COMP 480/580 — Probabilistic Algorithms and Data Structures September 7, 2025

Lecture 2

Lecturer: Anshumali Shrivastava *Scribe By: Dev S., Lajvi B., Saurav Kr. G.*

Pseudorandom Numbers, Universal Hashing, and Collision Resolution

1 Pseudorandom Number Generators

1.1 True Randomness vs. Pseudorandomness

Random numbers are fundamental to probabilistic algorithms. A truly random sequence may be generated by physical phenomena (flipping coins, electronic noise, quantum effects, etc.). However, such "true" randomness is often impractical to obtain in software at high speed. Instead, we use *pseudorandom number generators* (PRNGs), which produce a deterministic sequence of numbers that "looks" random. A PRNG is initialized with a *seed* value, and thereafter it produces a sequence of outputs by a fixed algorithm. Because the process is deterministic, the same seed will always yield the same sequence. The challenge is to design PRNGs that produce sequences with statistical properties indistinguishable from true randomness for algorithmic purposes.

Desirable Properties: A good PRNG should have a long period (the sequence length before it repeats), and the output should be *uniformly distributed* in the expected range. Ideally, it should pass various statistical tests for randomness. For cryptographic applications, even stronger unpredictability properties are required (a cryptographically secure PRNG must withstand attempts to distinguish its output from a truly random sequence without knowledge of the seed). In our context of probabilistic algorithms, we primarily require that the PRNG's outputs are uniform and independent enough to simulate random choices.

1.2 The Middle-Square Method

One of the earliest PRNGs, suggested by John von Neumann in 1946, is the **middle-square method**. The algorithm is simple: start with an n-digit seed. To generate the next number, square the current value (producing up to $2n$ digits), then extract the middle n digits of the result as the next output and also as the new seed. For example, with a 4-digit seed 1111, squaring yields $1111^2 = 1234321$, written as 8 digits 01234321. Taking the middle 4 digits gives 2343 as the next pseudorandom number. Repeating the process would then square 2343 to get the next number, and so on.

While simple, the middle-square method has serious flaws. Many sequences generated this way tend to degenerate quickly. For instance, some seeds enter short cycles or even reach 0, after which they will output only zeros forever (e.g. starting from seed 0000 obviously yields 0000 every time). In practice, most seeds eventually fall into a loop or repeat within a relatively small number of steps. Von Neumann was aware of these issues but used the method as a stopgap given the limited computing resources of the time. Today, the middle-square method is mostly of historical interest, having been supplanted by more robust generators.

Exercise: Try the middle-square method with a few different 4-digit seeds (for example, 1000, 3792, 6250). Observe how quickly the sequence repeats or reaches a trivial cycle. Can you find a seed (other than 0000) that causes an immediate repetition in one step?

Interestingly, a recent revival of the middle-square idea combines it with an additional sequence to improve its quality. This brings us to the concept of **Weyl sequences**.

1.3 Weyl Sequences and Linear Congruential Generators

A *Weyl sequence* is a sequence defined by repeated addition of a constant modulo some number. Specifically, choose an integer increment k and modulus m; start from an initial value X_0 and generate $X_{n+1} = (X_n + k) \bmod m$. If k and m are relatively prime (i.e. $\gcd(k, m) = 1$), then this sequence cycles through all m possible residues before repeating, and the values are *equidistributed* modulo m. In other words, over a full cycle, each value in $0, 1, \ldots, m-1$ appears exactly once, so the sequence "fills" the residue space uniformly. This is a direct consequence of number theory: adding a fixed increment k (with $\gcd(k, m) = 1$) permutes the residues mod m.

The Weyl sequence alone produces a simple linear progression modulo m, which is not very random (it's just an arithmetic progression). However, it can be used to improve other generators. For example, one proposal called the **Middle Square Weyl Sequence** (MSWS) generator combines the middle-square method with a Weyl sequence increment to avoid the pitfalls of the middle-square alone. In MSWS, each iteration squares the state (like the middle-square) and also adds a constant (like a Weyl sequence) before taking the middle bits. The added constant helps ensure the state continues to evolve even if the squared part becomes 0 or stagnant. The C code snippet from the slides:

```
d += 362437;
```

illustrates adding a constant increment each time (here 362437). In that example, 362437 is chosen as an odd integer seed/increment. The reason it must be odd is to ensure it is coprime with 2^{32} (assuming $m = 2^{32}$ for a 32-bit generator). If it were even, then the sequence $d \leftarrow d + 362437 \pmod{2^{32}}$ would only ever produce numbers of the same parity as the seed and would actually cycle through only half of the 2^{32} values (skipping all odd outputs) – not fully utilizing the 32-bit range. With 362437 (odd), the additive sequence has a full period of 2^{32}, achieving equidistribution of outputs in the 32-bit space. More generally, with modulus 2^n, using an *odd* increment yields a full-period additive (Weyl) sequence that visits every residue class exactly once.

