k\left(1 - \frac{1}{n}\right)^{n-k} \\ &\leq \frac{n^k}{k!}\frac{1}{n^k} = \frac{1}{k!}\end{aligned} \tag{7.1.4}$$

Question 5: *What is the probability of a particular bin having at least k balls?*

If we look at any subset of balls of size k, then the probability that the subset of balls falls into bin i is $\left(\frac{1}{n}\right)^k$. Note that we no longer have the $\left(1 - \frac{1}{n}\right)^{n-k}$ factor, because we don't care about where the rest of the balls fall. We then take a union bound of these probabilities over all $\binom{n}{k}$ subsets of size k. The events we are summing over, though, are not disjoint. Therefore, we can only show that the probability of a bin having at least k balls is at most $\binom{n}{k}\left(\frac{1}{n}\right)^k$.

$$\mathbf{Pr}[\text{bin } i \text{ has at least } k \text{ balls}] \leqslant \binom{n}{k}\left(\frac{1}{n}\right)^k \leq \left(\frac{e}{k}\right)^k \tag{7.1.5}$$

The last inequality follows from Stirling's approximation.

Expected maximum load

Having examined some basic properties of the balls and bins problem, we now move on to the key quantity in load balancing: the maximum load on any bin.

Specifically, we want to use the calculation done for Question 5 to determine some k so that the probability that there exists a bin with at least k balls is very small. What should k be to have $\frac{1}{k^k} \approx \frac{1}{n^2}$? The solution is $k = \mathcal{O}\left(\frac{\ln n}{\ln \ln n}\right)$.

Therefore, we present the following theorem:

Theorem 7.1.1 *With high probability, i.e.* $1 - \frac{1}{n}$*, all bins have at most* $\frac{3 \ln n}{\ln \ln n}$ *balls.*

Proof: Let $k = \frac{3 \ln n}{\ln \ln n}$. From equation (7.1.5), we have

$$\begin{aligned}
\mathbf{Pr}[\text{bin } i \text{ has at least } k \text{ balls}] &\leqslant \left(\frac{e}{k}\right)^k = \left(\frac{e \ln \ln n}{3 \ln n}\right)^{\frac{3 \ln n}{\ln \ln n}} \\
&\leqslant \exp\left(\frac{3 \ln n}{\ln \ln n}(\ln \ln \ln n - \ln \ln n)\right) \\
&= \exp\left(-3 \ln n + \frac{3 \ln n \ln \ln \ln n}{\ln \ln n}\right)
\end{aligned}$$

When n is large enough,

$$\mathbf{Pr}[\text{bin } i \text{ has at least } k \text{ balls}] \leqslant \exp\{-2 \ln n\} = \frac{1}{n^2}$$

Using Union Bound, we have

$$\mathbf{Pr}[\text{there exists a bin with at least } k \text{ balls}] \leqslant n \frac{1}{n^2} = \frac{1}{n}$$

which implies

$$\mathbf{Pr}[\text{all bins have at most } k \text{ balls}] \geqslant 1 - \frac{1}{n}$$

■

Note that if we throw balls uniformly at random into bins, while the balls are not uniformly distributed, every bin has equal probability of having k balls.

Theorem 7.1.1 shows that if we distribute load or balls independently and uniformly at random, we can get maximum load or largest number of balls in any bin of about $\frac{3 \ln n}{\ln \ln n}$ with high probability. We can also show that

$$\mathbf{E}[\text{maximum load}] = \frac{\ln n}{\ln \ln n}(1 + \mathcal{O}(1))$$

Can we improve this number and distribute things more evenly? In the previous process, at every step we pick a bin independently and uniformly at random and throw the ball into that bin. Instead suppose every time we pick two bins independently and uniformly at random and throw the ball

into the bin with fewer balls. This process gives us flexibility into dividing the load more evenly although it requires more randomness. It turns out that the expected maximum load becomes

$$\mathbf{E}[\text{maximum load given two choices}] = \frac{\ln \ln n}{\ln 2} + \mathcal{O}(1)$$

We see that this is a huge improvement over one random choice. More generally, we can allow ourselves t random choices for every ball, obtaining a maximum expected load of $\frac{\ln \ln n}{\ln t}$. The tradeoff between the amount of randomness used and the load achieved is best for $t = 2$. This phenomenon is called the *power of two choices*.

