15. Information Theory

Phase 5: Modern Software Math
⏱ ~40 minutes | 🔐 Foundational for Compression/Hashing/Security | 📊 Entropy as Uncertainty

---

What Problem This Solves

You're encountering:

- JPEG/PNG compression algorithms that "preserve" data while reducing size
- How hash functions actually work (why they're "random" yet deterministic)
- Entropy in password strength ("Why is 'H3llo!2' weak but 'HELLo123' not?")
- Data redundancy ("Why are some files compressible and others not?")
- Error correction in network protocols
- Random number generation and its pitfalls

Without information theory, compression seems magical ("how does it know what to throw away?"), hash collisions seem random, and password strength metrics seem arbitrary.

With information theory, you understand that information has structure, redundancy can be eliminated, and uncertainty can be quantified mathematically.

Intuition & Mental Model

The Core Insight: Entropy = Uncertainty

```
Information = Reduction in uncertainty

Before message:  "What will the outcome be?"
After message:   "I know the outcome is X"

Information quantifies the reduction
```

Mental Model: Surprise as Quantity

```
Common message:  "Sun rose today" → Low surprise → Low information
Rare message:    "Snow in August" → High surprise → High information

Entropy = Average surprise across all possible messages
```

Core Concepts

1. Entropy: Quantifying Uncertainty

```
function entropy(probabilities) {
  // Shannon entropy: H = -Σ p(x) × log2(p(x))
  // Units: bits (log base 2)

  return -probabilities
```

```
    .filter(p => p > 0)  // Ignore zero probabilities (0×log(0) = 0)
    .reduce((sum, p) => sum + p * Math.log2(p), 0);
}

// Example: Fair coin flip
entropy([0.5, 0.5]);  // 1 bit
// Maximum uncertainty: heads or tails equally likely

// Example: Biased coin (80% heads)
entropy([0.8, 0.2]);  // 0.722 bits
// Less uncertainty: probably heads

// Example: Rigged coin (always heads)
entropy([1.0, 0.0]);  // 0 bits
// No uncertainty: definitely heads
```

Intuition:

```
1 bit = Yes/No question
2 bits = Distinguishes 4 outcomes
n bits = Distinguishes 2^n outcomes

Fair 6-sided die:
log2(6) = 2.585 bits per roll
```

Real Example: Compression Potential

```
function compressionPotential(text) {
  // Count character frequencies
  const chars = new Map();
  for (const c of text) {
    chars.set(c, (chars.get(c) || 0) + 1);
  }

  // Calculate probability of each character
  const probs = Array.from(chars.values()).map(count => count / text.length);

  // Entropy tells us minimum bits per character
  const h = entropy(probs);
  const originalBits = text.length * 8;  // ASCII = 8 bits/char
  const compressedBits = text.length * h;

  return {
    characters: chars.size,
    entropy: h.toFixed(3),
    originalBits,
    compressedBits: compressedBits.toFixed(0),
    compressionRatio: (compressedBits / originalBits).toFixed(1),
    potential: `${(100 - (compressedBits / originalBits * 100)).toFixed(0)}%
reduction`
  };
```

```
}

// Highly repetitive text compresses well
compressionPotential('AAAAAABBBBBBCCCC');
/* {
  entropy: '1.623',
  originalBits: 128,
  compressedBits: '25.97',
  compressionRatio: '0.2',
  potential: '80% reduction'
}
// Highly structured = low entropy = high compressibility */

// Random text doesn't compress
compressionPotential('ABCDEFGHIJKLMNOP');
/* {
  entropy: '4.0',
  compressedBits: '64',
  compressionRatio: '0.5',
  potential: '50% reduction'
}
// Uniform distribution = high entropy = less compressible */
```

