14. Optimization

Phase 5: Modern Software Math
⏱ ~45 minutes | 🎯 Decision Framework | ⚖️ Constraints vs Objectives

---

What Problem This Solves

You're trying to:

- Minimize latency while staying within budget
- Maximize throughput given CPU/memory limits
- Find the best database index strategy (can't index everything)
- Allocate developers across projects to maximize productivity
- Price your SaaS to maximize profit (not just revenue)
- Route traffic to minimize response time
- Compress data as much as possible without losing quality

Without optimization thinking, you make decisions heuristically ("use more servers!") or arbitrarily ("let's split time equally"). You don't recognize when you're stuck in local optima or when constraints fundamentally limit solutions.

With optimization, you frame problems precisely, identify what you're really trying to maximize or minimize, and make decisions systematically.

Intuition & Mental Model

The Core Insight: Landscape Hiking

```
OPTIMIZATION = Finding the best point in a landscape

           ↑ Objective
           |     ⛰️ Global optimum (best)
           |    / \
           |   /   \  ← Local optima (stuck here?)
           |  /     \
           | /       \
           |___________→ Constraints/Feasible region
```

Types of Optima:

```
Global Optimum:  Absolute best possible solution
                 Rarely achievable (NP-hard in most real problems)

Local Optimum:   Best in neighborhood, but worse elsewhere
                 Where greedy algorithms get stuck

Feasible:        Satisfies all constraints (good enough)
                 Often the real goal in practice
```

Core Concepts

1. Framing the Problem

Pattern: What do we optimize? What are constraints?

```
function formulateProblem(scenario) {
  // Every optimization problem has 3 parts:

  return {
    objective: {
      maximize: 'revenue / cost ratio',
      or_minimize: 'latency / energy',
      description: 'What we want most'
    },
    constraints: {
      budget: '$100k',
      cpu_cores: 8,
      network: '1Gbps',
      description: 'Immovable limits'
    },
    decision_variables: {
      servers: 'how many?',
      instance_type: 'which type?',
      region: 'which region?',
      description: 'What we control'
    }
  };
}

// Example: E-commerce infrastructure
formulateProblem({
  maximize: 'Concurrent users (throughput)',
  constraints: [
    'Budget: $50k/month',
    'Team size: 2 engineers',
    'SLA: 99.9% uptime'
  ],
  variables: [
    'Instance count: 1-100',
    'Instance type: t3, m5, c5',
    'Region: us-east-1, eu-west-1'
  ]
});
```

Three Key Questions:

1. What are we really trying to optimize? (revenue, latency, accuracy, cost)
2. What constraints are actually hard? (regulatory, physical, financial)
3. What can we actually change? (budget, team, architecture)

2. Linear Programming (Intuition)

Problem: Maximize profit given constraints

```
function linearProgrammingExample() {
  // Bakery: Make brownies and cookies
  // Brownies: $2 profit, 1 hour baking time, 2 cups flour
  // Cookies:  $1.50 profit, 0.5 hours baking time, 1 cup flour
  // Constraints: 40 hours/week, 80 cups flour/week

  // Decision variables: b = brownies, c = cookies
  // Maximize: 2b + 1.5c
  // Subject to:
  //   b + 0.5c ≤ 40  (time constraint)
  //   2b + c ≤ 80    (flour constraint)
  //   b, c ≥ 0       (non-negativity)

  // Feasible region = polygon
  // Optimum at a corner point (linear programming property)

  const corners = [
    { b: 0, c: 0, profit: 0 },
    { b: 40, c: 0, profit: 80 },      // Make only brownies
    { b: 0, c: 80, profit: 120 },     // Make only cookies
    { b: 20, c: 40, profit: 100 },    // Mix
  ];

  return corners
    .sort((a, b) => b.profit - a.profit)[0];
}

console.log(linearProgrammingExample());
// { b: 0, c: 80, profit: 120 }
// Make only cookies! (better margin given constraints)
```

