17. Randomized Algorithms: Why Randomness Helps

Phase 6: Real-World Applications & Limitations
⏱ ~40 minutes | 🎲 Deterministic Algorithms Can Fail | 🚀 Randomness as a Feature

---

What Problem This Solves

You encounter:

- Hash table collisions in adversarial inputs (DOS attack)
- Quicksort worst-case $O(n^2)$ behavior
- Load balancing across servers causing hotspots
- Random sampling for analyzing huge datasets
- "Why is randomness ever good? Shouldn't we be deterministic?"
- Distributed systems where randomness reduces collisions

Without randomized algorithm intuition, you either avoid randomness ("I want reproducible results!") or use it arbitrarily. You're vulnerable to adversarial inputs that exploit deterministic patterns.

With randomized thinking, you leverage randomness to break worst-case inputs, balance loads, and solve hard problems with good expected behavior.

Intuition & Mental Model

The Core Insight: Randomness Breaks Patterns

```
Adversarial Scenario:
- Attacker sends hash collisions → hash table slow
- Adversary orders input → quicksort degrades
- Requests concentrate on one server → that server overloaded

Randomized Solution:
- Hash function randomized → attacker can't predict collisions
- Pivot selection randomized → unlikely to always pick bad pivots
- Request routing randomized → load spreads naturally
- Attacker's advantage disappears!
```

Mental Model: The Adversary-Free Future

```
Deterministic:  Input adversary can exploit → Worst case often achievable
Randomized:     Adversary can't know random choices → Can't force worst case
                Only exponentially unlikely inputs fail
```

Core Concepts

1. Randomized Quicksort

```
function quicksortDeterministic(arr, low = 0, high = arr.length - 1) {
  // Deterministic pivot (e.g., first element)
  // Adversary sends reversed array → O(n²) behavior

  if (low >= high) return;

  const pivot = arr[low];  // Naive choice (vulnerable!)
  let i = low + 1, j = high;

  while (i <= j) {
    while (i <= j && arr[i] <= pivot) i++;
    while (i <= j && arr[j] > pivot) j--;
    if (i < j) [arr[i], arr[j]] = [arr[j], arr[i]];
  }

  [arr[low], arr[j]] = [arr[j], arr[low]];

  quicksortDeterministic(arr, low, j - 1);
  quicksortDeterministic(arr, j + 1, high);
}

function quicksortRandomized(arr, low = 0, high = arr.length - 1) {
  // Randomized pivot (pick random element)
  // No adversarial input → O(n log n) expected behavior

  if (low >= high) return;

  // Pick random pivot
  const randomIdx = low + Math.floor(Math.random() * (high - low + 1));
  const pivot = arr[randomIdx];
  [arr[low], arr[randomIdx]] = [arr[randomIdx], arr[low]];  // Swap to front

  let i = low + 1, j = high;

  while (i <= j) {
    while (i <= j && arr[i] <= pivot) i++;
    while (i <= j && arr[j] > pivot) j--;
    if (i < j) [arr[i], arr[j]] = [arr[j], arr[i]];
  }

  [arr[low], arr[j]] = [arr[j], arr[low]];

  quicksortRandomized(arr, low, j - 1);
  quicksortRandomized(arr, j + 1, high);
}

// Adversary's attack: Sorted/reverse array
const attack = Array.from({ length: 1000 }, (_, i) => i).reverse();

console.time('Deterministic');
quicksortDeterministic([...attack]);
```

```
console.timeEnd('Deterministic');  // ~100ms (quadratic!)

console.time('Randomized');
quicksortRandomized([...attack]);
console.timeEnd('Randomized');  // ~2ms (linear-ish!)
```

Why It Works:

```
Deterministic pivot: Adversary picks input to make it worst
Randomized pivot: Pivot likely good (random choice)
                Expected: O(n log n) even against adversary
                Probability of bad pivot: exponentially small
```

