

OFFICIAL COURSE STUDY GUIDE

MIT 6.006: Introduction to Algorithms

The MIT algorithms course: data structures, graphs, and complexity.

COMPUTER SCIENCE · ADVANCED · 4 MODULES · 35 HR 20 MIN

Course by MIT OpenCourseWare · Curated and published by MegaCourse

MEGACOURSE.APP

MIT 6.006: Introduction to Algorithms: The Complete Study Guide

2026 Edition

Published by MegaCourse · megacourse.app

Learn Anything. Achieve Everything.

MegaCourse curates the world's best free courses into structured learning paths with study guides, practice material, and certificates. This guide accompanies the MegaCourse course "MIT 6.006: Introduction to Algorithms", taught by MIT OpenCourseWare. It is designed to be read alongside the video lessons and to serve as a standalone reference for review.

Study guide text copyright © 2026 MegaCourse. Course video content remains the property of its creators. You may print this guide for personal study. Redistribution or resale is not permitted.

Enroll, track progress, and earn your certificate at

megacourse.app/courses/mit-6-006-introduction-to-algorithms

Contents

<i>About This Course</i>	4
CHAPTER 1 Algorithms and Data Structures Fundamentals	6
CHAPTER 2 Advanced Data Structures and Graph Algorithms	14
CHAPTER 3 Shortest Paths and Dynamic Programming	20
CHAPTER 4 Advanced Dynamic Programming and Computational Complexity	26
<i>Glossary</i>	32
<i>Answer Key</i>	35
<i>Where to Go Next</i>	39

INTRODUCTION

About This Course

Welcome to MIT 6.006: Introduction to Algorithms. This course provides a comprehensive foundation in the design and analysis of algorithms. You will explore core data structures, graph algorithms, and computational complexity.

Algorithms are fundamental to computer science. They provide the systematic methods used to solve problems efficiently across all areas of computing. Understanding algorithms allows you to build robust and performant software systems.

This study guide complements the acclaimed MIT OpenCourseWare video lectures. MegaCourse has organized the material into a structured learning path, ensuring a focused and effective educational experience. This approach helps you master complex concepts through clear progression and practical application.

Course Facts

FORMAT	4 modules · 32 lessons
VIDEO INSTRUCTION	About 35 hr 20 min
LEVEL	Advanced
CATEGORY	Computer Science
INSTRUCTOR	MIT OpenCourseWare
CERTIFICATE	Included with course completion on MegaCourse
COURSE HOME	megacourse.app/courses/mit-6-006-introduction-to-algorithms

What You Will Learn

- Analyze algorithms with asymptotic (Big-O) notation.
- Master core data structures: hashing, trees, heaps, and graphs.
- Apply graph search, shortest paths, and dynamic programming.
- Reason about correctness and computational tractability.

Who This Course Is For

This course is designed for individuals seeking a rigorous introduction to algorithms. It serves computer science students, software developers, and anyone aiming to deepen their understanding of efficient

computation.

Before beginning, you should have programming experience. A basic understanding of discrete mathematics is also recommended to fully benefit from the course material.

Before You Start

- Programming experience is required.
- Discrete math basics are recommended.

How to Use This Guide

This guide is your companion to the MIT 6.006 video lectures. The course is structured into 4 modules and 32 lessons, totaling approximately 35 hours and 20 minutes of instruction. We recommend watching the video lectures for each lesson, then reviewing the corresponding sections in this guide.

Use this guide to reinforce your understanding of the concepts presented in the videos. The included exercises and review questions are designed to test your comprehension and application of algorithmic principles. The glossary defines key terms, helping you solidify your knowledge.

Course Structure

- **Module 1:** Section 1 (8 lessons)
- **Module 2:** Section 2 (8 lessons)
- **Module 3:** Section 3 (8 lessons)
- **Module 4:** Section 4 (8 lessons)

CHAPTER 1

Algorithms and Data Structures Fundamentals

This chapter introduces fundamental concepts in algorithms and data structures. It establishes a rigorous framework for understanding how computer programs solve problems efficiently. You will learn what an algorithm is, how to analyze its performance using asymptotic notation, and why efficient data organization is critical. We cover basic data structures like dynamic arrays and hash tables, along with their operational characteristics. Understanding these foundational topics is essential for designing and implementing high-performance software solutions.