2. Huffman Coding: Optimal Compression

Idea: Use shorter codes for common characters

```
function huffmanCoding(text) {
  // Build frequency table
  const freq = new Map();
  for (const c of text) {
    freq.set(c, (freq.get(c) || 0) + 1);
  }

  // Build Huffman tree (simplified for demo)
  // In practice: heap-based algorithm

  const codes = {};
  if (freq.size === 1) {
    codes[Array.from(freq.keys())[0]] = '0';
  } else {
    // Assign codes based on frequency
    const sorted = Array.from(freq.entries()).sort((a, b) => b[1] - a[1]);

    let code = '';
    for (const [char, _] of sorted) {
      codes[char] = code || '0';
      code += '0';
    }
  }
```

```
  // Encode text
  const encoded = text.split('').map(c => codes[c]).join('');

  return {
    codes,
    originalBits: text.length * 8,
    encodedBits: encoded.length,
    originalText: text,
    encoded,
    compressionRatio: (encoded.length / (text.length * 8) * 100).toFixed(1) + '%'
  };
}

huffmanCoding('AAABBC');
/* {
  codes: { A: '', B: '0', C: '00' },
  originalBits: 48,
  encodedBits: 12,
  encoded: '000000000000',
  compressionRatio: '25.0%'
}
// A is most common (no code!) → shortest encoding */
```

3. Hash Functions: One-Way Entropy Maps

Key Property: Small change in input → completely different hash

```
function simpleHash(text, tableSize = 256) {
  // Simple hash: map any input to fixed-size output
  let hash = 0;

  for (let i = 0; i < text.length; i++) {
    hash = (hash * 31 + text.charCodeAt(i)) % tableSize;
  }

  return hash;
}

// Deterministic
console.log(simpleHash('hello'));  // Always same value
console.log(simpleHash('hello'));

// Avalanche effect: small change → big difference
console.log(simpleHash('hello'));   // e.g. 42
console.log(simpleHash('hallo'));   // e.g. 157 (completely different)

// Uniform distribution: different inputs spread across output space
console.log(
  'hello'.split('').map((_, i) => simpleHash('hello' + i)).slice(0, 5)
);
// Shows different characters → very different hashes
```

Why This Matters:

```
// Hash table lookup: O(1) on average
function hashTable() {
  const table = new Map();
  const size = 1000;

  function set(key, value) {
    const hash = simpleHash(key, size);
    if (!table.has(hash)) {
      table.set(hash, []);
    }
    table.get(hash).push({ key, value });
  }

  function get(key) {
    const hash = simpleHash(key, size);
    const bucket = table.get(hash) || [];
    return bucket.find(item => item.key === key)?.value;
  }

  return { set, get };
}

// Cryptographic hashes (SHA-256): can't reverse, collision-resistant
```

4. Mutual Information: Data Relationships

Question: How much knowing X tells you about Y?

```
function mutualInformation(xy_joint, x_marginal, y_marginal) {
  // MI = Σ p(x,y) × log2(p(x,y) / (p(x) × p(y)))
  // If X and Y independent: MI = 0
  // If X determines Y: MI = H(Y)

  let mi = 0;

  for (const x in xy_joint) {
    for (const y in xy_joint[x]) {
      const pxy = xy_joint[x][y];
      const px = x_marginal[x];
      const py = y_marginal[y];

      if (pxy > 0) {
        mi += pxy * Math.log2(pxy / (px * py));
      }
    }
  }

  return mi;
}
```

```
// Example: Relationship between weather and ice cream sales
const data = {
  sunny_icecream: 0.4,   // P(sunny & bought ice cream)
  sunny_no: 0.1,         // P(sunny & no ice cream)
  rainy_icecream: 0.3,   // P(rainy & bought)
  rainy_no: 0.2          // P(rainy & no)
};

const sunny = 0.5, rainy = 0.5;
const icecream = 0.7, no = 0.3;

// High MI = weather tells you about ice cream purchases
```

Real Use Case: Feature Importance in ML

```
function featureImportance(features, target) {
  // Calculate MI between each feature and target
  // Higher MI = better predictor

  return features.map((feature, i) => ({
    featureName: feature.name,
    mutualInformation: calculateMI(feature.values, target),
    importance: 'High if MI > H(target)'
  }));
}
```