Real Example: Server Allocation

```
function allocateServers() {
  // Web servers: $500/month, 100 requests/sec, low-cost
  // API servers: $1000/month, 50 requests/sec, high-margin
  // Budget: $5000/month, need 300 req/sec minimum

  const budget = 5000;
  const minThroughput = 300;

  const solutions = [];

  for (let web = 0; web <= 10; web++) {
    for (let api = 0; api <= 5; api++) {
      const cost = web * 500 + api * 1000;
      const throughput = web * 100 + api * 50;
```

```
      const profit = throughput * 10;  // $10 per req

      if (cost <= budget && throughput >= minThroughput) {
        solutions.push({
          webServers: web,
          apiServers: api,
          cost,
          throughput,
          profit
        });
      }
    }
  }

  return solutions.sort((a, b) => b.profit - a.profit)[0];
}

console.log(allocateServers());
/* {
  webServers: 10,
  apiServers: 0,
  cost: 5000,
  throughput: 1000,
  profit: 10000
}
// Web servers more cost-effective in this scenario */
```

3. Greedy Algorithms: Local Optimization

```
function greedyScheduling(tasks) {
  // Greedy: Always pick the task with highest value first
  // (Not optimal globally, but fast)

  const sorted = [...tasks].sort((a, b) => b.value - a.value);
  let time = 0;
  let totalValue = 0;
  const scheduled = [];

  for (const task of sorted) {
    if (time + task.duration <= 8) {  // 8-hour day
      scheduled.push(task.name);
      time += task.duration;
      totalValue += task.value;
    }
  }

  return { scheduled, totalValue, timeUsed: time };
}

// Tasks: [name, duration, value]
const tasks = [
```

```
  { name: 'Feature A', duration: 3, value: 50 },
  { name: 'Feature B', duration: 2, value: 40 },
  { name: 'Feature C', duration: 4, value: 60 },
  { name: 'Bug fix', duration: 1, value: 30 }
];

greedyScheduling(tasks);
/* {
  scheduled: [ 'Feature C', 'Feature A', 'Bug fix' ],
  totalValue: 140,
  timeUsed: 8
}
// Greedy picked high-value tasks first, packed the 8-hour day well
// (Not always optimal: sometimes smaller high-value tasks beat bigger ones) */
```

When Greedy Fails:

```
function greedyFailsExample() {
  const budget = 100;
  const items = [
    { name: 'Item A', cost: 50, value: 60 },   // 1.2x value/cost
    { name: 'Item B', cost: 30, value: 40 },   // 1.33x ← Greedy picks first
    { name: 'Item C', cost: 20, value: 20 },   // 1.0x
    { name: 'Item D', cost: 50, value: 70 }    // 1.4x but can't fit with A
  ];

  // Greedy (by value/cost): A + B = cost 80, value 100
  // Optimal: B + D = cost 80, value 110

  // Greedy got stuck locally
}
```

4. Dynamic Programming: Optimal Substructure

Key Idea: If optimal solution contains smaller optimal solutions, use recursion + memoization

```
function fibonacciOptimization() {
  // Naive recursion: O(2^n) - exponential!
  function naiveFib(n) {
    if (n <= 1) return n;
    return naiveFib(n - 1) + naiveFib(n - 2);  // Recalculates same values!
  }

  // Dynamic programming: O(n) - linear!
  function dpFib(n, memo = {}) {
    if (n in memo) return memo[n];
    if (n <= 1) return n;

    memo[n] = dpFib(n - 1, memo) + dpFib(n - 2, memo);
    return memo[n];
  }
```

```
  // Time comparison for n=40
  console.time('Naive');
  naiveFib(35);  // Takes ~3 seconds
  console.timeEnd('Naive');

  console.time('DP');
  dpFib(35);  // Takes ~1ms
  console.timeEnd('DP');
}

// Key: Memoization avoids recalculating overlapping subproblems
```

Real Example: Longest Common Subsequence

```
function longestCommonSubsequence(str1, str2) {
  // DP table: lcs[i][j] = LCS of str1[0..i] and str2[0..j]
  const m = str1.length, n = str2.length;
  const dp = Array(m + 1).fill(0).map(() => Array(n + 1).fill(0));

  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      if (str1[i - 1] === str2[j - 1]) {
        dp[i][j] = dp[i - 1][j - 1] + 1;
      } else {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
      }
    }
  }

  return dp[m][n];
}

longestCommonSubsequence('ABCDGH', 'AEDFHR');  // 3 (ADH)

// Applications: Git diff, DNA sequence alignment, plagiarism detection
```

