Discrete Mathematics (Applied)

What Problem This Solves

Discrete mathematics is the mathematics of countable, separate things.

Unlike calculus (which deals with smooth curves and continuous change), discrete math handles:

- Individual items (users, requests, nodes)
- Distinct states (on/off, true/false, connected/disconnected)
- Step-by-step processes (algorithms, state machines)
- Structures with clear boundaries (sets, graphs, trees)

In software, almost everything is discrete.

You're not measuring the flow of water—you're counting database records, tracking API calls, managing relationships between entities, and routing through networks.

Intuition & Mental Model

Think: LEGO Blocks, Not Clay

Continuous math (calculus): Smooth, moldable, infinite precision

```
Clay → You can make infinitely small adjustments
```

Discrete math: Distinct pieces, countable, finite precision

```
LEGO → You have specific blocks that connect in specific ways
```

Software is LEGO. Your database has exactly 1,247 users (not 1,247.5). A request either succeeds or fails (not "73.2% succeeded"). A graph has 10 nodes, not "approximately 10".

Core Concepts

1. Sets: Collections Without Order

A set is a collection of distinct objects.

```
// Sets in code
const users = new Set(['alice', 'bob', 'charlie']);
const admins = new Set(['alice']);
const guests = new Set(['bob', 'charlie', 'david']);

// Key property: no duplicates
users.add('alice'); // No effect, already exists
console.log(users.size); // Still 3
```

Set notation:

```
A = {1, 2, 3}
B = {2, 3, 4}

|A| = 3  (cardinality/size)
```

Set operations:

```
Union (∪):        A ∪ B = {1, 2, 3, 4}        "Either in A or B"
Intersection (∩): A ∩ B = {2, 3}              "In both A and B"
Difference (−):   A − B = {1}                 "In A but not B"
```

Real-world examples:

```
// Users who can access feature
const canAccessFeature = premiumUsers.union(betaTesters);

// Users in both groups
const overlap = mobileUsers.intersection(webUsers);

// Unsubscribed
const unsubscribed = allUsers.difference(subscribers);
```

Visual: Set Operations

```
Union (A ∪ B):
    ┌─────┐
    │  A  │
    │ ┌───┼───┐
    │ │▓▓▓│▓▓▓│
    └─┼───┘   │
      │   B   │
      └───────┘

Intersection (A ∩ B):
    ┌─────┐
    │  A  │
    │ ┌───┼───┐
    │ │▓▓▓│   │
    └─┼───┘   │
      │   B   │
      └───────┘

Difference (A − B):
    ┌─────┐
    │▓▓A▓▓│
    │ ┌───┼───┐
    │ │   │   │
    └─┼───┘   │
      │   B   │
      └───────┘
```

2. Relations: How Things Connect

A relation describes connections between elements.

Examples from code:

```
// User → Posts (one-to-many)
const posts = {
  'alice': [1, 2, 3],
  'bob': [4, 5]
};

// User → User (many-to-many)
const friendships = [
  ['alice', 'bob'],
  ['bob', 'charlie'],
  ['alice', 'charlie']
];
```

Types of relations:

Relation	Meaning	Example
One-to-One	Each input has exactly one output	`userId → email`
One-to-Many	Each input can have multiple outputs	`userId → posts`
Many-to-Many	Both sides can have multiple connections	`users ↔ groups`

Properties of relations:

```
Reflexive:  Every element relates to itself
            Example: x ≤ x (always true)

Symmetric:  If A relates to B, then B relates to A
            Example: "is married to"

Transitive: If A relates to B and B relates to C, then A relates to C
            Example: "is ancestor of"
```

In databases:

```
-- One-to-many (reflexive)
CREATE TABLE posts (
  user_id INT REFERENCES users(id)
);

-- Many-to-many (symmetric)
CREATE TABLE friendships (
  user_a_id INT,
  user_b_id INT
);
```

```sql
-- Transitive closure
WITH RECURSIVE ancestors AS (
  -- "is parent of" becomes "is ancestor of"
  ...
)
```

---

3. Functions (Mappings): Reliable Transformations

A function maps each input to exactly one output.

```javascript
// Function (valid)
const square = x => x * x;
square(3); // Always 9

// Not a function (invalid)
const random = x => Math.random();
random(3); // Different result each time!
```

Mathematical definition:

```
f: A → B

For every element in A (domain),
there's exactly one element in B (codomain)
```