2. Hashing with Randomization

```
function hashTableAttack() {
  // Deterministic hash: h(x) = x % tableSize

  function simpleHash(x) {
    return x % 1000;  // Known pattern
  }

  // Attacker sends: 1000, 2000, 3000, ...
  // All hash to 0 → Hash table becomes linked list → O(n) lookup!

  return {
    attack: 'Send keys that hash to same bucket: 1000, 2000, 3000',
    result: 'All collide → O(1) becomes O(n) (table degrades)'
  };
}

function randomizedHashing() {
  // Universal hashing: Random hash function from family

  class UniversalHash {
    constructor(tableSize = 1000) {
      this.size = tableSize;
      this.prime = 1000000007;
      this.a = 1 + Math.floor(Math.random() * (this.prime - 1));
      this.b = Math.floor(Math.random() * this.prime);
    }

    hash(x) {
      // h(x) = (ax + b) mod prime mod tableSize
      return ((this.a * x + this.b) % this.prime) % this.size;
    }
  }

  // Same attack: 1000, 2000, 3000, ...
  const h = new UniversalHash(1000);
```

```
  console.log([1000, 2000, 3000].map(x => h.hash(x)));
  // Different hashes (random function chosen at init)
  // Attack fails because attacker doesn't know a, b

  // Key insight: Even if adversary knows inputs, hash function is random
  // Expected collisions = O(1) per operation
}
```

3. Load Balancing with Randomization

```
function loadBalancing() {
  // Problem: How to distribute requests across servers?

  class DeterministicBalancer {
    constructor(serverCount = 10) {
      this.servers = Array(serverCount).fill(0);
      this.nextServer = 0;
    }

    route(request) {
      // Round-robin (deterministic)
      const server = this.nextServer % this.servers.length;
      this.servers[server]++;
      this.nextServer++;
      return server;
    }
  }

  class RandomizedBalancer {
    constructor(serverCount = 10) {
      this.servers = Array(serverCount).fill(0);
    }

    route(request) {
      // Pick random server (with probability proportional to capacity)
      // Or simpler: just pick random
      const server = Math.floor(Math.random() * this.servers.length);
      this.servers[server]++;
      return server;
    }
  }

  // Adversary sends: requests that align with round-robin
  // Knows pattern → Can predict which server gets each request

  const det = new DeterministicBalancer(5);
  for (let i = 0; i < 20; i++) {
    det.route(i);
  }
  console.log('Deterministic load:', det.servers);  // [4, 4, 4, 4, 4]
  // Looks balanced! But if adversary sends bursts at certain times...
```

```
  const rand = new RandomizedBalancer(5);
  for (let i = 0; i < 20; i++) {
    rand.route(i);
  }
  console.log('Randomized load:', rand.servers);  // [3, 4, 4, 5, 4]
  // Also balanced, but adversary can't exploit pattern

  // Worst case for randomized:
  // Probability all 20 go to one server: (1/5)^20 ≈ 10^-14
}
```

4. Las Vegas vs Monte Carlo Algorithms

```
function algorithmClasses() {
  // Las Vegas: Always correct, randomized time
  // Monte Carlo: Randomized correctness, fixed time

  // Las Vegas: Randomized Quicksort
  // Always produces sorted array, but time varies
  function lasVegas(arr) {
    // Guaranteed correct, expected O(n log n)
    quicksortRandomized([...arr]);
    return arr;  // Correct!
  }

  // Monte Carlo: Randomized Primality Test
  // Fixed time, small probability of error
  function millerRabin(n, k = 40) {
    // Runs in O(k log n)
    // Returns "probably prime" (error prob < 2^-k)

    if (n < 2) return false;
    if (n === 2 || n === 3) return true;
    if (n % 2 === 0) return false;

    let r = 0, d = n - 1;
    while (d % 2 === 0) {
      r++;
      d /= 2;
    }

    for (let i = 0; i < k; i++) {
      const a = 2 + Math.floor(Math.random() * (n - 4));
      let x = Math.pow(a, d) % n;  // Simplified

      if (x === 1 || x === n - 1) continue;

      let composite = true;
      for (let j = 0; j < r - 1; j++) {
        x = (x * x) % n;
```

```
        if (x === n - 1) {
          composite = false;
          break;
        }
      }

      if (composite) return false;
    }

    return true;  // "Probably prime" (error < 2^-40)
  }

  return {
    lasVegas: {
      correctness: 'Always correct',
      time: 'Randomized (expected O(n log n), worst O(n²) with prob 2^-n)',
      example: 'Quicksort, randomized search'
    },
    monteCarlo: {
      correctness: 'Probably correct (bounded error)',
      time: 'Deterministic O(k log n)',
      example: 'Primality testing, randomized approximation'
    }
  };
}
```

