princeton univ. F'15   cos 521: Advanced Algorithm Design

Lecture 1: Course Intro and Hashing

Lecturer: *Sanjeev Arora*   Scribe:*Sanjeev*

Algorithms are integral to computer science and every computer scientist (even as an undergrad) has designed several algorithms. So has many a physicist, electrical engineer, mathematician etc. This course is meant to be your one-stop shop to learn how to design a variety of algorithms. The operative word is "variety. "In other words you will avoid the blinders that one often sees in domain experts. A bayesian needs to see priors on the data before he can begin designing algorithms; an optimization expert needs to cast all problems as convex optimization; a systems designer has never seen any problem that cannot be solved by hashing. (OK, mostly kidding but there is some truth in these stereotypes.) These and more domain-specific ideas make an appearance in our course, but we will learn to not be wedded to any single approach.

The primary skill you will learn in this course is how to *analyse* algorithms: prove their correctness and their running time and any other relevant properties. Learning to analyse a variety of algorithms (designed by others) will let you design better algorithms later in life. I will try to fill the course with beautiful algorithms. Be prepared for frequent rose-smelling stops, in other words.

1 Difference between grad and undergrad algorithms

Undergrad algorithms is largely about algorithms discovered before 1990; grad algorithms is a lot about algorithms discovered since 1990. OK, I picked 1990 as an arbitrary cutoff. Maybe it is 1985, or 1995. What happened in 1990 that caused this change, you may ask? Nothing. It was no single event but just a gradual shift in the emphasis and goals of computer science as it became a more mature field.

In the first few decades of computer science, algorithms research was driven by the goal of designing basic components of a computer: operating systems, compilers, networks, etc. Other motivations were classical problems in discrete mathematics, operations research, graph theory. The algorithmic ideas that came out of these quests form the core of undergraduate course: data structures, graph traversal, string matching, parsing, network flows, etc. Starting around 1990 theoretical computer science broadened its horizons and started looking at new problems: algorithms for bioinformatics, algorithms and mechanism design for e-commerce, algorithms to understand big data or big networks. This changed algorithms research and the change is ongoing. One big change is that it is often unclear *what the algorithmic problem even is.* Identifying it is part of the challenge. Thus good *modeling* is important. This in turn is shaped by understanding what is *possible* (given our understanding of computational complexity) and what is *reasonable* given the limitations of the type of inputs we are given.

Some examples of this change:

The changing graph. In undergrad algorithms the graph is given and arbitrary (worst-case). In grad algorithms we are willing to look at where the graph came from (social network, computer vision etc.) since those properties may be germane to designing a good algorithm. (This is not a radical idea of course but we will see that formulating good graph models is not easy. This is why you see a lot of heuristic work in practice, without any mathematical proofs of correctness.)

Changing data structures: In undergrad algorithms the data structures were simple and often designed to hold data generated by other algorithms. A stack allows you to hold vertices during depth-first search traversal of a graph, or instances of a recursive call to a procedure. A heap is useful for sorting and searching.

But in the newer applications, data often comes from sources we don't control. Thus it may be noisy, or inexact, or both. It may be high dimensional. Thus something like heaps will not work, and we need more advanced data structures.

We will encounter the "curse of dimensionality"which constrains algorithm design for high-dimensional data.

Changing notion of input/output: Algorithms in your undergrad course have a simple input/output model. But increasingly we see a more nuanced interpretation of what the input is: datastreams (useful in analytics involving routers and webservers), online (sequence of requests), social network graphs, etc. And there is a corresponding subtlety in settling on what an appropriate output is, since we have to balance output quality with algorithmic efficiency. In fact, design of a suitable algorithm often goes hand in hand with understanding what kind of output is reasonable to hope for.

Type of analysis: In undergrad algorithms the algorithms were often *exact* and work on *all* (i.e., worst-case) inputs. In grad algorithms we are willing to relax these requirements.

2 Hashing: Preliminaries

Now we briefly study hashing, both because it is such a basic data structure, and because it is a good setting to develop some fluency in probability calculations.

Hashing can be thought of as a way to *rename* an address space. For instance, a router at the internet backbone may wish to have a searchable database of destination IP addresses of packets that are whizing by. An IP address is 128 bits, so the number of possible IP addresses is 2^{128}, which is too large to let us have a table indexed by IP addresses. Hashing allows us to rename each IP address by fewer bits. Furthermore, this renaming is done probabilistically, and the renaming scheme is decided in advance before we have seen the actual addresses. In other words, the scheme is *oblivious* to the actual addresses.

Formally, we want to store a subset S of a large universe U (where $|U| = 2^{128}$ in the above example). And $|S| = m$ is a relatively small subset. For each $x \in U$, we want to support 3 operations:

- *insert*(x). Insert x into S.
- *delete*(x). Delete x from S.
- *query*(x). Check whether $x \in S$.