Types of functions:

```
Injective (one-to-one):
  Different inputs → different outputs
  Example: userId → email (no duplicates)

Surjective (onto):
  Every output is used
  Example: post → category (all categories have posts)

Bijective (both):
  Perfect pairing (invertible)
  Example: userId ↔ UUID
```

Real-world examples:

```javascript
// Injective (one-to-one)
const userIdToEmail = new Map([
  [1, 'alice@example.com'],
  [2, 'bob@example.com']
  // No two users share an email
]);

// Surjective (onto)
```

```javascript
const postToCategory = new Map([
  [101, 'tech'],
  [102, 'tech'],
  [103, 'finance'],
  [104, 'finance']
  // Every category is used
]);

// Bijective (both - invertible)
const userId ToUUID = new Map([
  [1, 'a1b2c3...'],
  [2, 'd4e5f6...']
]);
const uuidToUserId = new Map([
  ['a1b2c3...', 1],
  ['d4e5f6...', 2]
]);
```

---

4. Cardinality: Counting Carefully

Cardinality is the size of a set.

Simple cases:

```
{1, 2, 3} has cardinality 3
{a, b, c, d} has cardinality 4
∅ (empty set) has cardinality 0
```

Counting with operations:

```
|A ∪ B| = |A| + |B| - |A ∩ B|

Why subtract? Don't count the overlap twice!
```

Example:

```javascript
const mobileUsers = new Set(['alice', 'bob', 'charlie']);
const webUsers = new Set(['bob', 'charlie', 'david']);

// Total unique users?
// Wrong: 3 + 3 = 6
// Right: 3 + 3 - 2 = 4 (bob and charlie counted once)

const allUsers = new Set([...mobileUsers, ...webUsers]);
console.log(allUsers.size); // 4
```

Application: Database joins

```sql
-- How many users have EITHER mobile OR web sessions?
SELECT COUNT(DISTINCT user_id)
FROM (
```

```
  SELECT user_id FROM mobile_sessions
  UNION
  SELECT user_id FROM web_sessions
) combined;
```

---

5. Sequences: Ordered Collections

Unlike sets, sequences have order and allow duplicates.

```
// Set (no order, no duplicates)
const uniqueVisits = new Set([1, 2, 3, 2, 1]); // {1, 2, 3}

// Sequence (order matters, duplicates ok)
const clickSequence = [1, 2, 3, 2, 1]; // Exact order preserved
```

Common sequences:

```
Array:  [a, b, c, d]
String: "hello"  (sequence of characters)
Events: [login, click, scroll, logout]
```

Why order matters:

```
// Search history (sequence)
const searches = ['react', 'vue', 'react'];
// User searched React twice, with Vue in between

// Unique searches (set)
const uniqueSearches = new Set(searches); // {'react', 'vue'}
// Lost the repeated search and order
```

---

6. Combinatorics: Counting Possibilities

How many ways can you arrange or select things?

Permutations (order matters):

```
3 people (A, B, C) in 3 seats:
ABC, ACB, BAC, BCA, CAB, CBA = 6 ways = 3!
```

Combinations (order doesn't matter):

```
Choose 2 people from {A, B, C}:
AB, AC, BC = 3 ways = C(3,2) = 3
```

Formulas:

```
Permutations: P(n, r) = n! / (n-r)!
Combinations: C(n, r) = n! / (r!(n-r)!)
```

Real-world examples:

```
// Password permutations (order matters)
// 4-digit PIN from 0-9
const pinCombinations = 10 * 10 * 10 * 10; // 10,000

// Selecting team members (order doesn't matter)
// Choose 3 from 10 candidates
function combinations(n, r) {
  return factorial(n) / (factorial(r) * factorial(n - r));
}
console.log(combinations(10, 3)); // 120 possible teams

// Feature flag combinations (2^n)
// 5 feature flags, each on/off
const configurations = 2 ** 5; // 32 possible states
```

Software Engineering Connections

Type Systems (Set Theory)

```
// Union types (set union)
type Status = 'pending' | 'active' | 'inactive';

// Intersection types (set intersection)
type Admin = User & { role: 'admin' };

// Difference (exclude)
type NonAdmin = Exclude<User, Admin>;
```

Database Design

```
-- Sets: Tables are sets of rows
SELECT DISTINCT user_id FROM orders;

-- Relations: Foreign keys
CREATE TABLE orders (
  user_id INT REFERENCES users(id)
);

-- Cardinality: Counting
SELECT COUNT(*) FROM users WHERE active = true;

-- Set operations
SELECT * FROM users WHERE id IN (SELECT user_id FROM premium)
UNION
SELECT * FROM users WHERE id IN (SELECT user_id FROM beta);
```