7.2 Hashing

Another problem related to load balancing is hashing, which can be used to maintain a data structure for a set of elements for fast look-up. Hashing has a number of applications that we won't discuss here. The basic setup is that we have a large universe of potential elements. We need to maintain an array or other data-structure over a subset of these with fast insert and look up (constant time per element). This mapping is called a hash function. The problem is formally described as this:

Given: A set of elements S with size $|S| = m$ from universe U ($|U| = u$). A data-structure with storage for n elements.

Goal: For $n = \mathcal{O}(m)$, design a mapping h from U to $[n]$ such that for every subset S of U of size m, the number of memory accesses required for an insert and a lookup is $\mathcal{O}(1)$ per element.

As a first attempt, we might try the following approaches:

- Store S in a binary search tree or other similar data-structure. Insert and lookup time is $O(\log m)$.
- Use an array of size u to record for every element of U whether it belongs to S or not. Insert and lookup time is $O(1)$ but storage requirement is prohibitively large.

Ideally we would like the performance of the data-structure to be similar to the second case, but at the same time require small storage space. This is ofcourse impossible to do deterministically, but randomization gives us an answer.

7.2.1 Ideal Situation - Truly Random

A simple solution for the mapping is to use a truly random function. There are a total of n^u functions mapping U to $[n]$. Suppose we pick a function uniformly at random from this set. Note that each element gets mapped to a uniformly random position in the array. Now, in order to insert an element i in the array, we just compute $h(i)$ and write a 1 in that position in the array. In order to lookup element i, we compute $h(i)$, and determine whether position $h(i)$ is 1 or 0.

Note that we may make an error on lookups with a small probability if two elements map to the same location in the array. We say that these two elements *collide*. One way to avoid making the

error is to build a link list under each array position and add every element that gets mapped to that position to the linked list. Then the look-up process for any element i will look like this: (1) Get the array position by calculating $h(i)$; (2) Scan the linked list under $h(i)$ to see if i exists in this linked list.

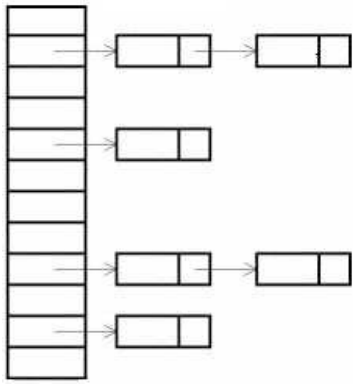

Figure 7.2.1: Chain-hashing

This hashing structure (in Figure 7.2.1) is called *chain-hashing*. As we know from our earlier analysis, the probability that two elements i and j collide is:

$$\mathbf{Pr}[\text{elements } i, j \in S \text{ collide}] = \frac{1}{n} \tag{7.2.6}$$

If we fix i, the expected number of collisions for i is:

$$\mathbf{E}[\text{number of collisions with } i] = \frac{m-1}{n} \tag{7.2.7}$$

If we pick n to be m or larger, this number is less than 1, so our expected lookup time is a constant.

However, there are two problems with this approach. First, the function h is completely random. It takes at least $u \log n$ bits to express and store this function, defeating the purpose of using a hash function. Second, the worst case lookup time is $\frac{\ln n}{\ln \ln n}$ (from Theorem 7.1.1), which is only slightly better than binary search.

7.2.2 2-Universal Family of Hash Functions

The first problem arises because we use too much randomness. Can we achieve the same collision probability with a smaller amount of randomness? Suppose that we draw a hash function uniformly at random from a family H of functions. The only property we need is that for every $x, y \in U$, we have

$$\mathbf{Pr}_{h \in H}\left[h(x) = h(y)\right] \leq \frac{1}{n} \tag{7.2.8}$$

A family H satisfying the above property is called a 2-*universal* hash family.

As an aside, sometimes we may want a stronger condition on the correlations between hash values of elements. A *strongly d-universal* hash family H is defined as: $\forall x_1, x_2, \ldots, x_d \in U,\ i_1, i_2, \ldots, i_d \in [n]$,

$$\mathbf{Pr}_{h \in H}\left[h(x_1) = i_1, h(x_2) = i_2, \ldots, h(x_d) = i_d\right] \leq \frac{1}{n^d} \tag{7.2.9}$$

For our bounds on the number of collisions, we only require 2-way independence between the mappings of the hash functions. Therefore a 2-universal family of hash functions is enough.