5. Constraint Satisfaction Problems (CSP)

Question: Find an assignment that satisfies all constraints

```
function solveScheduling(meetings, rooms) {
  // Meetings: [name, duration, participants]
  // Rooms: [name, capacity]
  // Constraints: No room conflicts, enough capacity

  function isValid(assignment, meeting, room, time) {
    // Check capacity
    if (room.capacity < meeting.participants) return false;

    // Check room availability
```

```javascript
    for (const [m, r, t] of assignment) {
      if (r === room.name) {
        // Time conflict?
        if (!(time + meeting.duration <= t || t + m.duration <= time)) {
          return false;
        }
      }
    }

    return true;
  }

  function backtrack(assignment, meetingIndex) {
    if (meetingIndex === meetings.length) {
      return assignment;  // Found valid assignment!
    }

    const meeting = meetings[meetingIndex];

    for (const room of rooms) {
      for (let time = 9; time < 17; time++) {
        if (isValid(assignment, meeting, room, time)) {
          assignment.push([meeting, room.name, time]);
          const result = backtrack(assignment, meetingIndex + 1);
          if (result) return result;
          assignment.pop();  // Backtrack
        }
      }
    }

    return null;  // No solution
  }

  return backtrack([], 0);
}

const meetings = [
  { name: 'Standup', duration: 1, participants: 5 },
  { name: 'Pairing', duration: 2, participants: 2 },
  { name: 'Planning', duration: 3, participants: 8 }
];

const rooms = [
  { name: 'Conference A', capacity: 10 },
  { name: 'Conference B', capacity: 6 }
];

// Finds valid schedule respecting all constraints
```

6. Gradient Descent: Continuous Optimization

Problem: Minimize a smooth function by following the slope

```
function gradientDescent(startX, learningRate = 0.01, iterations = 1000) {
  // Objective: Minimize f(x) = (x - 5)^2 (has minimum at x=5)

  const f = x => Math.pow(x - 5, 2);
  const df = x => 2 * (x - 5);  // Derivative

  let x = startX;
  const history = [{ x, fx: f(x), iter: 0 }];

  for (let i = 1; i <= iterations; i++) {
    const gradient = df(x);
    x = x - learningRate * gradient;  // Move in opposite direction of gradient

    if (i % 100 === 0) {
      history.push({ x: x.toFixed(4), fx: f(x).toFixed(4), iter: i });
    }
  }

  return history;
}

gradientDescent(0);
/* [
  { x: 0, fx: 25, iter: 0 },
  { x: '3.62', fx: '1.91', iter: 100 },
  { x: '4.74', fx: '0.07', iter: 200 },
  { x: '4.96', fx: '0.0018', iter: 300 },
  ...
  → Converges to x=5, f(x)=0
]
// This is how neural networks learn! */
```

Real Example: Price Optimization

```
function optimizePrice() {
  // Revenue = price × demand
  // Demand = 1000 - 20 × price (demand decreases with price)
  // Revenue(p) = p × (1000 - 20p) = 1000p - 20p²

  // Maximum at: dRevenue/dp = 0 → 1000 - 40p = 0 → p = 25

  const revenue = p => p * (1000 - 20 * p);
  const dRevenue = p => 1000 - 40 * p;

  let price = 0;
  for (let iter = 0; iter < 100; iter++) {
    const gradient = dRevenue(price);
    price += 0.01 * gradient;  // Gradient ascent (maximize)
  }

  return {
```

```
    optimalPrice: price.toFixed(2),
    maxRevenue: revenue(price).toFixed(0)
  };
}

console.log(optimizePrice());
// { optimalPrice: '25.00', maxRevenue: '12500' }
// Price at $25 maximizes revenue (not always highest price!)
```

Software Engineering Connections

1. Resource Allocation

```
function allocateDevResources(sprints, projects, developers) {
  // Problem: Assign developers to projects to maximize delivered value
  // Constraints: Each dev works one project, limited time

  const maxValue = {};

  function backtrack(devIndex, assignments, value) {
    if (devIndex === developers.length) {
      maxValue.current = Math.max(maxValue.current || 0, value);
      maxValue.assignment = [...assignments];
      return;
    }

    for (let project = 0; project < projects.length; project++) {
      const dev = developers[devIndex];
      const proj = projects[project];
      const capacity = dev.productivity * sprints;
      const contribution = Math.min(capacity, proj.work) * proj.priority;

      assignments.push({ dev: dev.name, project: proj.name });
      backtrack(devIndex + 1, assignments, value + contribution);
      assignments.pop();
    }
  }

  backtrack(0, [], 0);
  return maxValue;
}

const devs = [
  { name: 'Alice', productivity: 10 },
  { name: 'Bob', productivity: 8 },
  { name: 'Carol', productivity: 12 }
];

const projects = [
  { name: 'Feature A', work: 40, priority: 5 },
```

```
  { name: 'Feature B', work: 30, priority: 4 },
  { name: 'Technical Debt', work: 20, priority: 3 }
];

// Finds optimal assignment maximizing value delivered
```

