O(log n) - LOGARITHMIC TIME
The Smart Divider - Divide and Conquer Excellence!

CS 101 - Fall 2025

What is O(log n) - Logarithmic Time?

> 💡 The Smart Problem Solver
>
> **O(log n)** means the algorithm **halves the problem** with each step, creating incredibly efficient performance!
> **Real-World Analogy**:
>
> - Like playing **"20 Questions"** - each question eliminates half the possibilities
> - Guessing a number from 1-1000? "Is it > 500?" cuts it in half!
> - **32 billion items** → Only **32 steps** to find anything!

Incredible Scaling

- 1,000 items → ~10 steps
- 1,000,000 items → ~20 steps

- 1,000,000,000 items → ~30 steps
- **Mind-blowing efficiency!**

Key Insight   The algorithm **eliminates half** the remaining possibilities with each step.

Magic Question: *"Can I eliminate half the data without checking it?"*

If yes → You might achieve O(log n)!

What Makes Algorithms O(log n)?

> **i** The "Divide and Conquer" Strategy
>
> O(log n) algorithms use smart strategies to avoid checking most of the data.

Binary Search - The Classic

```python
def binary_search(sorted_array, target):
    left = 0
    right = len(sorted_array) - 1
    steps = 0

    while left <= right:
        steps += 1
        mid = (left + right) // 2

        if sorted_array[mid] == target:
            return mid, steps  # Found it!
        elif sorted_array[mid] < target:
            left = mid + 1     # Search right half
        else:
            right = mid - 1    # Search left half

    return -1, steps  # Not found

# Example: Find 7 in [1, 3, 5, 7, 9, 11, 13, 15]
# Step 1: Check middle (7) - Found it!
# Only 1 step for 8 items!
search_space = [i for i in range(16) if i % 2 ==1]
value_to_find = 7
print(f"Search space = {search_space}")
print(f"Value to find = {value_to_find}")
pos, steps = binary_search(search_space,7)
print(f"Position where found = {pos}, steps = {steps}")
# Explain the following
# binary_search([i for i in range(100)],4000)
```

Why O(log n)?

- Each step eliminates half the remaining items
- log (n) = number of times you can halve n

- Incredibly efficient!

Tree Operations

```python
# Binary Search Tree operations
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def tree_search(root, target):
    steps = 0
    current = root

    while current:
        steps += 1
        if current.value == target:
            return True, steps  # Found!
        elif target < current.value:
            current = current.left   # Go left
        else:
            current = current.right  # Go right

    return False, steps  # Not found

# Each step eliminates half the tree!
# Perfectly balanced tree: O(log n) guaranteed
```

The Tree Advantage:

- Data is pre-organized for smart searching
- Never need to check more than tree height
- Height   log (n) for balanced trees