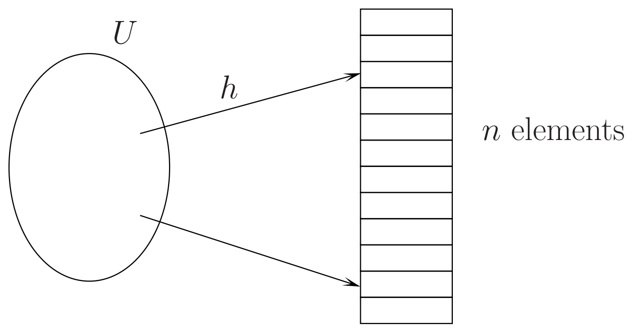

Figure 1: Hash table. x is placed in $T[h(x)]$.

A hash table can support all these 3 operations. We design a hash function

$$h : U \longrightarrow \{0, 1, \ldots, n-1\} \tag{1}$$

such that $x \in U$ is placed in $T[h(x)]$, where T is a table of size n.

Since $|U| \gg n$, multiple elements can be mapped into the same location in T, and we deal with these collisions by constructing a linked list at each location in the table.

One natural question to ask is: how long is the linked list at each location?

This can be analysed under two kinds of assumptions:

1. Assume the input is the random.
2. Assume the input is arbitrary, but the hash function is random.

Assumption 1 may not be valid for many applications.

Hashing is a concrete method towards Assumption 2. We designate a set of hash functions $\mathcal{H}$, and when it is time to hash S, we choose a random function $h \in \mathcal{H}$ and hope that on average we will achieve good performance for S. This is a frequent benefit of a randomized approach: no single hash function works well for every input, but the average hash function may be good enough.

3 Hash Functions

Say we have a family of hash functions $\mathcal{H}$, and for each $h \in \mathcal{H}$, $h : U \longrightarrow [n]$[1]. What do mean if we say these functions are random?

For any $x_1, x_2, \ldots, x_m \in S$ ($x_i \neq x_j$ when $i \neq j$), and any $a_1, a_2, \ldots, a_m \in [n]$, ideally a random $\mathcal{H}$ should satisfy:

[1]We use $[n]$ to denote the set $\{0, 1, \ldots, n-1\}$

- $\mathbf{Pr}_{h\in\mathcal{H}}[h(x_1) = a_1] = \frac{1}{n}$.
- $\mathbf{Pr}_{h\in\mathcal{H}}[h(x_1) = a_1 \wedge h(x_2) = a_2] = \frac{1}{n^2}$. Pairwise independence.
- $\mathbf{Pr}_{h\in\mathcal{H}}[h(x_1) = a_1 \wedge h(x_2) = a_2 \wedge \cdots \wedge h(x_k) = a_k] = \frac{1}{n^k}$. k-wise independence.
- $\mathbf{Pr}_{h\in\mathcal{H}}[h(x_1) = a_1 \wedge h(x_2) = a_2 \wedge \cdots \wedge h(x_m) = a_m] = \frac{1}{n^m}$. Full independence (note that $|U| = m$).

Generally speaking, we encounter a tradeoff. The more random $\mathcal{H}$ is, the greater the number of random bits needed to generate a function h from this class, and the higher the cost of computing h.

For example, if $\mathcal{H}$ is a fully random family, there are n^m possible h, since each of the m elements at S have n possible locations they can hash to. So we need $\log|\mathcal{H}| = m \log n$ bits to represent each hash function. Since m is usually very large, this is not practical.

But the advantage of a random hash function is that it ensures very few collisions with high probability. Let L_x be the length of the linked list containing x; this is just the number of elements with the same hash value as x. Let random variable

$$I_y = \begin{cases} 1 & \text{if } h(y) = h(x), \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

So $L_x = 1 + \sum_{y\in S; y\neq x} I_y$, and

$$E[L_x] = 1 + \sum_{y\in S; y\neq x} E[I_y] = 1 + \frac{m-1}{n} \tag{3}$$

Usually we choose $n > m$, so this expected length is less than 2. Later we will analyse this in more detail, asking how likely is L_x to exceed say 100.

The expectation calculation above doesn't need full independence; pairwise independence would actually suffice. This motivates the next idea.

4 2-Universal Hash Families

Definition 1 (Carter Wegman 1979) *Family $\mathcal{H}$ of hash functions is 2-universal if for any $x \neq y \in U$,*

$$\mathop{\mathbf{Pr}}_{h\in\mathcal{H}}[h(x) = h(y)] \leq \frac{1}{n} \tag{4}$$

Note that this property is even weaker than 2 independence.