The material here forms the bedrock for all subsequent algorithm studies. Mastering these concepts will enable you to evaluate and select appropriate tools for various computational challenges. We will move from abstract definitions to concrete analysis techniques and practical data structure implementations.

Algorithms, Computation, and Asymptotic Analysis

Covers: 1. Algorithms and Computation (video, 46 min) · 3. Introduction to Algorithms - Problem Session 1: Asymptotic Behavior of Functions and Double-ended... (video, 1 hr 27 min)

What is an Algorithm?

An algorithm is a well-defined computational procedure that takes some value or set of values as input and produces some value or set of values as output. It is a sequence of computational steps that transform the input into the output. Algorithms must be finite, unambiguous, and effective. They must terminate after a finite number of steps, each step must be precisely defined, and each step must be practically executable.

The RAM Model of Computation

To analyze algorithms consistently, we need a model of computation. The Random Access Machine (RAM) model is standard. In this model, we assume:

- Each elementary operation (arithmetic, logical, data movement, memory access) takes a fixed amount of time, say one unit.
- Memory is an array of cells, each holding a word of data. Accessing any memory cell takes one unit of time.
- The size of a word is fixed, typically large enough to hold an integer or a memory address.

This model simplifies analysis by abstracting away hardware specifics. We focus on the number of operations, not their exact duration.

Analyzing Algorithm Efficiency

Algorithm analysis primarily concerns efficiency, specifically time complexity and space complexity. Time complexity measures the number of elementary operations an algorithm performs as a function of its input size. Space complexity measures the amount of memory it uses. We are interested in how these grow as the input size increases.

Asymptotic Notation

Asymptotic notation describes the limiting behavior of a function as its input tends towards infinity. It helps us classify algorithms based on their growth rates, ignoring constant factors and lower-order terms that become insignificant for large inputs.

Big-O Notation (O)

Big-O notation provides an *upper bound* for the growth rate of a function. We say $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. This means that for sufficiently large n , $f(n)$ grows no faster than $g(n)$ multiplied by some constant. Big-O is most commonly used to describe the worst-case performance of an algorithm.

Big-Omega Notation (Ω)

Big-Omega notation provides a *lower bound*. We say $f(n)$ is $\Omega(g(n))$ if there exist positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$. This means that for sufficiently large n , $f(n)$ grows at least as fast as $g(n)$ multiplied by some constant.

Big-Theta Notation (Θ)

Big-Theta notation provides a *tight bound*. We say $f(n)$ is $\Theta(g(n))$ if there exist positive constants c_1 , c_2 , and n_0 such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$. This means $f(n)$ grows at the same rate as $g(n)$, both as an upper and lower bound.

Common Growth Rates

Understanding common growth rates is crucial:

- $O(1)$: Constant time. The number of operations does not depend on the input size.
- $O(\log N)$: Logarithmic time. Operations decrease by a constant factor with each step.
- $O(N)$: Linear time. Operations grow proportionally to the input size.
- $O(N \log N)$: "Linearithmic" time. Common for efficient sorting algorithms.
- $O(N^2)$: Quadratic time. Operations grow with the square of the input size.
- $O(2^N)$: Exponential time. Very slow for even moderately sized inputs.

Analyzing Code for Big-O

To find the Big-O complexity of an algorithm, identify the operations that contribute most to its runtime as the input size grows.

Consider this Python example:

```
def sum_array(arr):
    total = 0 # O(1)
    for x in arr: # Loop runs N times
        total += x # O(1) operation inside loop
    return total # O(1)
```

The loop runs N times, and inside the loop, a constant number of operations occur. Thus, the total time complexity is $O(N)$.