The Tree Search In Action

> **!** Important
>
> To use the tree search, a balanced tree first must be created from the data. This may take more time initially, but the search itself is fast. Each additional search will also be time efficient.

```python
# Binary Search Tree operations
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def tree_search(root, target):
    steps = 0
    current = root

    while current:
        steps += 1
        if current.value == target:
            return True, steps   # Found!
        elif target < current.value:
            current = current.left    # Go left
        else:
            current = current.right   # Go right

    return False, steps   # Not found

# Each step eliminates half the tree!
# Perfectly balanced tree: O(log n) guaranteed

def insert_node(root, value):
    """Insert a value into the binary search tree"""
    if root is None:
        return TreeNode(value)

    if value < root.value:
        root.left = insert_node(root.left, value)
    else:
        root.right = insert_node(root.right, value)
```

```python
        return root

def create_balanced_tree(values):
    """Create a balanced BST from a sorted list"""
    if not values:
        return None

    mid = len(values) // 2
    root = TreeNode(values[mid])

    root.left = create_balanced_tree(values[:mid])
    root.right = create_balanced_tree(values[mid + 1:])

    return root

def print_tree_inorder(root):
    """Print tree values in order (for verification)"""
    if root:
        print_tree_inorder(root.left)
        print(root.value, end=" ")
        print_tree_inorder(root.right)

# Demonstration of O(log n) search
if __name__ == "__main__":
    import math

    print("=== Binary Search Tree O(log n) Demonstration ===\n")

    # Create a balanced tree with values 1-15
    values = list(range(1, 16))  # [1, 2, 3, ..., 15]
    root = create_balanced_tree(values)

    print("Created balanced BST with values:", values)
    print("Tree structure (in-order traversal):", end=" ")
    print_tree_inorder(root)
    print("\n")

    # Test searches for different values
    test_values = [1, 5, 8, 12, 15, 20]  # Include one that doesn't exist

    print("Search Results:")
    print("-" * 50)
```

```python
    print(f"{'Value':<6} {'Found':<7} {'Steps':<6} {'Expected Max Steps'}")
    print("-" * 50)

    n = len(values)
    expected_max_steps = math.ceil(math.log2(n)) + 1

    for target in test_values:
        found, steps = tree_search(root, target)
        print(f"{target:<6} {'Yes' if found else 'No':<7} {steps:<6} {expected_max_steps}")

    print("-" * 50)
    print(f"\nFor a balanced BST with {n} nodes:")
    print(f"Maximum expected steps: {expected_max_steps} ( log ({n}) + 1)")
    print(f"This demonstrates O(log n) time complexity!")

    # Demonstrate with larger tree
    print("\n=== Larger Tree Demonstration ===")
    large_values = list(range(1, 1001))  # 1000 values
    large_root = create_balanced_tree(large_values)

    # Test a few searches
    large_test_values = [1, 500, 999, 1500]  # Include one that doesn't exist
    n_large = len(large_values)
    expected_max_large = math.ceil(math.log2(n_large)) + 1

    print(f"\nTesting with {n_large} nodes:")
    print(f"{'Value':<6} {'Found':<7} {'Steps':<6} {'Expected Max Steps'}")
    print("-" * 50)

    for target in large_test_values:
        found, steps = tree_search(large_root, target)
        print(f"{target:<6} {'Yes' if found else 'No':<7} {steps:<6} {expected_max_large}")

    print("-" * 50)
    print(f"Maximum expected steps for {n_large} nodes: {expected_max_large} ( log ({n_large}
    print("Notice how the steps remain very small even with 1000 nodes!")
```

Interactive O(log n) Binary Search Demo

> **i** Watch the Halving Magic!
>
> See how binary search eliminates half the possibilities with each step. Try finding different numbers!

Python O(log n) Examples - The Efficient Ones!

Binary Search Implementation

```python
import bisect  # Python's binary search module

# Manual binary search
def binary_search_manual(arr, target):
    left, right = 0, len(arr) - 1
    steps = 0

    while left <= right:
        steps += 1
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid, steps
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1, steps

# Using Python's bisect module (optimized)
def binary_search_builtin(arr, target):
    pos = bisect.bisect_left(arr, target)
    if pos < len(arr) and arr[pos] == target:
        return pos
    return -1

# Performance comparison
numbers = list(range(0, 100000, 7))  # 0, 7, 14, 21, ...
target = 9961
```

```python
# Manual version
position, steps = binary_search_manual(numbers, target)
print(f"Manual: Found at {position} in {steps} steps")

# Built-in version (also O(log n))
position = binary_search_builtin(numbers, target)
print(f"Built-in: Found at {position}")
```

Heap Operations

```python
import heapq

# Min-heap operations - all O(log n)
heap = []

# Insert elements - O(log n) each
for value in [64, 34, 25, 12, 22, 11, 90]:
    heapq.heappush(heap, value)      # O(log n)
    print(f"Inserted {value}, heap: {heap}")

# Extract minimum - O(log n)
while heap:
    min_value = heapq.heappop(heap)  # O(log n)
    print(f"Removed {min_value}, remaining: {heap}")

# Why O(log n)?
# Heap is a binary tree structure
# Height = log(n)
# Insert/delete only travel up/down one path
# Path length = tree height = O(log n)

# Priority queue example
class PriorityQueue:
    def __init__(self):
        self.heap = []

    def push(self, item, priority):
        heapq.heappush(self.heap, (priority, item))  # O(log n)

    def pop(self):
        return heapq.heappop(self.heap)[1]  # O(log n)
```

> 💡 **Pro Tip**
>
> Python's `bisect` module provides highly optimized O(log n) operations for sorted lists!