We can design 2-universal hash families in the following way. Choose a prime $p \in \{|U|, \ldots, 2|U|\}$, and let

$$f_{a,b}(x) = ax + b \mod p \qquad (a, b \in [p], a \neq 0) \tag{5}$$

And let

$$h_{a,b}(x) = f_{a,b}(x) \mod n \tag{6}$$

Lemma 1
For any $x_1 \neq x_2$ and $s \neq t$, the following system

$$ax_1 + b = s \mod p \tag{7}$$
$$ax_2 + b = t \mod p \tag{8}$$

has exactly one solution.

Since $[p]$ constitutes a finite field, we have that $a = (x_1 - x_2)^{-1}(s - t)$ and $b = s - ax_1$. Since we have $p(p-1)$ different hash functions in $\mathcal{H}$ in this case,

$$\mathop{\mathbf{Pr}}_{h \in \mathcal{H}}[h(x_1) = s \wedge h(x_2) = t] = \frac{1}{p(p-1)} \tag{9}$$

Claim $\mathcal{H} = \{h_{a,b} : a, b \in [p] \wedge a \neq 0\}$ *is 2-universal.*

Proof: For any $x_1 \neq x_2$,

$$\begin{aligned}
&\mathbf{Pr}[h_{a,b}(x_1) = h_{a,b}(x_2)] && (10)\\
=&\sum_{s,t \in [p], s \neq t} \delta_{(s=t \mod n)} \mathbf{Pr}[f_{a,b}(x_1) = s \wedge f_{a,b}(x_2) = t] && (11)\\
=&\frac{1}{p(p-1)} \sum_{s,t \in [p], s \neq t} \delta_{(s=t \mod n)} && (12)\\
\leq&\frac{1}{p(p-1)} \frac{p(p-1)}{n} && (13)\\
=&\frac{1}{n} && (14)
\end{aligned}$$

where δ is the Dirac delta function. Equation (13) follows because for each $s \in [p]$, we have at most $(p-1)/n$ different t such that $s \neq t$ and $s = t \mod n$. □

Can we design a collision free hash table then? Say we have m elements, and the hash table is of size n. Since for any $x_1 \neq x_2$, $\mathbf{Pr}_h[h(x_1) = h(x_2)] \leq \frac{1}{n}$, the expected number of total collisions is just

$$E[\sum_{x_1 \neq x_2} h(x_1) = h(x_2)] = \sum_{x_1 \neq x_2} E[h(x_1) = h(x_2)] \leq \binom{m}{2} \frac{1}{n} \tag{15}$$

Let's pick $m \geq n^2$, then

$$E[\text{number of collisions}] \leq \frac{1}{2} \tag{16}$$

and so

$$\mathop{\mathbf{Pr}}_{h \in H}[\exists \text{ a collision}] \leq \frac{1}{2} \tag{17}$$

So if the size the hash table is large enough $m \geq n^2$, we can easily find a collision free hash functions. But in reality, such a large table is often unrealistic. We may use a two-layer hash table to avoid this problem.

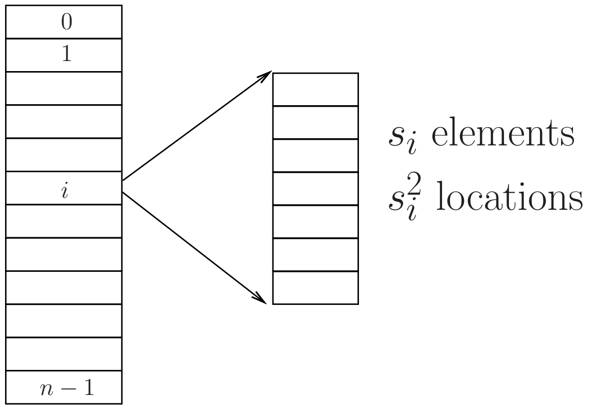

Figure 2: Two layer hash tables.

Specifically, let s_i denote the number of collisions at location i. If we can construct a second layer table of size s_i^2, we can easily find a collision-free hash table to store all the s_i elements. Thus the total size of the second-layer hash tables is $\sum_{i=0}^{m-1} s_i^2$.

Note that $\sum_{i=0}^{m-1} s_i(s_i-1)$ is just the number of collisions calculated in Equation (15), so

$$E[\sum_i s_i^2] = E[\sum_i s_i(s_i-1)] + E[\sum_i s_i] = \frac{m(m-1)}{n} + m \leq 2m \tag{18}$$

5 Load Balance

Now we think a bit about how large the linked lists (ie number of collisions) can get. Let us think for simplicity about hashing n keys in a hash table of size n. This is the famous balls-and-bins calculation, also called load balance problem. We have n balls and n bins, and we randomly put the balls into bins. Clearly, the expected number of balls in each bin is 1. But the maximum can be a fair bit higher.