The most widely used class of PRNGs in practice (for non-cryptographic purposes) is the **linear congruential generator (LCG)**. An LCG produces a sequence according to the recurrence:

$$X_{n+1} = (a \cdot X_n + c) \bmod m,$$

with multiplier a, increment c, modulus m, and initial seed X_0. The Weyl sequence is essentially a special case of an LCG with $a = 1$ (often called an *additive congruential* generator). More generally, by choosing appropriate a, c, m, one can obtain a much longer period and better statistical properties. In fact, a well-chosen LCG can have a period equal to m, meaning it cycles through all residues before repeating. For an LCG to have full period m, it must satisfy the standard Hull-Dobell conditions:

- c is coprime with m (ensuring the additive part can reach all residues).
- $a - 1$ is divisible by all prime factors of m.

- If m is a multiple of 4, then $a - 1$ is divisible by 4.

We won't derive these conditions here, but intuitively the second and third conditions ensure the multiplier a has certain residue cycle properties relative to m. For a *mixed* LCG $X_{n+1} = (aX_n + c) \bmod m$ to have full period m, the Hull–Dobell conditions must hold: $\gcd(c, m) = 1$; every prime $q \mid m$ divides $(a - 1)$; and if $4 \mid m$ then $4 \mid (a - 1)$. In particular, when m is prime these conditions force $a \equiv 1 \pmod{m}$ (not that a be a generator), so taking $a = 1$ with $c \neq 0$ achieves period m.

Example. A widely cited LCG (popularized by *Numerical Recipes*) uses $m = 2^{32}$, $a = 1664525$, $c = 1013904223$ and achieves full period under Hull–Dobell. By contrast, many classic C `rand()` implementations historically used different parameters, e.g., $a = 1103515245$, $c = 12345$, $m = 2^{31}$; However, not all LCGs are good. A notorious bad example was `RANDU` from the 1960s, which used $m = 2^{31}$, $a = 65539$, $c = 0$. Its choice of a caused the sequence to fall into planes in 3D space, failing spectral tests (its choice $a = 65539 = 2^{16} + 3$ with $m = 2^{31}$ and $c = 0$ led to severe correlations, notoriously, triples fall on 15 parallel planes in 3D).

Even with a full-period, LCG outputs can have subtle statistical shortcomings. For instance, using an increment c ensures full coverage of residues (Weyl sequence property), but the sequence still has a linear structure. For this reason, modern general-purpose PRNGs often use more complex algorithms (like **Mersenne Twister** (MT19937), which has period $2^{19937} - 1$ and excellent equidistribution in high dimensions,[1] or cryptographic generators based on ciphers). Nonetheless, LCGs remain popular for their simplicity and speed.

Exercise: (i) Prove that if $\gcd(k, m) = 1$, the sequence $0, k, 2k, 3k, \ldots \pmod{m}$ is uniformly distributed over $\{0, \ldots, m - 1\}$. (Hint: show that the first m terms are a permutation of all residues mod m.) (ii) Using the above, explain why adding an odd constant in a modulus 2^n PRNG guarantees hitting all values, whereas adding an even constant would not.

2 Hash Functions and Universal Hashing

2.1 Hashing Basics

Hashing is a technique to map a potentially large universe of keys (e.g. strings, integers, etc.) into a fixed range of indices (slot numbers for a table). A **hash function** $h : U \to \{0, 1, \ldots, m - 1\}$ takes an input key and computes an index in a table of size m. We store the key (and its associated data) at that index. Ideally, h should scatter keys uniformly across the m slots. However, since $|U|$ (universe size) is typically much larger than m, *collisions* are inevitable: two distinct keys $x \neq y$ may have $h(x) = h(y)$. Designing good hash functions and strategies to handle collisions is critical for efficient hashing.

A simple and common choice is the **division method**: pick a table size m and let

$$h(k) = k \bmod m.$$

This is easy to compute (one modulo operation). For example, if $m = 10$ and the key universe is integers, $h(k) = k \mod 10$ just gives the last decimal digit of k. In choosing m, one should avoid certain values that could lead to non-uniform distributions. Powers of 2 are often poor choices for m because $k \mod 2^p$ is just the lowest p bits of k. If, say, many keys share the same lower bits (e.g. all keys are even, or all are multiples of 100), they would all hash to a small subset of slots. A prime m not close to a power of 2 is usually recommended. As an

[1] MT19937 is *not* cryptographically secure; use a CSPRNG for adversarial settings.

example, if keys are character strings interpreted in base 128, a prime m like 10,007 is preferable to $m = 8192$ (which is 2^{13}) to avoid only hashing on a few bits of the sum or value. Another method, the **multiplication method**, avoids picking bad m by multiplying k by a constant $A \in (0, 1)$ and extracting the fractional part times m; this can distribute keys evenly for any m (Knuth recommends $A \approx (\sqrt{5} - 1)/2$), but it involves floating-point multiplication which can be slower.