O(log n) vs Other Complexities

Performance Comparison : Champion-like Qualities!

Data Size	O(1)	O(log n)	O(n)	O(n²)
10	1	3	10	100
100	1	7	100	10,000
1,000	1	10	1,000	1,000,000
1,000,000	1	20	1,000,000	1,000,000,000,000
1,000,000,000	1	30	1,000,000,000	∞ (impractical)

The O(log n) Sweet Spot:

- Almost as fast as O(1)
- Dramatically better than O(n)
- Scales beautifully with big data
- Requires organized/sorted data

When O(log n) Shines

Database Lookups!

```
# Pseudocode

# Database indexing - O(log n) lookups
# Even with millions of records!

# File system searches
# Modern filesystems use B-trees (O(log n))

# Sorted data structures
sorted_students = ["Alice", "Bob", "Charlie", ...]  # 10,000 students
import bisect

# O(log n) - incredibly fast!
position = bisect.bisect_left(sorted_students, "Emma")
```

```
# Compare with O(n) linear search
position = sorted_students.index("Emma")   # Much slower!

# Geographic searches (quad-trees)
# GPS navigation systems
# Image processing (pyramid algorithms)
# Game development (spatial partitioning)

# The secret: Smart data organization
# pays huge dividends in search speed!
```

Real-World O(log n) Applications

> 💡 Where You Use O(log n) Every Day!
>
> O(log n) algorithms power many systems you interact with daily.

Tech You Use Daily

```
# Pseudocode

# Database queries with indexes
# When you search your emails, photos, contacts
SELECT * FROM emails WHERE subject LIKE '%meeting%'
# Database uses B-tree index: O(log n)

# Autocomplete systems
def autocomplete(prefix, word_list):
    # Binary search to find start position
    start = bisect.bisect_left(word_list, prefix)  # O(log n)
    results = []

    for i in range(start, len(word_list)):
        if word_list[i].startswith(prefix):
            results.append(word_list[i])
        else:
            break
    return results

# Version control systems (Git)
# Git uses binary search to find bugs
```

```
git bisect start       # Start binary search
git bisect bad         # Current version has bug
git bisect good v1.0   # v1.0 was good
# Git automatically finds the problematic commit!
```

Behind-the-Scenes Magic

```python
# Pseudocode

# Memory management
# Operating systems use balanced trees
# to track free memory blocks

# Network routing
# Internet routers use prefix trees
# to find optimal paths: O(log n)

# Graphics and gaming
# Collision detection uses spatial trees
# Ray tracing uses BVH trees

# Machine learning
# Decision trees make predictions
# in O(log n) time

# Example: Simple decision tree
def classify_student_performance(hours_studied):
    if hours_studied >= 10:       # O(1) decision
        if hours_studied >= 15:   # O(1) decision
            return "Excellent"    # Total: O(log n) depth
        return "Good"
    else:
        if hours_studied >= 5:
            return "Average"
        return "Needs Improvement"

# Tree height determines performance
# Balanced tree = O(log n) predictions
```

Partner Activity: The Power of Divide and Conquer!

Work with a partner to discover the incredible scaling properties of logarithmic algorithms.

Increase and decrease the search space to experiment. To start, look for *BEGIN YOUR EXPERIMENTS HERE!* in the code.

Experiment 1: Binary Search Scaling Test

```python
import time
import bisect
import math


def create_sorted_data(size):
    # Create sorted data for testing
    return list(range(0, size * 2, 2))  # [0, 2, 4, 6, 8, ...]


def binary_search_manual(arr, target):
    left, right = 0, len(arr) - 1
    steps = 0

    while left <= right:
        steps += 1
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid, steps
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1, steps

# Partner A: Test sizes [1000, 2000, 4000]
# Partner B: Test sizes [8000, 16000, 32000]
def test_logarithmic_scaling(sizes):
    results = []

    for size in sizes:
        data = create_sorted_data(size)
        target = data[-1]  # Search for last element (worst case)
```

```python
        # Manual binary search with step counting
        start = time.time()
        pos, steps = binary_search_manual(data, target)
        manual_time = time.time() - start

        # Built-in binary search
        start = time.time()
        builtin_pos = bisect.bisect_left(data, target)
        builtin_time = time.time() - start

        theoretical_steps = math.ceil(math.log2(size))

        results.append({
            'size': size,
            'actual_steps': steps,
            'theoretical_steps': theoretical_steps,
            'manual_time': manual_time,
            'builtin_time': builtin_time
        })

        print(f"Size {size:5d}: {steps:2d} steps (theory: {theoretical_steps:2d}), {manual_t:

    return results


# ---------------------------
# BEGIN YOUR EXPERIMENTS HERE!
# Your partner assignment:
my_sizes = [1000, 2000, 4000]  # Change based on assignment
results = test_logarithmic_scaling(my_sizes)

# Partner Discussion:
# 1. How close were actual steps to theoretical log (n)?
# 2. What happened to search time as size doubled?
# 3. Compare with your partner's larger sizes - what pattern emerges?
```