Now consider nested loops:

```
def multiply_matrix(matrix_a, matrix_b):
    n = len(matrix_a)
    result = [[0 for _ in range(n)] for _ in range(n)] # O(N^2) for initialization

    for i in range(n): # Outer loop runs N times
        for j in range(n): # Middle loop runs N times
            for k in range(n): # Inner loop runs N times
                result[i][j] += matrix_a[i][k] * matrix_b[k][j] # O(1)
    return result
```

Here, three nested loops each run N times. The innermost operation executes $N * N * N = N^3$ times. The initialization of `result` takes $O(N^2)$ time. Since N^3 dominates N^2 for large N , the overall time complexity is $O(N^3)$.

Common Mistakes

A common mistake is to include constant factors or lower-order terms in the Big-O expression. For instance, an algorithm that takes $2N + 5$ steps is $O(N)$, not $O(2N)$ or $O(2N + 5)$. Another error is to confuse worst-case, average-case, and best-case analysis. Big-O typically refers to the worst-case, which provides a guarantee.

Practice Cue: Take any simple function you have written recently. Determine its time complexity using Big-O notation. Consider different input sizes and how the number of operations changes.

Data Structures: Dynamic Arrays, Sets, and Hashing

Covers: 2. Data Structures and Dynamic Arrays (video, 50 min) · 4. Sets and Sorting (video, 53 min) · 5. Hashing (video, 53 min)

Data Structures: Organizing Information

A data structure is a particular way of organizing data in a computer so that it can be accessed and modified efficiently. The choice of data structure significantly impacts an algorithm's performance. Different structures are optimized for different operations.

Arrays and Dynamic Arrays

An array is a fundamental data structure storing a fixed-size sequential collection of elements.

- **Access:** Elements are accessed directly by index in $O(1)$ time due to contiguous memory.

- **Insertion/Deletion:** Inserting or deleting an element in the middle requires shifting subsequent elements, taking $O(N)$ time in the worst case. Adding to the end is $O(1)$ if space is available.

Dynamic Arrays (Resizable Arrays)

Dynamic arrays, also known as ArrayLists or Vectors, provide flexibility by growing or shrinking as needed. When a dynamic array runs out of space, it allocates a new, larger underlying array (typically doubling its capacity), then copies all existing elements. This copying takes $O(N)$ time.

To understand dynamic array efficiency, we use *amortized analysis*. While a single `append` operation might take $O(N)$ when a resize occurs, this cost is averaged over a sequence of operations. If we perform N `append` operations starting with an empty array, the total cost for all resizes and appends is $O(N)$. This means the *amortized cost* per `append` operation is $O(1)$. The total cost for N appends is roughly $1 + 2 + 4 + \dots + N/2 + N$, which sums to approximately $2N$.

Sets: An Abstract Data Type

A set is an abstract data type (ADT) that stores a collection of unique elements, without any particular order. Key operations include:

- `add(element)` : Adds an element if not present.
- `remove(element)` : Removes an element if present.
- `contains(element)` : Checks for element membership.

Various data structures can implement a set, such as sorted arrays, balanced binary search trees, or hash tables.

Hashing and Hash Tables

Hashing is a technique used to implement dictionaries or sets efficiently. A hash table maps keys to values using a hash function.

- **Hash Function:** Takes a key and returns an integer hash code, which serves as an index into an array (the hash table). An ideal hash function is fast to compute and distributes keys uniformly.

```
def simple_hash(key, table_size):  
    return hash(key) % table_size
```

Collision Resolution

A *collision* occurs when two different keys hash to the same index. Effective collision resolution is critical.

1. Chaining

Each slot in the hash table array stores a pointer to a linked list. When a collision occurs, the new key-value pair is added to the linked list at that index. Operations like `add`, `remove`, or `contains` involve hashing the key, going to the index, and then traversing the linked list. Average case performance is $O(1)$ with good distribution; worst case is $O(N)$ if all keys collide.

2. Open Addressing

All elements are stored directly within the hash table array. When a collision occurs, we systematically "probe" for an alternative empty slot.