5. Randomized Data Structures

```
function skipList() {
  // Skip List: Like linked list but with "express lanes"
  // Search O(log n) expected time with randomization

  class SkipNode {
    constructor(value, level) {
      this.value = value;
      this.next = Array(level).fill(null);
    }
  }

  class SkipList {
    constructor(maxLevel = 16, p = 0.5) {
      this.maxLevel = maxLevel;
      this.p = p;  // Probability of promoting to next level
      this.head = new SkipNode(-Infinity, maxLevel);
      this.level = 0;
    }

    randomLevel() {
      let level = 0;
      while (Math.random() < this.p && level < this.maxLevel - 1) {
        level++;
```

```
    }
    return level;
  }

  search(value) {
    let current = this.head;

    for (let i = this.level; i >= 0; i--) {
      while (current.next[i] && current.next[i].value < value) {
        current = current.next[i];
      }
    }

    current = current.next[0];
    return current && current.value === value;
  }

  insert(value) {
    const newLevel = this.randomLevel();
    const newNode = new SkipNode(value, newLevel + 1);

    // Simplified insert logic
    let current = this.head;
    const updates = Array(newLevel + 1).fill(this.head);

    for (let i = Math.min(newLevel, this.level); i >= 0; i--) {
      while (current.next[i] && current.next[i].value < value) {
        current = current.next[i];
      }
      updates[i] = current;
    }

    for (let i = 0; i <= newLevel; i++) {
      newNode.next[i] = updates[i].next[i];
      updates[i].next[i] = newNode;
    }

    if (newLevel > this.level) {
      this.level = newLevel;
    }
  }
}

const skip = new SkipList();
[3, 7, 1, 9, 2, 5].forEach(v => skip.insert(v));

console.log(skip.search(7));   // true
console.log(skip.search(10));  // false

// Expected search time: O(log n)
```

```
  // Much simpler than balanced trees!
}
```

6. Shuffle and Sampling

```
function randomSampling() {
  // Fisher-Yates Shuffle: Uniformly random permutation

  function shuffle(arr) {
    for (let i = arr.length - 1; i > 0; i--) {
      const j = Math.floor(Math.random() * (i + 1));
      [arr[i], arr[j]] = [arr[j], arr[i]];
    }
    return arr;
  }

  // Reservoir Sampling: Sample k items from stream of unknown length

  function reservoirSample(stream, k) {
    const reservoir = stream.slice(0, k);

    for (let i = k; i < stream.length; i++) {
      const j = Math.floor(Math.random() * (i + 1));
      if (j < k) {
        reservoir[j] = stream[i];
      }
    }

    return reservoir;
  }

  const data = Array.from({ length: 10000 }, (_, i) => i);

  // Shuffle
  console.log(shuffle([1, 2, 3, 4, 5]));  // Random order each time

  // Sample 10 items from 10000 uniformly
  console.log(reservoirSample(data, 10).length);  // 10

  return {
    shuffleTime: 'O(n)',
    reservoirTime: 'O(stream length)',
    uniformity: 'Each permutation equally likely'
  };
}
```

Software Engineering Connections

1. Hash Table Implementation

```
function cryptographicHashTable() {
  // Use a randomized seed for hash function

  const seed = Math.random() * 1e9;

  function hash(x) {
    return ((x * 2654435761 + seed) >>> 0) % 1000;  // Multiply-shift hash
  }

  // Same malicious input: 1000, 2000, 3000, ...
  // Hash function different on each run → Adversary can't exploit
  // Expected collision rate: 1/tableSize regardless of input
}
```

2. Distributed Systems: Random Backoff

```
function exponentialBackoffRandomized() {
  // When retrying failed operation, add randomness to avoid thundering herd

  async function retryWithBackoff(operation, maxAttempts = 5) {
    for (let attempt = 0; attempt < maxAttempts; attempt++) {
      try {
        return await operation();
      } catch (error) {
        if (attempt === maxAttempts - 1) throw error;

        // Exponential backoff with jitter
        const baseDelay = Math.pow(2, attempt) * 1000;  // 1s, 2s, 4s, 8s, 16s
        const jitter = Math.random() * baseDelay;
        const delay = baseDelay + jitter;

        console.log(`Retry ${attempt + 1} after ${delay.toFixed(0)}ms`);
        await new Promise(resolve => setTimeout(resolve, delay));
      }
    }
  }

  // Without jitter: All clients retry at T=1s, T=2s, T=4s → Thundering herd
  // With jitter: Retries spread out randomly → Smooth load
}
```