Experiment 2: Heap vs Linear Priority Queue

```python
import heapq
import time
import random
```

```python
class LinearPriorityQueue:
    """Naive O(n) priority queue for comparison"""
    def __init__(self):
        self.items = []

    def push(self, item, priority):
        self.items.append((priority, item))

    def pop(self):
        if not self.items:
            return None
        # Find minimum priority (O(n))
        min_idx = min(range(len(self.items)), key=lambda i: self.items[i][0])
        return self.items.pop(min_idx)[1]

def compare_priority_queues(num_operations):
    # Generate random tasks with priorities
    tasks = [(random.randint(1, 100), f"Task_{i}") for i in range(num_operations)]

    # Test heap-based priority queue O(log n)
    heap_pq = []
    start = time.time()

    for priority, task in tasks:
        heapq.heappush(heap_pq, (priority, task))

    processed_heap = []
    while heap_pq:
        processed_heap.append(heapq.heappop(heap_pq)[1])

    heap_time = time.time() - start

    # Test linear priority queue O(n)
    linear_pq = LinearPriorityQueue()
    start = time.time()

    for priority, task in tasks:
        linear_pq.push(task, priority)

    processed_linear = []
    while linear_pq.items:
        processed_linear.append(linear_pq.pop())
```

```
    linear_time = time.time() - start

    return heap_time, linear_time


# ----------------------------
# BEGIN YOUR EXPERIMENTS HERE!
# Partner A: Test with 500 operations
# Partner B: Test with 2000 operations
my_operations = 500  # Change based on assignment

heap_time, linear_time = compare_priority_queues(my_operations)

print(f"Operations: {my_operations}")
print(f"Heap PQ (O(log n)): {heap_time:.4f}s")
print(f"Linear PQ (O(n)):   {linear_time:.4f}s")
print(f"Heap is {linear_time/heap_time:.1f}x faster!")

# Partner Discussion:
# - How did the performance gap change with more operations?
# - What would happen with 10,000 operations?
```

Group Activity: Logarithmic Thinking Challenge!

Group Problem-Solving: When to Use O(log n)!

> **Important**
>
> **Solve these real-world scenarios using logarithmic thinking!**
> **Option 1: Library Book System** You are designing a system for a library with
> 100,000 books. - Books are sorted by ISBN number - Students need to find books quickly
> - New books are added daily
> **Questions:** 1. How would you implement book lookup? (No code necessary, explain
> steps) 2. What's the maximum number of steps to find any book? 3. How would this
> scale to 1,000,000 books?
> **Option 2: Student Grade Ranking** Your school wants to rank 5,000 students by
> GPA. - GPAs range from 0.0 to 4.0 - Need to quickly find a student's rank - Rankings
> update when grades change
> **Questions:** 1. How would you store the data for fast ranking lookup? (No code nec-
> essary, explain steps) 2. How many steps to find where a 3.7 GPA ranks? 3. What

> happens when a student's GPA changes?

Your Turn: Experience O(log n) Power!

> 💡 Individual Exploration: Feel the Logarithmic Magic!
>
> After group problem-solving, try these hands-on exercises to experience O(log n) efficiency.

