CS 201

CS 201: Class Notes

Comprehensive Lecture Notes & Course Companion

Professor: Dr. Alex Morgan | **Term:** Fall 2026 | Department of Computer Science

Contents

Lecture 1: Introduction to Algorithm Analysis 2

Lecture 2: Dynamic Programming & Optimization 3

LECTURE 1 September 2, 2026

Introduction to Algorithm Analysis

Welcome to CS 201! In this first lecture, we set the foundation for computational complexity analysis, mathematical modeling of program performance, and asymptotic notation.

[DEF] Definition 1 (Algorithm)

An **algorithm** is a finite, well-defined sequence of step-by-step computational instructions that transforms an input into a desired output.

When analyzing algorithms, we evaluate performance across two main resource axes:

1. **Time Complexity**: How the execution runtime scales as the input size n approaches infinity.
2. **Space Complexity**: Memory auxiliary consumption required by the data structures during execution.

[THM] Theorem 1 (Master Theorem for Divide-and-Conquer)

Let $T(n)$ be a recurrence relation defined by $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$. If $f(n) = \Theta(n^c)$, then:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } c < \log_b a \\ \Theta(n^c \log n) & \text{if } c = \log_b a \\ \Theta(n^c) & \text{if } c > \log_b a \end{cases}$$

Proof: By constructing a recurrence tree of depth $\log_b n$, at level i there are a^i subproblems, each of size n/b^i. Summing work across all levels yields the geometric series solution above. ■

[EX] Example 1 (Binary Search Analysis)

In Binary Search, $a = 1, b = 2$, and $f(n) = \Theta(1)$. Since $c = 0$ and $\log_2 1 = 0$, Case 2 of the Master Theorem applies:

$$T(n) = \Theta(\log n)$$

[!] Warning

Master Theorem does not apply if $f(n)$ is not polynomial (e.g., $f(n) = 2^n$) or if $a < 1$.

[KEY] Key Takeaways

- Big-O provides an asymptotic upper bound on growth rate.
- Master Theorem simplifies divide-and-conquer recurrences.
- Always consider both average-case and worst-case bounds.

LECTURE 2 September 7, 2026

Dynamic Programming & Optimization

Dynamic Programming (DP) is an algorithmic technique for solving optimization problems by breaking them down into simpler overlapping subproblems.

[DEF] Definition 2 (Optimal Substructure)

A problem exhibits **optimal substructure** if an optimal solution to the problem contains optimal solutions to its subproblems.

Python Implementation: 0/1 Knapsack Bottom-Up

```python
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(values[i-1] + dp[i-1][w - weights[i
                    -1]], dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][capacity]
```

[*] Pro Tip

When designing DP state transitions, start by writing down the mathematical recurrence relation before touching code.

[EXR] Exercise 1 (Fibonacci DP Space Optimization)

Reduce the space complexity of bottom-up Fibonacci calculation from $O(n)$ to $O(1)$.

Solution:

Instead of storing an array of size n, track only the last two state variables (a and b) and update them iteratively in a loop:

$$(a, b) \leftarrow (b, a + b)$$