For a given i,

$$\mathbf{Pr}[\text{bin}_i \text{ gets more than } k \text{ elements}] \leq \binom{n}{k} \cdot \frac{1}{n^k} \leq \frac{1}{k!} \tag{19}$$

(This uses the union bound, that the probability that any of the $\binom{n}{k}$ events happen is at most the sum of the their individual probabilities.) By Stirling's formula,

$$k! \sim \sqrt{2nk}(\frac{k}{e})^k \tag{20}$$

If we choose $k = O(\frac{\log n}{\log\log n})$, we can let $\frac{1}{k!} \leq \frac{1}{n^2}$. Then

$$\mathbf{Pr}[\exists \text{ a bin} \geq k \text{ balls}] \leq n \cdot \frac{1}{n^2} = \frac{1}{n} \tag{21}$$

So with probability larger than $1 - \frac{1}{n}$[2],

$$\max \text{load} \leq O(\frac{\log n}{\log \log n}) \tag{22}$$

Exercise: Show that with high probability the max load is indeed $\Omega(\log n / \log \log n)$.

5.1 Improved load balancing: Power of Two Choices

The above load balancing is not bad; no more than $O(\frac{\log n}{\log \log n})$ balls in a bin with high probability. Can we modify the method of throwing balls into bins to improve the load balancing? How about the method you use at the supermarket checkout: instead of going to a random checkout counter you try to go to the counter with the shortest queue? In the load balancing case (especially in distributed settings) this is computationally too expensive: one has to check all n queues. A much simpler version is the following: when the ball comes in, pick 2 random bins, and place the ball in the one that has fewer balls. Turns out this modified rule ensures that the maximal load drops to $O(\log \log n)$, which is a huge improvement. This called the *power of two choices.* The intuition why this helps is that even though the max load is $O(\log n / \log \log n)$, most bins have very few balls. For instance, at most 1/10th of the bins will have more than 10 balls. Thus when we pick two bins randomly, the chance is good that the ball goes to a bin with constant number of balls. Let us give a proof sketch.

For a ball b let us define

$$\textit{height}(b) = \text{load of its bin when } b \text{ was placed in it.}$$

Let $\rho(k, t)$ be the fraction of bins with at least k balls in it at time t.

Then the probability that the $t + 1$'th ball has height $k + 1$ is at most $\rho(k, t)^2$, since both of its choices must have had at least k balls when it arrived.

Noting that $\rho(2, t) \leq 1/2$ since the total number of balls is n, we use the above logic above to obtain:

$$E[\text{fraction of balls in } B \text{ w/ } \textit{height}(b) \geq i] \leq \left(\frac{1}{2}\right)^{2^{i-2}}.$$

In order to bound the size of largest the bin, we want to find the value of i such that the probability on the right is $\frac{1}{n}$. For this we need,

$$2^i \sim \Omega(\log n)$$
$$\Rightarrow i \sim \Omega(\log \log n).$$

This argument is hand-wavy and not 100% precise; for a full proof please see either various lecture notes around the web, or:

M. Mitzenmacher, A. Richa, and R. Sitaraman *The Power of Two Random Choices: A Survey of Techniques and Results.* Book chapter, in Handbook of Randomized Computing: volume 1, edited by P. Pardalos, S. Rajasekaran, and J. Rolim, pp. 255-312.

[2]this can be easily improve to $1 - \frac{1}{n^c}$ for any constant c

5.2 Cuckoo Hashing

Can we do the ultimate load balancing, and obtain a hashing scheme with $O(1)$ lookup time? Yes, provided we are willing to take a hit on insert operations.

A simple and practical way to do this is *cuckoo hashing*, invented by Pagh and Rodler in 2001. The name refers to the cuckoo's habit of putting its eggs in crows' nests —shifting its parental duties to others.

The idea is that we pick two hash functions h_1, h_2 instead of one. Thus each key x has two possible designated spots $h_1(x), h_2(x)$ to go to. When key x arrives, it randomly picks one of these two, say $h_1(x)$. If $h_1(x)$ happens to be occupied by some other key y, then y gets kicked out and x takes this spot. Now y has to go to its *other* designated location. If that happens to be occupied, then its occupant is kicked out and and forced to go to *its* other designated location. And so on. Thus insert can take an unbounded amount of time, though it is possible to prove probabilistic bounds on the running time, as we will explore in the homeworks.

Notice however that look up takes $O(1)$ time: just check both of the designated locations, and if the key is not in either, return `fail`.

This basic hashing idea has (inevitably) many more variants, as a web search will show.

BIBLIOGRAPHY

Rasmus Pagh, and Flemming Eriche Rodler (2001). "Cuckoo Hashing".*Algorithms ESA 2001. Lecture Notes in Computer Science* 2161. pp. 121-133.