Computing a modulo can be slow on some hardware, especially if m is not a power of 2. An optimization (often a discussion point in class) is that if m is a power of 2, say $m = 2^p$, then $h(k) = k \bmod 2^p$ can be computed by taking the p lowest-order bits of k (essentially a bitmask). This eliminates the expensive division operation. *However*, as noted above, using $m = 2^p$ can harm the uniformity of h if the key values exhibit patterns in their lower bits. **Class Exercise (Mod operation is slow):** Consider scenarios where using $m = 2^p$ might be acceptable or beneficial. For example, if we know our keys are uniformly distributed 32-bit integers, $m = 2^{10} = 1024$ might work fine and allow using a bitmask. On the other hand, if keys are, say, all multiples of 1024 (such as addresses aligned to 1024 bytes), then k mod 1024 is always 0, causing a disastrous hash distribution. This exercise highlights the trade-off between computational efficiency and hash quality.

In summary, a good hash function should balance *speed* and *uniformity*. In practice, for general keys (like strings), one might combine operations (bit shifts, multiplications, etc.) to mix the bits of the key thoroughly, then take mod m. For example, a simple string hash might do:

$$h(s) = \Big(\sum_i c_i \cdot 128^i \Big) \bmod m,$$

where c_i are character codes. This treats the string as a number in base 128. A potential issue arises if m divides $128^k - 1$ for some k, which can induce structured collisions among *related* strings; choosing a prime m not near a power of 128 mitigates this. (Note: polynomial/base-128 hashes do *not* collapse arbitrary anagrams, since positions change exponents.)

A Simple Interview Question: "How would you design a hash function for telephone numbers?" One answer: take a telephone number (which is essentially a large integer) and mod by a suitable prime. A better answer might note patterns in phone numbers (like area codes) and suggest multiplying parts or using a base representation to ensure all digits influence the hash. The goal is to demonstrate understanding of making a hash function that distributes real-world data uniformly.

2.2 Universal Hashing

One problem with choosing a fixed hash function h is that a clever adversary could deliberately supply many keys that collide (e.g., all keys mapping to the same slot), causing worst-case performance to degrade. **Universal hashing** is a strategy, introduced by Carter and Wegman (1977), to thwart such adversaries by using randomness in the choice of h. Instead of a single static hash function, we design a *family* $\mathcal{H}$ of hash functions with a nice property, and then select one h from $\mathcal{H}$ at random at the start of the algorithm. This way, an adversary doesn't know exactly which hash function is in use, and for a well-designed family, the chance of heavy collisions can be made low.

Formally, a family of functions $\mathcal{H} = \{h : U \to [m]\}$ is **universal** if for any two distinct keys $x \neq y$ in U, the probability (over a random choice of h from $\mathcal{H}$) that x and y collide is at most

$1/m$. Equivalently:

$$\forall x \neq y, \qquad \Pr_{h \in \mathcal{H}}[\, h(x) = h(y)\,] \;\leq\; \frac{1}{m}.$$

In a perfectly random function, the collision probability would be exactly $1/m$, so a universal family achieves (up to constant factors) the ideal collision chance. Sometimes the term **2-universal** (or **pairwise independent**) is used to mean that for any two distinct keys x, y, not only is $\Pr[h(x) = h(y)] = 1/m$, but the two hash values $h(x)$ and $h(y)$ are independent random variables. In many contexts, "universal" as defined above is sufficient, and it's the definition we will use here.

The main benefit of universal hashing is a provable guarantee on expected performance. Suppose we build a hash table of size m and choose $h \in \mathcal{H}$ uniformly at random from a universal family. Insert n keys into the table (we'll discuss collision resolution soon). For any fixed key x, the expected number of other keys that collide with it is at most $(n-1)/m$. This is because each of the $n-1$ other keys y has $\Pr(h(y) = h(x)) \leq 1/m$, and by linearity of expectation:

$$E[\text{collisions on } x] = \sum_{y \neq x} \Pr[h(y) = h(x)] \leq \frac{n-1}{m}.$$

So the expected *chain length* or cluster size for x (including x itself, in case x is in the table) is at most $1 + \frac{n-1}{m}$. If n is on the order of m (load factor $\alpha = n/m$ is a constant), this is $O(1)$. We emphasize this is an expectation taken over the random choice of h (which in turn randomizes the placement of keys). No matter how adversarial the key set is, a random $h \in \mathcal{H}$ will, on average, scatter them enough to keep collisions low.

