Advanced Algorithms: Lecture 4 and 5 Notes

Dana Randall Spring 2024

Sarthak Mohanty

Randomized Algorithms

What is a randomized algorithm? Recall that for an algorithm, we start with an input, feed it into a program, get an output. We are hoping it is **correct, efficient (in terms of input size), etc**. Now for a randomized algorithm, we also have a string of independent random 0-1 bits that we are feeding in. We want to know how this program is going to behave. One example is Quicksort[1].

- Deterministic: does ok? for a random input but some inputs are terrible
- Randomized: is pretty good (and good enough) on all inputs.

Some applications of of randomized algorithms:

1. **Load Balancing**: Dynamically allocating resources to different servers, want to minimize worst-case load (see "Balls and Bins" section below).
2. **Resistance to Adversarial Inputs**: Randomized algorithms dilute the power of an adversary to manipulate the input in a way that causes the algorithm to perform poorly, a significant advantage in security-sensitive applications.
3. **Secret Keeping**: In cryptographic protocols, randomization is a key element in ensuring that secrets (like keys or passwords) cannot be easily guessed or reverse-engineered.
4. **Breaking Symmetry in Distributed Algorithms**: Randomization is often used to break symmetry in distributed systems, ensuring that no two nodes make the same decision simultaneously, which is critical in deadlock avoidance and resource allocation.

Motivating question for this lecture: Can we "randomize an input" to create a provably effective randomized algorithm?

[1]Should be prior knowledge, but reviewed in class.

Hashing: Introduction and Definitions

The formal setup for hashing is as follows:

- Our keys come from some large universe U (e.g, think of U as the set of all ascii strings of length at most 100.)
- The keys we actually care about are in some set $S \subseteq U$. This set can be assumed to be much smaller than U. For instance, perhaps S is the set of names of students in this class, which is certainly smaller than 128^{80}.
- The primary goal of hashing is to store these keys in a array of size n equipped with a **hash function** $h\colon U \to \{0, \ldots, n-1\}$ that can enable efficient operations on a hash table while minimizing collisions. These operations include LOOKUP(x), INSERT(x), and DELETE(x).

There are a few desired properties we want h to have. In particular,

- h is easily computable / doesn't take much space to describe. In our analysis today we will be viewing the time to compute $h(x)$ as a constant. However, it is worth remembering in the back of our heads that h shouldn't be too complicated, because that affects the overall runtime.
- # of collisions is small, since collisions affect the time to perform our operations.
- $n = \mathcal{O}(|S|)$; we would like a performant schema without causing our table size n to be much larger than the number of elements $|S|$.

We discussed why a deterministic approach wouldn't work, through the following scenario:

Student

You need to come up with a hash function that will hash any input in constant time.

Professor Randall

As your (evil) professor, no matter what hash function you give her, she can find some bad input for it. For example, if $h(x) \equiv x \pmod{n}$, Professor Randall would choose the inputs $i, i+n, i+2n \ldots, i+n^2$.

As we will soon see, the solution for this student is to not just choose one hash function, but take multiple and then choose at random.

Completely Random Hash Function

Back to our original discussion of hashing. As I'm sure many of you have already thought, "Why can't we just hash with a completely random hash function?" The issue with this idea is that there are $n^{|U|}$ possible hash functions. To represent *just one* of these hash functions, it would take $\log\left(n^{|U|}\right) = |U| \log(n)$ bits. This is completely impractical.

Regardless, it is a good primer for the next lecture to (assuming we do have a random hash function) check its validity (i.e. can it hash in constant time?)

Suppose we have a random hash function h. Then by definition, $\Pr_{h\sim\mathcal{H}}(h(x) = h(y)) = 1/n$. Now let X be the size of the hash bucket that x_i maps to. X is a random variable, with expected value

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{j=1}^{n} \Pr(h(x_i) = h(x_j)) \\
&= 1 + \sum_{j\neq i} \Pr(h(x_i) = h(x_j)) \\
&= 1 + \frac{n-1}{n} \leq 2
\end{aligned}$$

Thus the expected cost of hashing is a constant.

Universal Hash Functions

Note that mathematically, the only assumption we really made in our analysis above was that $\Pr_{h\sim\mathcal{H}}(h(x) = h(y)) = 1/n$. That got some folks[2] thinking **"can we replicate this behavior with a much simpler family of hash functions?"**

Definition: A family $\mathcal{H} = \{h_i\}_{i=1}^{k} = \{h_1, \ldots, h_k\}$ where $h_i \colon U \to \{0, \ldots, n-1\}$ for each i is **2-universal** if $\forall x, y \in U$ such that $x \neq y$ we have

$$\Pr_{h\sim\mathcal{H}}(h(x) = h(y)) \leq \frac{1}{n}$$

where $h \sim \mathcal{H}$ means that h is selected uniformly at random from $\mathcal{H}$.

Let's explore how a 2-universal hash family can achieve our goals with significantly reduced storage requirements.