- **Linear Probing:** If slot $h(\text{key})$ is occupied, try $h(\text{key}) + 1$, then $h(\text{key}) + 2$, etc. (modulo table size). This can lead to *primary clustering*, where long runs of occupied slots form, increasing search times.
- **Quadratic Probing:** Probes with increasing quadratic offsets: $h(\text{key}) + 1^2$, $h(\text{key}) + 2^2$, etc. This reduces primary clustering but can suffer from *secondary clustering*.
- **Double Hashing:** Uses a second hash function $h_2(\text{key})$ to determine the step size for probing. Probes are $h(\text{key}) + h_2(\text{key})$, $h(\text{key}) + 2 \cdot h_2(\text{key})$, etc. This generally offers better performance.

Load Factor

The *load factor* is the ratio of elements (N) to table size (M): $\text{load_factor} = N / M$.

- For chaining, a load factor > 1 is acceptable.
- For open addressing, the load factor must be < 1 . Maintaining a low load factor (e.g., below 0.7 for open addressing) is crucial for keeping hash table operations close to $O(1)$ on average. When the load factor gets too high, the table is typically resized to a larger capacity, and all elements are rehashed.

Practice Cue: Design a simple hash function for strings. Consider how you would handle collisions using both chaining and linear probing for a small hash table of size 10.

Linear Sorting Algorithms

Covers: 7. 5. Linear Sorting (video, 52 min)

The Comparison Sort Lower Bound

Comparison sorts, like Merge Sort or Quick Sort, rely solely on comparing elements to determine their sorted order. A fundamental mathematical proof establishes that any comparison sort requires at least $\Omega(N \log N)$ comparisons in the worst case to sort N elements. This means $O(N \log N)$ is the theoretical minimum time complexity for this class of algorithms.

Linear Sorting Algorithms: Beyond Comparisons

Linear sorting algorithms achieve $O(N)$ time complexity by *not* relying solely on comparisons. They exploit specific properties of the input data, such as the range of key values or their digit structure. This allows them to bypass the $\Omega(N \log N)$ lower bound.

1. Counting Sort

Counting Sort efficiently sorts a collection of objects where keys are small non-negative integers.

- **Assumptions:** Input elements are integers in a specific, small range (0 to k).
- **Mechanism:**

1. Initialize a `count` array of size `k+1` to zeros, storing frequencies of numbers.
 2. Iterate input, incrementing counts.
 3. Modify `count` array to store cumulative counts, indicating final positions. `count[i]` becomes the number of elements less than or equal to `i`.
 4. Create an `output` array. Iterate through the input array *from right to left*. For each element `x`, place it into `output[count[x] - 1]`, then decrement `count[x]`. The right-to-left scan ensures stability.
- **Time Complexity:** $O(N + k)$, where N is elements and k is the range.
 - **Space Complexity:** $O(N + k)$.
 - **Stability:** Yes, it preserves the relative order of equal elements.

Example: Counting Sort Sort `arr = [4, 2, 2, 8, 3, 3, 1]` with `k = 8`.

1. `count` array (size 9) initialized to zeros.
2. After counting: `count = [0, 1, 2, 2, 1, 0, 0, 0, 1]` (e.g., two 2s, one 4).
3. After accumulating: `count = [0, 1, 3, 5, 6, 6, 6, 6, 7]` (e.g., three elements ≤ 2).
4. Populate `output` array (size 7) by iterating `arr` right-to-left:
 - `arr[6]=1` -> `output[count[1]-1] = output[0] = 1`, `count[1]` becomes 0.
 - `arr[5]=3` -> `output[count[3]-1] = output[4] = 3`, `count[3]` becomes 4.
 - ...and so on. Result: `[1, 2, 2, 3, 3, 4, 8]`

2. Radix Sort

Radix Sort sorts numbers by processing individual digits. It repeatedly applies a stable sorting algorithm (like Counting Sort) to sort numbers by each digit position, typically from the least significant digit (LSD) to the most significant digit (MSD).