Conclusion: With universal hashing, the expected time for a search (or insert or delete) in a hash table is $\Theta(1)$, assuming the load factor $n/m = O(1)$. This expected time is *input distribution independent*, meaning it holds even in the worst-case scenario for the keys, as long as our hash function was chosen at random from a universal family.

Designing a universal family $\mathcal{H}$ is an interesting task. A classic and widely used construction for hashing integers is as follows: - Fix a prime number p larger than the maximum possible key value (or at least larger than $|U|$, the universe size). For example, if keys are 32-bit, one might choose a 32-bit prime like $p = 2^{31} - 1 = 2147483647$. - Define $\mathcal{H} = \{h_{a,b} : h_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod m\}$, where a and b are integers. We let a range over $\{1, 2, \ldots, p-1\}$ (i.e. $a \neq 0 \pmod p$) and b range over $\{0, 1, \ldots, p-1\}$. This gives $|\mathcal{H}| = (p-1) \cdot p$ possible hash functions. - To pick a random h from $\mathcal{H}$, we just choose a and b uniformly at random from those ranges and use $h_{a,b}$.

It can be shown that the family $h_{a,b}$ is *universal* for $[m]$ when a is uniform over $1, \ldots, p-1$ and b over $0, \ldots, p-1$. Moreover, the map $\tilde{h}_{a,b}(x) = (ax + b) \bmod p$ is pairwise independent over $[p]$; after the final reduction $\bmod, m$ we retain universality $\Pr[h(x) = h(y)] \leq 1/m$ for $x \neq y$, but full pairwise independence need not hold unless $m = p$.

Other universal families exist for different key types (e.g., for strings, one can interpret the string as a polynomial and choose a random mod p evaluation). The important takeaway is that by randomizing the hash function choice, we obtain guarantees on collision probability. Many modern hash table implementations use a fixed hash but rely on randomness in keys or assume average-case distributions. However, truly adversary-resistant implementations (e.g., some hash tables in cryptographic contexts or in programming languages for security) do employ random hashing under the hood.

Exercise: Prove that the family $H = \{h_{a,b}\}$ defined above is universal, by showing for any fixed $x \neq y$ and any $i, j \in [m]$, $\Pr[h_{a,b}(x) = i \wedge h_{a,b}(y) = j] = 1/m^2$. You will need to use the primeness of p in the argument.

3 Examples for Hashing Methods

3.1 Middle-Square Method

Start with seed 1111. Then $1111^2 = 1234321$ (pad to 8 digits: 01234321). Take middle 4 digits: 2343.

3.2 Modular Sequence Method

Let $p = 7$ and $k = 3$. Sequence: $3, 6, 2, 5, 1, 4, 0 \mod 7$.

3.3 Linear Congruential Generator (LCG)

Parameters: $a = 5$, $c = 3$, $m = 16$, seed $x_0 = 7$. Then:

$$\begin{aligned} x_1 &= (5 \cdot 7 + 3) \mod 16 = 38 \mod 16 = 6 \\ x_2 &= (5 \cdot 6 + 3) \mod 16 = 33 \mod 16 = 1 \\ x_3 &= (5 \cdot 1 + 3) \mod 16 = 8 \mod 16 = 8 \end{aligned}$$

4 Proof of 2-Universal Hashing

We define $h_{a,b}(x) = ((a \cdot x + b) \mod p) \mod m$ where $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$. Let $x \neq y$ be two keys. We want to show: $\Pr[h_{a,b}(x) = h_{a,b}(y)] \leq \frac{1}{m}$. Because $a(x - y) \not\equiv 0 \mod p$ (as $x \neq y$ and $a \neq 0$), the expression $(a(x - y))$ ranges over all nonzero values mod p uniformly. So for any fixed i, $\Pr[h_{a,b}(x) = h_{a,b}(y) = i] = \frac{1}{m^2}$, implying collision probability is bounded by $1/m$.

5 Generalization to k-wise Independence

A hash family is k-wise independent if for any k distinct keys, their hash values are mutually independent. For example, with 3-wise independence, we have: $\Pr[h(x_1) = i_1 \wedge h(x_2) = i_2 \wedge h(x_3) = i_3] = \frac{1}{m^3}$ for all $x_1 \neq x_2 \neq x_3$ and $i_1, i_2, i_3 \in [m]$. This stronger notion can be constructed using degree-$(k-1)$ polynomials over finite fields.