Exercise 1: Search Race

```python
import time
import bisect
import random

# Create test data
size = 100000
sorted_data = sorted([random.randint(1, 1000000) for _ in range(size)])
target = sorted_data[size // 2]  # Middle element

# Race 1: Linear search O(n)
start = time.time()
linear_pos = -1
for i, value in enumerate(sorted_data):
    if value == target:
        linear_pos = i
        break
linear_time = time.time() - start

# Race 2: Binary search O(log n)
start = time.time()
binary_pos = bisect.bisect_left(sorted_data, target)
binary_time = time.time() - start + 0.001 # add error

print(f"Linear search (O(n)):  {linear_time:.6f}s")
print(f"Binary search (O(log n)): {binary_time:.6f}s")
print(f"Binary search is {linear_time/binary_time:.0f}x faster!")

# Try with different sizes
# Notice how binary search stays fast
# while linear search gets slower
```

Exercise 2: Heap Priority Queue

```python
import heapq
import time

# Simulate a hospital emergency room
# Priority queue: lower number = higher priority
emergency_room = []

# Add patients with priorities
patients = [
    (1, "Heart Attack"),     # Highest priority
    (5, "Broken Arm"),
    (2, "Severe Bleeding"),
    (8, "Routine Checkup"), # Lowest priority
    (3, "Chest Pain"),
    (7, "Headache"),
    (1, "Stroke"),           # Also highest priority
]

print("Adding patients to emergency queue:")
for priority, condition in patients:
    heapq.heappush(emergency_room, (priority, condition))  # O(log n)
    print(f"Added: {condition} (Priority {priority})")

print("\nTreating patients in priority order:")
while emergency_room:
    priority, condition = heapq.heappop(emergency_room)  # O(log n)
    print(f"Treating: {condition} (Priority {priority})")

# Each operation is O(log n)
# Total time: O(n log n) for n operations
# Much better than sorting repeatedly: O(n² log n)!
```

If you want to know more about how the Heap Priority Queue code works, check out the Supplemental Slides 10_heap_sorting_slides

The Math Behind O(log n)

> **i** Understanding Logarithms
>
> Don't worry - the math is simpler than you think!

What is log (n)?

```python
# log (n) = "How many times can I divide n by 2?"

def calculate_steps(n):
    steps = 0
    while n > 1:
        n = n // 2   # Divide by 2
        steps += 1
    return steps


# Examples:
print(f"log (8)   {calculate_steps(8)}")    # 3 steps
print(f"log (16)  {calculate_steps(16)}")   # 4 steps
print(f"log (1024)  {calculate_steps(1024)}")  # 10 steps


# Binary search does exactly this!
# Each step eliminates half the data
# Until only 1 item remains

# Real logarithms:
import math
print(f"Actual log (1024) = {math.log2(1024)}")   # 10.0
print(f"Actual log (1000000) = {math.log2(1000000):.1f}")   # ~20
```

Why Logarithms are Magical

```python
# The incredible scaling of O(log n)
import math

sizes = [10, 100, 1000, 10000, 100000, 1000000]

print("Size\t\tO(n) steps\tO(log n) steps")
print("-" * 45)
for n in sizes:
```

```
    linear_steps = n
    log_steps = math.ceil(math.log2(n))
    speedup = linear_steps / log_steps

    print(f"{n:,}\t\t{linear_steps:,}\t\t{log_steps}")

# Output shows the dramatic difference!
# Size          O(n) steps    O(log n) steps
# 10            10            4
# 100           100           7
# 1,000         1,000         10
# 10,000        10,000        14
# 100,000       100,000       17
# 1,000,000     1,000,000     20

# Notice: O(log n) barely increases!
```

Achieving O(log n) - Requirements and Trade-offs

> **!** The O(log n) Prerequisites
>
> O(log n) isn't magic - it requires smart data organization!

What You Need for O(log n)

```
# 1. Sorted/Organized Data
unsorted = [64, 34, 25, 12, 22, 11, 90]
# Can't use binary search!

sorted_data = [11, 12, 22, 25, 34, 64, 90]
# Perfect for binary search!

# 2. Tree Structure
# Balanced binary search tree
# Heap (for priority operations)
# B-trees (for databases)

# 3. Divide-and-conquer opportunity
# Can you eliminate half the possibilities?
# If yes, O(log n) might be possible
```

```python
# Example: Finding square root (Newton's method)
def sqrt_binary_search(n, precision=0.001):
    low, high = 0, n
    while high - low > precision:
        mid = (low + high) / 2
        if mid * mid > n:
            high = mid  # Too big, search lower half
        else:
            low = mid   # Too small, search upper half
    return (low + high) / 2
```