- **Assumptions:** Keys are numbers with `d` digits.
- **Mechanism (LSD Radix Sort):**
 1. Determine `d`, the maximum number of digits.
 2. For each digit position `i` from 0 (LSD) to `d-1` (MSD):
 - Sort the array using a stable sort based on the digit at position `i`.
- **Time Complexity:** $O(d * (N + k))$, where `d` is number of digits, N is elements, and k is range of digit values (e.g., 10 for decimal). If `d` is constant or logarithmic, this approaches $O(N)$.
- **Space Complexity:** $O(N + k)$ from the stable sorting subroutine.

Example: Radix Sort Sort `arr = [170, 45, 75, 90, 802, 24, 2, 66]`

- **Pass 1 (Units digit):** Sort using Counting Sort based on units digit. `[170, 90, 802, 2, 24, 45, 75, 66]`
- **Pass 2 (Tens digit):** Sort using Counting Sort based on tens digit. `[802, 2, 24, 45, 66, 170, 75, 90]`

- **Pass 3 (Hundreds digit):** Sort using Counting Sort based on hundreds digit. [2, 24, 45, 66, 75, 90, 170, 802] Result: [2, 24, 45, 66, 75, 90, 170, 802]

3. Bucket Sort

Bucket Sort distributes elements into a fixed number of "buckets". Each bucket is then sorted individually, and the results are concatenated.

- **Assumptions:** Input is uniformly distributed over a range.
- **Mechanism:**
 1. Create m empty buckets.
 2. Place each input element into its corresponding bucket.
 3. Sort each non-empty bucket (e.g., using Insertion Sort).
 4. Concatenate the sorted buckets.
- **Time Complexity:** $O(N)$ on average if elements are uniformly distributed. Worst case is $O(N^2)$ if all elements fall into one bucket.
- **Space Complexity:** $O(N + M)$ where M is the number of buckets.

Common Mistakes

A frequent mistake is applying linear sorting algorithms without considering their specific assumptions. Counting Sort is inefficient for a very wide range of integer keys (large k). Bucket Sort performs poorly if data is not uniformly distributed. Always verify input characteristics before choosing a linear sort.

Practice Cue: Trace Radix Sort for a list of three-digit numbers. Pay close attention to how the stability of the inner sorting algorithm (like Counting Sort) maintains the relative order of numbers with identical digits in the current pass.

KEY TAKEAWAYS

- Algorithms are precise sequences of steps for problem-solving, analyzed using the RAM model for efficiency.
- Asymptotic notation, especially Big-O, describes an algorithm's worst-case performance growth rate for large inputs.
- Dynamic arrays offer $O(1)$ amortized time for appends by doubling capacity during resizes.
- Hash tables provide average $O(1)$ time for add, remove, and contains operations by mapping keys to array indices.
- Collision resolution techniques like chaining and open addressing are essential for hash table functionality.
- Linear sorting algorithms like Counting Sort and Radix Sort can achieve $O(N)$ time by using specific input properties, bypassing the comparison sort lower bound.

CHECK YOUR UNDERSTANDING

Answer these before moving on. Model answers are in the Answer Key at the back of this guide.

1. Explain the difference between Big-O, Big-Omega, and Big-Theta notation.
2. How does a dynamic array achieve $O(1)$ amortized time for appends, even though resizing involves copying all elements?
3. What is a collision in the context of hash tables, and name two common methods for resolving them?
4. What are the main assumptions or conditions under which linear sorting algorithms like Counting Sort or Radix Sort can achieve $O(N)$ time complexity?
5. Why is the stability of Counting Sort important when it is used as a subroutine in Radix Sort?

CHAPTER 2

Advanced Data Structures and Graph Algorithms

Chapter 2 of your MegaCourse study guide works through essential advanced data structures and fundamental graph algorithms. You will master the intricacies of self-balancing binary search trees, understand the mechanics of binary heaps, and explore the foundational graph traversal techniques of Breadth-First Search and Depth-First Search. The chapter concludes with a detailed examination of Dijkstra's algorithm for finding shortest paths in weighted graphs.

This material forms a crucial foundation for designing efficient algorithms and solving complex computational problems. Each section provides practical explanations, step-by-step procedures, and concrete examples to ensure a solid understanding, even without internet access.