Proposition. There exists a 2-universal family of hash functions whose members take values in n bins, take $\mathcal{O}(\log(N))$ bits to represent uniquely, and take $\mathcal{O}(1)$ time to evaluate.

Proof. We proceed constructively. Pick some $p \in [N, 2N]$[3]. Define

$$g_{a,b}(x) = ax + b \pmod{p}$$
$$h_{a,b}(x) = g_{a,b}(x) \pmod{n}$$

Now consider the set

$$\mathcal{H} = \{h_{a,b}(x) : a, b \in \{0, \ldots, p-1\}\}$$

A given h is parameterized by a and b. Each of these values are in the range $\mathcal{O}(N)$, so the number of bits needed to sample a hash function from this family is $\mathcal{O}(\log(N))$. We now claim this set is 2-universal.

Lemma. For every fixed two *distinct* $x, y \in \{0, \ldots, n-1\}$, the number of pairs a, b such that $h_{a,b}(x) = h_{a,b}(y)$ is at most $\frac{p(p-1)}{n}$.

[2] Carter and Wegmen, in 1978.

[3] Bertrand's postulate tells us this is always possible

Proof. Every collision $h_{a,b}(x) = h_{a,b}(y)$ stems from some pair $s, t, \in \{0, \ldots, p-1\}$ so that $s \equiv t \pmod{n}$, $g_{a,b}(x) = s$, and $g_{a,b}(y) = t$. So, let's count the number of such pairs.

Before we do any counting, notice that $g_{a,b}(x) \neq g_{a,b}(y)$, as it would imply that $p \mid a(x-y)$.

There are p choices for s. Once we fix s, there are $\lceil \frac{p}{n} \rceil$ values of $t \in \{0, \ldots, p-1\}$ with $s \equiv t \pmod{n}$; however, note that we must exclude the value $t = s$ for the reason above, so really we have $\lceil \frac{p}{n} \rceil - 1$ choices of t for every choice of s. Multiplying, we see that the total number of valid pairs (s, t) is $p(\lceil \frac{p}{n} \rceil - 1)$, which we can bound as

$$p\left\lceil \frac{p}{n} \right\rceil - 1 \leq p\left(\frac{p+n-1}{n}\right) - 1 = \frac{p(p-1)}{n}.$$

Next, for any given valid pair (s, t), we will count how many pairs (a, b) there are such that $g_{a,b}(x) \equiv s \pmod{p}$ and $g_{a,b}(y) \equiv t \pmod{p}$. In other words, we are asking how many values $a, b \in \{0, \ldots, p-1\}$ there are that satisfy the following system

$$\begin{aligned} ax + b &\equiv s \pmod{p} \\ ay + b &\equiv t \pmod{p} \end{aligned}$$

This is a system of 2 linear congruences in 2 unknowns. In this case the modulus is prime, so everything you know from linear algebra goes over to systems of linear congruences. (The reason is that $\mathbb{Z}_p$ is a **field**, for p prime, and linear algebra works fine over any field — not just $\mathbb{R}$ and $\mathbb{C}$.[4]) We can rewrite the above system as

$$\begin{bmatrix} x & 1 \\ y & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \equiv \begin{bmatrix} s \\ t \end{bmatrix} \pmod{p}.$$

The matrix $\left[\begin{smallmatrix} x & 1 \\ y & 1 \end{smallmatrix}\right]$ is invertible, so the system has a unique solution $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$. This means that for each valid pair (s, t), there is exactly one pair of values (a, b) such that $g_{a,b}(x) = s$ and $g_{a,b}(y) = t$. Consequently, the number of pairs (a, b) for which $h_{a,b}(x) = h_{a,b}(y)$ is at most $\frac{p(p-1)}{n}$.

[4] Looks like that 3406 class finally came in handy!

Since h is parameterized by a and b, there are $p(p-1)$ total functions in $\mathcal{H}$. We conclude that

$$\Pr_{h\sim\mathcal{H}}(h(x)=h(y)) = \frac{\#\{h\in\mathcal{H} : h(x)=h(y)\}}{p(p-1)} \leq \frac{\frac{p(p-1)}{n}}{p(p-1)} = \frac{1}{n}.$$

Balls and Bins

It is worth taking a little detour to ponder a classic problem (which actually relates back to our discussion above about load balancing!)

We have m balls and want to throw them into n bins. We throw them uniformly and independently. After all the balls are in the bins, what is the expected maximum number of balls in a single bin?