2. Database Query Optimization

```
function optimizeQueryPlan(query, tables) {
  // Different join orders = different costs
  // Goal: Minimize total I/O operations

  function estimateCost(joinOrder) {
    let cost = 0;
    let resultSize = tables[joinOrder[0]].rows;

    for (let i = 1; i < joinOrder.length; i++) {
      const nextTable = tables[joinOrder[i]];
      cost += resultSize * nextTable.rows;  // Cross product cost
      resultSize = Math.floor(resultSize * nextTable.selectivity);
    }

    return cost;
  }

  // Try all join orders (factorial complexity, but small in practice)
  const orders = [];
  function permute(arr, start = 0) {
    if (start === arr.length - 1) {
      orders.push([...arr]);
    } else {
      for (let i = start; i < arr.length; i++) {
        [arr[start], arr[i]] = [arr[i], arr[start]];
        permute(arr, start + 1);
        [arr[start], arr[i]] = [arr[i], arr[start]];
      }
    }
  }

  permute([...Array(tables.length).keys()]);

  return orders
    .map(order => ({
      order: order.map(i => tables[i].name),
      cost: estimateCost(order)
    }))
    .sort((a, b) => a.cost - b.cost)[0];
}

const tables = [
  { name: 'Users', rows: 1000, selectivity: 0.1 },
```

```
  { name: 'Orders', rows: 10000, selectivity: 0.2 },
  { name: 'Items', rows: 100000, selectivity: 0.05 }
];

// Finds optimal join order for query execution
```

3. Caching Strategy

```
function cachingStrategy(requests, cacheSize) {
  // Problem: Which items to keep in cache?
  // Constraints: Limited cache space
  // Objective: Minimize cache misses

  // Greedy approach: LRU (Least Recently Used)
  class LRUCache {
    constructor(size) {
      this.size = size;
      this.cache = new Map();
    }

    get(key) {
      if (this.cache.has(key)) {
        // Move to front (most recently used)
        const value = this.cache.get(key);
        this.cache.delete(key);
        this.cache.set(key, value);
        return value;
      }
      return null;
    }

    set(key, value) {
      if (this.cache.has(key)) {
        this.cache.delete(key);
      } else if (this.cache.size >= this.size) {
        // Evict least recently used
        const oldestKey = this.cache.keys().next().value;
        this.cache.delete(oldestKey);
      }
      this.cache.set(key, value);
    }
  }

  const cache = new LRUCache(cacheSize);
  let hits = 0, misses = 0;

  for (const key of requests) {
    if (!cache.get(key)) {
      misses++;
      cache.set(key, `value_${key}`);
    } else {
```

```
      hits++;
    }
  }

  return { hits, misses, hitRate: (hits / (hits + misses) * 100).toFixed(1) + '%' };
}

// Simulate: Access patterns [1, 2, 3, 2, 1, 4, 2, 3, ...]
const requests = [1, 2, 3, 2, 1, 4, 2, 3, 4, 1, 5, 2];
console.log(cachingStrategy(requests, 3));
// { hits: 6, misses: 6, hitRate: '50%' }
```