Binary Search Trees and AVL Trees

Covers: 6. Binary Trees, Part 1 · 7. Binary Trees, Part 2: AVL · Problem Session 4

Binary Search Trees (BSTs) are a fundamental data structure for storing ordered data. Each node in a BST contains a key, and it adheres to the property that all keys in its left subtree are smaller than its own key, while all keys in its right subtree are larger. This property allows for efficient searching, insertion, and deletion operations. In a well-balanced BST, these operations take $O(\log n)$ time, where n is the number of nodes. However, if insertions occur in a sorted or reverse-sorted order, a BST can degenerate into a linked list, leading to $O(n)$ worst-case performance for all operations.

To overcome this limitation, self-balancing binary search trees were developed. AVL trees are among the earliest and most widely used self-balancing BSTs. An AVL tree maintains a strict balance property: for every node, the height difference between its left and right subtrees must be at most 1. This property guarantees that the tree's height remains $O(\log n)$, ensuring $O(\log n)$ performance for search, insertion, and deletion operations.

When an insertion or deletion operation violates the AVL balance property, the tree performs rotations to restore balance. There are four types of rotations: single left (RR), single right (LL), double left-right (LR), and double right-left (RL).

Single Rotations

- **Right Rotation (LL case):** Occurs when a node becomes left-heavy and its left child is also left-heavy. The left child becomes the new root of the subtree, and the original root becomes its right child.
- **Left Rotation (RR case):** Occurs when a node becomes right-heavy and its right child is also right-heavy. The right child becomes the new root of the subtree, and the original root becomes its left child.

Double Rotations

- **Left-Right Rotation (LR case):** Occurs when a node becomes left-heavy, but its left child is right-heavy. This involves a left rotation on the left child, followed by a right rotation on the original node.
- **Right-Left Rotation (RL case):** Occurs when a node becomes right-heavy, but its right child is left-heavy. This involves a right rotation on the right child, followed by a left rotation on the original node.

Understanding these rotations is key to implementing AVL trees. Practice drawing out tree insertions and deletions, manually applying the correct rotations to maintain balance. This hands-on approach will solidify your understanding of how AVL trees guarantee efficient operations even under adversarial input sequences. Problem Session 4 offers a good opportunity to reinforce these concepts through practical exercises.

Binary Heaps

Covers: 8. Binary Heaps

A binary heap is a complete binary tree that satisfies the heap property. A complete binary tree is one where all levels are fully filled, except possibly the last level, which is filled from left to right. The heap property comes in two forms: min-heap and max-heap.

- **Min-Heap Property:** The key of each node is less than or equal to the keys of its children. The smallest key is always at the root.
- **Max-Heap Property:** The key of each node is greater than or equal to the keys of its children. The largest key is always at the root.

Binary heaps are typically implemented using an array, taking advantage of the complete binary tree property. For a node at index i (0-indexed):

- Its parent is at $(i - 1) / 2$ (integer division).
- Its left child is at $2 * i + 1$.
- Its right child is at $2 * i + 2$. This array representation is space-efficient as it avoids pointers.

Key operations on a binary heap include insertion, extraction of the minimum (or maximum), and building a heap from an unsorted array.

Insertion (Insert)

To insert a new element into a min-heap, place it at the next available position in the array (the end of the heap). Then, 'bubble up' (or 'percolate up') the element by repeatedly comparing it with its parent and swapping them if the child is smaller (for a min-heap). This process continues until the heap property is restored or the element reaches the root. This operation takes $O(\log n)$ time.

Extraction (Extract-Min/Extract-Max)

To extract the minimum element from a min-heap, remove the root (which is the minimum). Replace the root with the last element in the heap array. Then, 'bubble down' (or 'percolate down') the new root by

repeatedly comparing it with its children and swapping it with the smaller child (for a min-heap). This continues until the heap property is restored. This operation also takes $O(\log n)$ time.

Building a Heap (Heapify)

Building a heap from an arbitrary array of n elements can be done in $O(n)$ time. This 'heapify' process starts from the last non-leaf node and works upwards to the root. For each node, perform a 'bubble down' operation to ensure its subtree satisfies the heap property. The efficiency comes from the fact that most nodes are close to the leaves, requiring few swaps.