5. Data Redundancy & Error Correction

Problem: Data gets corrupted in transmission. How to recover?

```
function hammingCode(data) {
  // Add parity bits to detect and correct single-bit errors
  // data: 4-bit message, result: 7-bit encoded with 3 parity bits

  const encoded = [...data, 0, 0, 0];  // 7 bits total

  // Parity bit 1: covers positions 1,3,5,7
  let p1 = 0;
  for (let i of [0, 2, 4, 6]) {
    p1 ^= data[i];  // XOR (parity check)
  }
  encoded[4] = p1;

  // Parity bit 2: covers positions 2,3,6,7
  let p2 = 0;
  for (let i of [1, 2, 5, 6]) {
    p2 ^= (i < data.length ? data[i] : 0);
  }
  encoded[5] = p2;
```

```
  // Parity bit 3: covers positions 4,5,6,7
  let p3 = 0;
  for (let i of [3, 4, 5, 6]) {
    p3 ^= (i < data.length ? data[i] : 0);
  }
  encoded[6] = p3;

  return encoded;
}

// 4 data bits → 7 encoded bits (3 redundant bits)
// Can detect AND correct any single-bit error
hammingCode([1, 0, 1, 1]);  // [1, 0, 1, 1, ?, ?, ?]

// Trade-off: 75% redundancy for 1-bit error correction
// Used in DRAM, network protocols, space missions
```

6. Random Number Generation: Entropy Sources

Problem: How to generate truly random numbers?

```
function randomQuality(numbers) {
  // Measure randomness using entropy
  const freq = new Map();

  for (const n of numbers) {
    freq.set(n, (freq.get(n) || 0) + 1);
  }

  const probs = Array.from(freq.values()).map(f => f / numbers.length);
  const h = entropy(probs);

  // Expected entropy for uniform distribution
  const maxEntropy = Math.log2(freq.size);

  return {
    uniqueValues: freq.size,
    entropy: h.toFixed(3),
    maxPossible: maxEntropy.toFixed(3),
    qualityScore: (h / maxEntropy * 100).toFixed(1) + '%'
  };
}

// Bad randomness (Math.random() in some engines)
const bad = Array(1000).fill(0).map(() => Math.floor(Math.random() * 10));
randomQuality(bad);  // Should be ~3.32 for uniform

// Cryptographically secure (low bias)
const crypto = require('crypto');
const good = Array(1000).fill(0).map(() =>
  crypto.randomBytes(1)[0] % 10
```

```
);
randomQuality(good);  // Closer to theoretical max
```

Software Engineering Connections

1. Password Strength

```
function passwordEntropy(password) {
  // Character space: lowercase (26) + uppercase (26) + digits (10) + symbols (32)

  let charset = 0;
  if (/[a-z]/.test(password)) charset += 26;
  if (/[A-Z]/.test(password)) charset += 26;
  if (/[0-9]/.test(password)) charset += 10;
  if (/[^a-zA-Z0-9]/.test(password)) charset += 32;

  // Entropy = password_length × log2(charset)
  const bits = password.length * Math.log2(charset);

  return {
    password,
    charset,
    length: password.length,
    entropy: bits.toFixed(1) + ' bits',
    strength: bits >= 80 ? 'Strong' : bits >= 60 ? 'OK' : 'Weak',
    crackTime: bits >= 80 ? '>1000 years' : bits >= 60 ? 'Weeks to years' : 'Days'
  };
}

passwordEntropy('password123');      // ~47 bits (weak)
passwordEntropy('p@ssw0rd_ABC!x');  // ~80 bits (strong)
passwordEntropy('HELLo123');        // ~48 bits (weak)
passwordEntropy('H3llo!2');         // ~40 bits (very weak)

// Length and character diversity both matter!
```

2. Data Format Efficiency

```
function compareFormats(data) {
  // JSON vs binary (Protocol Buffers, MessagePack)

  const json = JSON.stringify(data);
  const jsonBytes = new Blob([json]).size;

  // Estimate binary (very rough)
  // Real protobuf is more complex
  const binaryEstimate =
    4 +  // Field IDs
    Object.values(data).reduce((sum, v) =>
```

```
      sum + (typeof v === 'string' ? v.length : 4), 0
    );

  return {
    format_json: jsonBytes + ' bytes',
    format_binary_est: binaryEstimate + ' bytes',
    ratio: (binaryEstimate / jsonBytes).toFixed(2) + 'x smaller',
    benefit: 'Important for networks and storage'
  };
}

compareFormats({
  userId: 12345,
  name: 'John Doe',
  email: 'john@example.com'
});
```