4. Load Balancing

```
function loadBalance(requests, servers) {
  // Objective: Minimize max server load (balance)

  function getLoad(assignment) {
    const loads = servers.map(s => s.capacity);
    assignment.forEach((serverIndex, requestIndex) => {
      loads[serverIndex] -= requests[requestIndex].weight;
    });
    return Math.min(...loads);  // Bottleneck (max load)
  }

  // Greedy: Always assign to least-loaded server
  const assignment = Array(requests.length).fill(0);

  for (let i = 0; i < requests.length; i++) {
    let minLoadIdx = 0;
    let minLoad = servers[0].capacity;

    for (let s = 0; s < servers.length; s++) {
      const currentLoad = servers[s].capacity -
        assignment.filter(idx => idx === s).reduce((sum, _) => sum +
requests[_].weight, 0);

      if (currentLoad < minLoad) {
        minLoad = currentLoad;
        minLoadIdx = s;
      }
    }

    assignment[i] = minLoadIdx;
  }

  return assignment;
}

const requests = [
  { id: 1, weight: 10 },
```

```
  { id: 2, weight: 15 },
  { id: 3, weight: 12 },
  { id: 4, weight: 20 }
];

const servers = [
  { name: 'Server 1', capacity: 50 },
  { name: 'Server 2', capacity: 50 }
];

// Distributes load evenly
```

5. Algorithm Selection

```
function chooseAlgorithm(dataSize) {
  const algorithms = [
    { name: 'Bubble Sort', complexity: 'O(n²)', best: 100 },
    { name: 'Merge Sort', complexity: 'O(n log n)', best: 10000 },
    { name: 'Quick Sort', complexity: 'O(n log n)', best: 5000 }
  ];

  // Choose algorithm optimized for data size
  const choice = {
    small: algorithms[0],     // n² is fine for small data
    medium: algorithms[2],    // Quick sort good average
    large: algorithms[1]      // Merge sort guaranteed O(n log n)
  };

  if (dataSize < 1000) return choice.small;
  if (dataSize < 100000) return choice.medium;
  return choice.large;
}

// Optimization = Choosing right tool for problem size
```

Common Misconceptions

❌ "Optimization always means best possible"

Wrong: Optimal usually means "best under constraints"

```
// Global optimum: Hire infinite developers, ship all features, spend nothing
// Realistic optimum: Ship most valuable features within budget and team size
```

❌ "We should always optimize for performance"

Sometimes cost/simplicity matters more:

```
// Task: Sort 100 items
// Option A: Quick sort O(n log n) - complex, 1ms
// Option B: Bubble sort O(n²) - simple, 10ms

// For 100 items: B is better (10ms is fine, code is maintainable)
// For 1M items: A is necessary (B would take hours)

// Optimization must consider actual constraints
```

❌ "Greedy always finds the best solution"

Counter-example: Activity selection with weights

```
// Greedy (by earliest end time): [Activity A, Activity C] = value 50
// Optimal: [Activity B, Activity D] = value 70

// Greedy is fast but often suboptimal
```

❌ "More iterations always finds better optimum"

Risk: Overfitting, local minima

```
// Gradient descent can get stuck in local minima
// Iterations without improvement = likely found local optimum
// More iterations won't escape without changing approach
```

Practical Mini-Exercises

- ▶ **Exercise 1: Knapsack Problem** (Click to expand)
- ▶ **Exercise 2: Server Capacity Planning** (Click to expand)
- ▶ **Exercise 3: Task Prioritization** (Click to expand)

Summary Cheat Sheet

```
// OPTIMIZATION FRAMEWORK
1. Identify objective (minimize/maximize)
2. List all constraints
3. Define decision variables
4. Choose algorithm:
   - Linear programming: Convex, deterministic
   - Greedy: Fast, often suboptimal
   - Dynamic programming: Overlapping subproblems
   - CSP/Backtracking: Discrete constraints
   - Gradient descent: Continuous smooth functions

// COMMON PATTERNS
Knapsack:       Dynamic programming
Scheduling:     Greedy or backtracking
```

```
Resource alloc: Linear program or search
Shortest path:  Dijkstra (greedy-like)
ML training:    Gradient descent
```

When to optimize:

- Constraints are tight
- Trade-offs matter (cost vs performance)
- Problem repeats (worth effort)

When NOT to optimize:

- Good enough solves problem
- Premature optimization (measure first)
- Complexity outweighs benefit

Next Steps

✅ **You've completed**: Optimization foundations
➡️ **Up next**: [15. Information Theory] - Entropy, compression, hashing, why information matters

Before moving on:

```
// Challenge: Minimize latency given: $100 budget
// - t2.micro: $10/month, 50ms latency
// - t2.small: $25/month, 30ms latency
// - t2.medium: $40/month, 10ms latency
// How many of each to achieve <20ms average latency at minimum cost?

function latencyOptimization() {
  // Your solution
}
```