$m = 6$ balls into $n = 5$ bins

The answer, with high probability, is $\mathcal{O}(\log(n))$.[a] Note the base is not important, as

$$\mathcal{O}(\log_b(n)) = \mathcal{O}\left(\frac{\log_a(n)}{\log_a(b)}\right) = \mathcal{O}(\log_a(n))$$

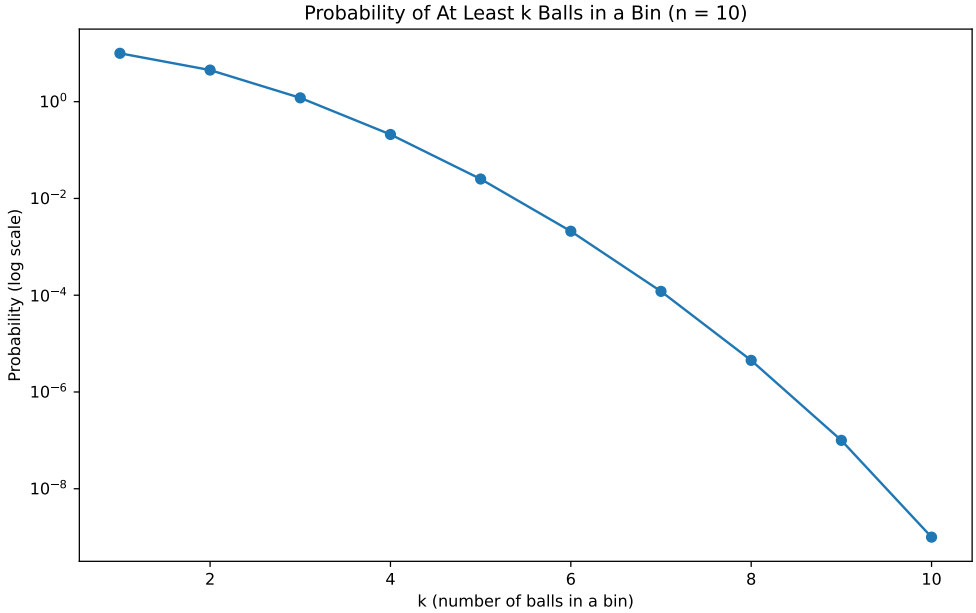
[a] The true answer is a little more nuanced, see the discussion here.

To illustrate the power of 2-universal hash function, suppose we have $m = n$

and our 'throw' is determined by a 2-universal hash function. Then

$$\begin{aligned}
\Pr(\text{Some bin gets} \geq k \text{ balls}) &\leq \sum_j \Pr(\text{Bin } j \text{ gets} \geq k \text{ balls}) \\
&\leq \sum_j \sum_{S \subseteq [n], |S|=k} \Pr(\text{balls } S \text{ all fall in bin } j) \\
&\leq \sum_j \binom{n}{k} \frac{1}{n^k} \\
&= n \binom{n}{k} \frac{1}{n^k}
\end{aligned}$$

Using the code in the Appendix, we plot this function for a fixed $n = 10$ below. Note the probability of a bin containing more than k balls decreases exponentially as k increases. Even for relatively small n, the probability becomes minuscule.



Further Discussion

This section is taken from the set of lecture notes here. It is a good resource if you are further interested in these topics.

There is another important question regarding hashing: if we fix the set S,

can we find a hash function h such that all lookups are constant-time? The answer is yes, and such hash functions are known as **perfect**. There was a big open question for some time about whether we could achieve perfect hashing with $\mathcal{O}(n)$ space, until it was finally achieved by exploiting the properties of universal hashing.

Finally, one more application of hashing I found interesting. Suppose we have a long sequence of items and we want to see how many different items are in the list. What is a good way of doing that?

One way is we can create a hash table, and then make a single pass through our sequence, for each element doing a lookup and then inserting if it is not in the table already. The number of distinct elements is just the number of inserts.

Now what if the list is really huge, so we don't have space to store them all, but we are OK with just an approximate answer. E.g., imagine we are a router and watching a lot of packets go by, and we want to see (roughly) how many different source IP addresses there are.

Here is a neat idea: say we have a hash function h that behaves like a random function, and let's think of $h(x)$ as a real number between 0 and 1. One thing we can do is just keep track of the minimum hash value produced so far (so we won't have a table at all). E.g., if keys are 3,10,3,3,12,10,12 and $h(3) = 0.4, h(10) = 0.2, h(12) = 0.7$, then we get 0.2.

The point is: if we pick N random numbers in $[0, 1]$, the expected value of the minimum is $1/(N + 1)$. Furthermore, there's a good chance it is fairly close (we can improve our estimate by running several hash functions and taking the median of the minimums).

Appendix

```
import matplotlib.pyplot as plt
from math import comb
import numpy as np

# Function to calculate the probability
def probability(k, n):
    return n * comb(n, k) * (1/n**k)

# Fixed number of bins/balls
n = 10

# Calculate probabilities for various values of k
k_values = np.arange(1, n+1)  # Considering values from 1 to n
probabilities = [probability(k, n) for k in k_values]

# Plotting the probabilities
plt.figure(figsize=(10, 6))
plt.plot(k_values, probabilities, marker='o')
plt.yscale('log')  # Set y-axis to log scale
plt.xlabel('k (number of balls in a bin)')
plt.ylabel('Probability (log scale)')
plt.title('Probability of At Least k Balls in a Bin  (n = 10)')
plt.grid(True)
plt.show()
```