The Trade-offs

```python
# Advantage: Incredible search speed
# Disadvantage: Must maintain organization

# Example: Maintaining a sorted list
sorted_scores = [65, 72, 78, 85, 92]

# Insert new score - O(n) to maintain order!
def insert_sorted(sorted_list, value):
    import bisect
    pos = bisect.bisect_left(sorted_list, value)  # O(log n) to find
    sorted_list.insert(pos, value)                # O(n) to shift!
    return sorted_list

# Better: Use a balanced tree or heap
import heapq

# For min/max operations, use heap
heap = [65, 72, 78, 85, 92]
heapq.heapify(heap)              # O(n) once
heapq.heappush(heap, 80)         # O(log n) always!
min_score = heapq.heappop(heap)  # O(log n) always!

# Decision guide:
# - Frequent searches, rare updates → Use sorted list
# - Frequent updates → Use heap or balanced tree
# - Need both → Use advanced data structures (B-trees)
```

Advanced O(log n) Patterns

Divide and Conquer Algorithms

```python
# Merge sort - O(n log n)
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])     # Recursive divide
    right = merge_sort(arr[mid:])    # Recursive divide

    return merge(left, right)        # O(n) merge

# Why O(n log n)?
# - log n levels of recursion (divide by 2 each time)
# - O(n) work at each level
# - Total: O(n) × O(log n) = O(n log n)

# Binary exponentiation - O(log n)
def power_fast(base, exp):
    if exp == 0:
        return 1
    if exp % 2 == 0:
        half = power_fast(base, exp // 2)
        return half * half
    else:
        return base * power_fast(base, exp - 1)

# Calculate 2^100 in only 7 steps instead of 100!
```

Tree Traversal Patterns

```python
# Binary search tree operations
class BST:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    def insert(self, value):        # O(log n) average
```

```python
        if value < self.value:
            if self.left:
                self.left.insert(value)
            else:
                self.left = BST(value)
        else:
            if self.right:
                self.right.insert(value)
            else:
                self.right = BST(value)

    def search(self, value):         # O(log n) average
        if value == self.value:
            return True
        elif value < self.value and self.left:
            return self.left.search(value)
        elif value > self.value and self.right:
            return self.right.search(value)
        return False

# Segment trees for range queries - O(log n)
# Fenwick trees for cumulative sums - O(log n)
# Trie trees for string operations - O(log n) depth
```

Summary: O(log n) - The Scaling Superstar!

> 💡 Key Takeaways
>
> **O(log n) - Logarithmic Time** is the sweet spot between O(1) and O(n)!

What Makes O(log n) Special -   **Incredible scaling** - handles billions of items easily - **Smart elimination** - halves the problem each step -   **Requires organization** - data must be structured -   **Great trade-off** - almost as fast as O(1), much better than O(n)

Python O(log n) Champions: - Binary search: `bisect` module - Heap operations: `heapq` module - Tree operations: balanced trees - Database queries: indexed lookups

Programming Wisdom

```python
# When to choose O(log n):
```

```
# For searching large sorted datasets
import bisect
position = bisect.bisect_left(sorted_data, target)  # O(log n)

# For priority queues
import heapq
heapq.heappush(heap, item)     # O(log n)
priority_item = heapq.heappop(heap)  # O(log n)

# For maintaining sorted order with frequent updates
# Use balanced trees or heaps

# Remember the trade-off:
# - O(1): Instant but needs hash tables
# - O(log n): Nearly instant, needs sorted data
# - O(n): Acceptable for small data, no requirements

# Choose based on your data size and update patterns!
```

Next: Exploring O(2^n) - The Exponential Challenge!

> **i** Coming Up Next
>
> **O(2^n) - Exponential Time** * Recursive algorithms and the combinatorial explosion * When algorithms become impractically slow * Interactive Fibonacci and subset generation demos * Dynamic programming to the rescue * Why some problems are inherently difficult
>
> **Questions to Think About:** * When does recursion become dangerous? * How can we optimize exponential algorithms? * What problems are fundamentally hard to solve? * When should we accept "good enough" solutions?

Ready to explore the extreme end of algorithm complexity?