Graph Structures

```
// Relations as adjacency list
const graph = {
  'A': ['B', 'C'],
  'B': ['C', 'D'],
  'C': ['D'],
  'D': []
};

// Set of visited nodes
const visited = new Set();

function dfs(node) {
  if (visited.has(node)) return;
  visited.add(node);
  for (const neighbor of graph[node]) {
    dfs(neighbor);
  }
}
```

API Design

```
// Functions as mappings
app.get('/users/:id', (req, res) => {
  // Map: userId → userData
  const user = db.users.findById(req.params.id);
  res.json(user);
});

// Idempotent = same input always gives same output
app.delete('/users/:id', (req, res) => {
  // DELETE is idempotent: repeating has no additional effect
  db.users.delete(req.params.id);
});
```

Caching (Sets & Functions)

```
// Cached values = set of computed inputs
const cache = new Map();

function expensiveFunction(x) {
  if (cache.has(x)) {
    return cache.get(x); // O(1) lookup
  }
  const result = /* expensive computation */;
  cache.set(x, result);
  return result;
}
```

```
// Cache invalidation = removing from set
cache.delete(x);
```

Common Misconceptions

❌ "Sets are just arrays without duplicates"

Partially true, but sets have no inherent order. `{1, 2, 3}` equals `{3, 2, 1}`.

JavaScript `Set` maintains insertion order (implementation detail), but mathematical sets don't.

❌ "Relations are just functions"

Functions are special relations where each input maps to exactly one output. Relations can have one input map to multiple outputs.

```
// Relation (not a function)
const manages = [
  ['alice', 'team1'],
  ['alice', 'team2'], // Alice manages multiple teams
];

// Function
const email = new Map([['alice', 'alice@example.com']]);
```

❌ "Discrete math = no real numbers"

Not quite. Discrete math studies countable structures. You can have sets of real numbers, but the SET itself is countable (finite elements).

```
const prices = new Set([9.99, 19.99, 29.99]); // Real numbers, but finite set
```

❌ "Cardinality is always finite"

In theory, no. Infinite sets exist (like all natural numbers). But in programming, we work with finite sets (limited memory).

Practical Mini-Exercises

Exercise 1: User Permissions

You have three sets:

```
const readers = new Set(['alice', 'bob', 'charlie']);
const writers = new Set(['alice', 'david']);
const admins = new Set(['alice']);
```

Compute:

1. Users with any access (union)
2. Users who can both read AND write (intersection)
3. Users who can read but NOT write (difference)

▶ Solution

Exercise 2: API Mapping

Determine if each API endpoint represents an injective, surjective, or bijective function:

```
// A: Get user by ID
GET /users/:id → User object

// B: Get posts by user
GET /users/:userId/posts → Post[]

// C: Get user by email
GET /users/by-email/:email → User object

// D: Get category by post
GET /posts/:postId/category → Category
```

▶ Solution

Exercise 3: Counting Configurations

You're building a feature flag system with 4 flags:

- `darkMode` : on/off
- `betaFeatures` : on/off
- `analytics` : on/off
- `notifications` : on/off

How many possible configurations exist?

▶ Solution

Summary Cheat Sheet

Core Concepts

Concept	Definition	Code Example
Set	Collection of unique elements	`new Set([1, 2, 3])`
Relation	Connection between elements	`[['a', 'b'], ['b', 'c']]`
Function	Reliable one-output mapping	`x => x * 2`
Cardinality	Size of a set	`set.size`
Sequence	Ordered collection	`[1, 2, 3]`

Set Operations

```
const A = new Set([1, 2, 3]);
const B = new Set([2, 3, 4]);

// Union: A ∪ B
const union = new Set([...A, ...B]); // {1,2,3,4}

// Intersection: A ∩ B
const intersection = new Set([...A].filter(x => B.has(x))); // {2,3}

// Difference: A - B
const difference = new Set([...A].filter(x => !B.has(x))); // {1}
```

Function Types

```
Injective:  Different inputs → different outputs
Surjective: Every output is used
Bijective:  Both (invertible)
```

Counting

```
Permutations (order matters): n! / (n-r)!
Combinations (order doesn't matter): n! / (r!(n-r)!)
Boolean configs: 2^n
```

Next Steps

Discrete math provides the vocabulary for talking about structure and relationships.

Next, we'll explore **graph theory**—where relations become visual networks and algorithms come to life.

Continue to: 02-graph-theory.md