3. A/B Testing: Random Assignment

```
function randomABTest() {
  function getUserGroup(userId, testId) {
    // Hash user + test → 0-1 range → Assign to A or B
    const hash = hashString(userId + testId);
    return hash < 0.5 ? 'A' : 'B';
  }
```

```
  // Key property: Same user always gets same group
  // But randomized assignment across user population
  // Prevents systematic bias

  function hashString(s) {
    let hash = 0;
    for (let i = 0; i < s.length; i++) {
      hash = ((hash << 5) - hash) + s.charCodeAt(i);
      hash |= 0;
    }
    return (Math.abs(hash) % 1000) / 1000;
  }

  return {
    user123: getUserGroup('user123', 'test1'),  // Consistent
    user456: getUserGroup('user456', 'test1'),  // Different
    user123Again: getUserGroup('user123', 'test1')  // Same again
  };
}
```

4. Bloom Filters: Randomized Membership

```
function bloomFilter() {
  // Space-efficient approximate set membership
  // False positives: maybe, False negatives: definitely not

  class BloomFilter {
    constructor(size = 1000, hashFunctions = 3) {
      this.bits = new Uint8Array(size);
      this.size = size;
      this.k = hashFunctions;
    }

    _hash(item, seed) {
      // Simple hash (in practice: use cryptographic hash)
      let h = seed;
      for (let i = 0; i < item.length; i++) {
        h = ((h << 5) - h) + item.charCodeAt(i);
      }
      return Math.abs(h) % this.size;
    }

    add(item) {
      for (let i = 0; i < this.k; i++) {
        const index = this._hash(item, i);
        this.bits[index] = 1;
      }
    }

    contains(item) {
```

```
      for (let i = 0; i < this.k; i++) {
        const index = this._hash(item, i);
        if (!this.bits[index]) return false;  // Definitely not there
      }
      return true;  // Maybe there (false positive possible)
    }
  }

  const bf = new BloomFilter(1000, 3);

  bf.add('apple');
  bf.add('banana');

  console.log(bf.contains('apple'));   // true
  console.log(bf.contains('apple'));   // true (always yes)
  console.log(bf.contains('cherry'));  // false (might be false positive)

  // Space: O(n) bits vs O(n) bytes for hash set
  // Use: Check membership before expensive operation (DB query)
}
```

Common Misconceptions

❌ "Randomization means unpredictable results"

No: Randomized algorithms have bounded randomness

```
// Las Vegas: Always correct, randomized time
// You WILL get right answer
const sorted = quicksortRandomized([5, 2, 8, 1]);  // Always sorted

// Monte Carlo: Randomized correctness, but bounded
const probably_prime = millerRabin(1000000007, 40);
// 99.99999999% sure it's prime (2^-40 error)
```

❌ "Randomization is inefficient"

Often faster than deterministic alternatives:

```
// Randomized Quicksort: O(n log n) expected
// vs Guaranteed O(n log n) mergesort

// In practice: Quicksort faster despite same complexity
// Why: Cache locality, fewer data moves, better constants
```

❌ "We should always use deterministic algorithms"

Deterministic can be vulnerable:

```
// Deterministic algorithm with worst-case O(n²):
// Works fine until adversary sends crafted input

// Randomized algorithm with O(n log n) expected:
// Adversary can't do anything (chooses random, not adversary)
```

Practical Mini-Exercises

▶ **Exercise 1: Shuffle** (Click to expand)

▶ **Exercise 2: Reservoir Sample** (Click to expand)

Summary Cheat Sheet

```
// RANDOMIZED VS DETERMINISTIC
Randomized:    Breaks patterns, no worst-case inputs, constant time
Deterministic: Predictable, can be slow on adversarial inputs

// ALGORITHM TYPES
Las Vegas:   Always correct, randomized time (e.g., Quicksort)
Monte Carlo: Fixed time, probably correct (e.g., Primality testing)

// USE CASES
Hash tables:     Randomized hash function seed
Load balancing:  Random server selection + jitter
Sorting:         Randomized pivot selection
Sampling:        Reservoir sampling
Shuffling:       Fisher-Yates
Membership:      Bloom filters
Retries:         Exponential backoff with jitter
```

Next Steps

✅ **You've completed**: Randomized algorithms

➡️ **Final topic**: 18. Risk, Uncertainty & When Math Breaks - Models vs reality, assumptions, sensitivity analysis

Before moving on:

```
// Challenge: Implement randomized binary search (pick random pivot)
function randomBinarySearch(arr, target) {
  // Your solution
}
```