An Example of 2-Universal Hash Functions

Suppose that our universe U is given by $U = \{0,1\}^{\lg u} \setminus \{0^{\lg u}\}$, and the array we are mapping to is $A = \{0,1\}^{\lg n}$. Let $H = \{0,1\}^{\lg n \times \lg u}$, the set of all $\lg n \times \lg u$ 0-1 matrices. For any hash function $h \in H$, we define $h(x)$ to be h times x where h is a $\lg n \times \lg u$ matrix and x is a vector of size $\lg u$. The matrix h maps a long vector to a short uniformly random vector.

For example, for $u = 32$, $n = 8$, $x = (1,1,0,0,1)$, the matrix below maps x to $i = (0,1,1)$:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

From this matrix construction, we get the following claim:

Lemma 7.2.1 *Fix input $x, y \in S, (x \neq y)$. The probability that the j_{th} position in $h(x)$ is the same as the j_{th} position in $h(y)$ is $\frac{1}{2}$.*

Proof: For any position i in which x and y differ, if the corresponding position in h, h_{ij}, is 1, then $h(x)$ is different from $h(y)$ at position j. Otherwise, $h(x)$ is the same as $h(y)$ at position j. Therefore the claim. ■

Theorem 7.2.2 *The hash family H described above is a 2-universal family of hash functions.*

Proof: From Lemma 7.2.1 we know that the probability that any position in $h(x)$ and $h(y)$ is the same is $\frac{1}{2}$. Furthermore, these events corresponding to different positions in the vector are independent. Therefore the probability that $h(x) = h(y)$ is $\left(\frac{1}{2}\right)^{\lg n} = \frac{1}{n}$. Therefore H is 2-universal hash functions family. ■:

Finally, if mapping integer values: $[1, \ldots, u] \rightarrow [1, \ldots, n]$, a widely used hash family is to pick a prime $p(p \geq u)$, and $a, b \in [u], a \neq 0$, and $h_{a,b}(x) = ((ax + b)\ mod\ p)\ mod\ n$. This also turns out to be 2-universal.

7.2.3 Perfect hashing – reducing worst case loopkup time

We now turn to problem 2, namely that the worst case lookup time of a completely random hash function can be logarithmic. Can we reduce this to $O(1)$? Recall that if we use X as a random variable to represent the number of pairs of elements in S mapped to the same bin, its expectation,

the expected number of collisions, is given by

$$\mathbf{E}[X] = 1/n\binom{m}{2} \tag{7.2.10}$$

In order for the worst-case lookup time to be $O(1)$ the quantity on the right hand side of (7.2.10) must be $O(1)$ which means that n must be $O(m^2)$. In order to ensure that $\mathbf{E}[X] = O(1)$ we can repeatedly choose new hash functions h from a 2-Universal family of functions $H : U \to [n^2]$ until one is found which produces no more than $O(1)$ collisions in any bin.

Perfect hashing with linear space

An improvement on this method of hashing which reduces its space complexity is to choose $n \approx m$, and instead of using a linked list to represent elements that map to the same bin, to use another hash table of appropriate size to store these elements. Let b_i be the number of elements that map to bin i. Then in order to guarantee a constant lookup time in this second-level array, we need to allocate a storage size of b_i^2. Then the total amount of space used by this method is given by:

$$\text{total space} = n + \mathbf{E}\left[\sum_i b_i^2\right] \tag{7.2.11}$$

We can note that

$$\sum_i b_i^2 = \sum_i 2 * \binom{b_i}{2} + \sum_i b_i \tag{7.2.12}$$

The $\sum_i b_i$ term is equal to m, the total number of items stored. On the other hand, $\sum_i \binom{b_i}{2}$ is equal to the total number of collisions, which in expectation is $\frac{1}{n}\binom{m}{2}$. Noting that $m = n$ we get

$$\mathbf{E}[\text{total space}] = m + m + \frac{2}{m}\frac{m(m-1)}{2} < 3m \tag{7.2.13}$$

Therefore, using this approach we can get constant worst-case lookup time along with linear storage.