3. Cache Efficiency

```
function cacheEntropy(accessPattern) {
  // Uniform access: High entropy, low cache effectiveness
  // Skewed access: Low entropy, high cache effectiveness

  const freq = new Map();
  accessPattern.forEach(item =>
    freq.set(item, (freq.get(item) || 0) + 1)
  );

  const probs = Array.from(freq.values()).map(f => f / accessPattern.length);
  const h = entropy(probs);

  // Pareto principle: 80/20 rule
  const sorted = Array.from(freq.values()).sort((a, b) => b - a);
  const top20 = sorted.slice(0, Math.ceil(sorted.length * 0.2));
  const coverage = top20.reduce((sum, f) => sum + f, 0) / accessPattern.length;

  return {
    entropy: h.toFixed(2),
    paretoCoverage: (coverage * 100).toFixed(1) + '%',
    recommendation: coverage > 0.8
      ? 'Excellent: 20% of items cover 80%+ of accesses. Cache 20% for great ROI.'
      : 'Poor: Uniform access, cache less effective'
  };
}

// Real-world: 80/20 rule in many systems
cacheEntropy([1, 1, 1, 1, 1, 2, 2, 2, 3, 4, 5]);
/* {
  entropy: '2.46',
  paretoCoverage: '90.9%',
  recommendation: 'Excellent: ...'
```

```
}
// Cache item 1 hits 45% of requests! Great for cache */
```

4. Error Detection

```
function errorDetection(data) {
  // CRC (Cyclic Redundancy Check): cheap error detection
  // SHA-256: expensive but cryptographically secure

  function simpleCRC32(data) {
    let crc = 0xffffffff;
    for (let i = 0; i < data.length; i++) {
      crc = crc ^ data.charCodeAt(i);
      for (let j = 0; j < 8; j++) {
        crc = (crc >>> 1) ^ (0xedb88320 & -(crc & 1));
      }
    }
    return (crc ^ 0xffffffff).toString(16);
  }

  const crc = simpleCRC32(data);

  return {
    data,
    checksum: crc,
    detection: 'Catches bit flips, transmission errors',
    use: 'Network packets, file downloads'
  };
}

errorDetection('Hello World');
// { checksum: 'abc123', detection: '...', use: '...' }
```

Common Misconceptions

❌ "More entropy = always better"

Wrong: Trade-off with usability

```
// Very high entropy password: R$9jK#@2mP!xQ%
// Often can't be typed on some keyboards, hard to remember

// Better: Passphrase "CorrectHorseBatteryStaple"
// Lower entropy per character, but total entropy is still high
// Much more usable!
```

❌ "Random looks random"

Pseudorandom sequences have patterns:

```
// Math.random() output seems random, but has structure
// If you know internal state, you can predict all future values

// True randomness requires entropy source:
// - Atmospheric noise
// - Hardware entropy (CPU timing jitter)
// - /dev/urandom on Unix
```

❌ "Compression means losing data"

Lossless compression preserves all data:

```
// GZIP: Removes redundancy, file shrinks, decompress gets original
// JPEG: Removes imperceptible detail, can't fully recover

// Shannon limit: Can't compress random data below entropy
```

❌ "Hash collisions don't matter"

Depends on context:

```
// Weak hash (simple): Collisions likely in reasonable time
// Cryptographic hash (SHA-256): Collision requires 2^128 operations
// For hash table: Collision means slow lookup (acceptable)
// For security: Collision is catastrophic
```

Practical Mini-Exercises

▶ **Exercise 1: Entropy Calculation** (Click to expand)

▶ **Exercise 2: Compression Ratio** (Click to expand)

Summary Cheat Sheet

```
// ENTROPY (Uncertainty/Information)
H(X) = -Σ p(x) × log2(p(x))
Units: bits

// Key Values
1 bit = Distinguishes 2 outcomes
4 bits = Distinguishes 16 outcomes
80 bits ≈ Safe password strength

// Application Guidelines
Low entropy → High compressibility
High entropy → Cannot compress (random)
Low entropy source → Insecure RNG
High entropy source → Good randomness
```

```
// Hash Properties
- Deterministic: same input → same hash
- Avalanche: tiny change → different hash
- Uniform: outputs spread evenly
```

Next Steps

✅ **You've completed Phase 5 (Modern Software Math)**
➡️ **Up next**: Phase 6 - Real-World Applications

Topic Summary:

- 13: Linear Algebra → Vectors, matrices, transformations
- 14: Optimization → Trade-offs, constraints, algorithms
- 15: Information Theory → Entropy, compression, hashing

Before moving on:

```
// Challenge: Design a password meter
function passwordMeter(password) {
  // Return entropy and strength (weak/ok/strong)
}
```