Binary heaps are crucial for implementing priority queues and are the foundation of the Heapsort algorithm. Understanding the array representation and the bubble-up/bubble-down mechanics is essential. Practice tracing these operations on small arrays to see how the heap property is maintained.

Graph Traversal: BFS and DFS

Covers: 9. Breadth-First Search · 10. Depth-First Search · Quiz 1 review

Graphs are powerful data structures used to model relationships between entities. They consist of vertices (nodes) and edges (connections). Graphs can be directed or undirected, and edges can have weights or be unweighted. Common ways to represent graphs include adjacency lists and adjacency matrices.

- **Adjacency List:** An array where each index i stores a list of vertices adjacent to vertex i . This is efficient for sparse graphs (few edges).
- **Adjacency Matrix:** A $V \times V$ matrix (where V is the number of vertices) where $matrix[i][j]$ is 1 if an edge exists from i to j , and 0 otherwise. This is efficient for dense graphs (many edges).

Graph traversal algorithms systematically visit every vertex and edge in a graph. The two primary methods are Breadth-First Search (BFS) and Depth-First Search (DFS).

Breadth-First Search (BFS)

BFS explores a graph level by level. It starts at a source vertex, visits all its immediate neighbors, then all their unvisited neighbors, and so on. BFS uses a queue to manage the order of vertices to visit.

Algorithm Steps (BFS):

1. Initialize all vertices as unvisited.
2. Create a queue and enqueue the starting vertex.
3. Mark the starting vertex as visited.
4. While the queue is not empty: a. Dequeue a vertex u . b. For each unvisited neighbor v of u : i. Mark v as visited. ii. Enqueue v .

Properties of BFS:

- Finds the shortest path (in terms of number of edges) from the source to all reachable vertices in an unweighted graph.
- Explores uniformly outward from the source.

- Time complexity: $O(V + E)$, where V is the number of vertices and E is the number of edges.

Depth-First Search (DFS)

DFS explores as deeply as possible along each branch before backtracking. It uses a stack (or recursion, which implicitly uses the call stack) to keep track of vertices to visit.

Algorithm Steps (DFS):

1. Initialize all vertices as unvisited.
2. Create a stack and push the starting vertex (or call a recursive function).
3. While the stack is not empty: a. Pop a vertex u . b. If u is unvisited: i. Mark u as visited. ii. Process u . iii. For each neighbor v of u (in reverse order for consistent results with stack):
4. Push v onto the stack.

Properties of DFS:

- Useful for detecting cycles, finding connected components, topological sorting, and solving maze-like problems.
- Explores along a single path as far as possible.
- Time complexity: $O(V + E)$.

Both BFS and DFS are foundational for many graph algorithms. Practice tracing both algorithms on various graphs, noting the order in which nodes are visited. Quiz 1 review will likely cover questions related to these traversal methods.

Weighted Shortest Paths

Covers: 11. Weighted Shortest Paths

The weighted shortest path problem aims to find a path between two vertices in a weighted graph such that the sum of the weights of its edges is minimized. Unlike unweighted shortest paths found by BFS, here edge weights introduce varying 'costs' or 'distances' for traversing edges.

Dijkstra's Algorithm is a greedy algorithm that solves the single-source shortest path problem for a graph with non-negative edge weights. It maintains a set of visited vertices and an array of shortest distances found so far from the source to all other vertices.

Dijkstra's Algorithm Steps:

1. **Initialization:** For a source vertex s :
 - Set $\text{dist}[s] = 0$.
 - Set $\text{dist}[v] = \text{infinity}$ for all other vertices v .
 - Initialize a priority queue (min-heap) with all vertices, prioritizing those with smaller dist values.
2. **Iteration:** While the priority queue is not empty: a. **Extract Minimum:** Extract the vertex u with the smallest $\text{dist}[u]$ from the priority queue. This u is now considered 'settled' and its $\text{dist}[u]$ is the

final shortest path from s to u . b. **Relaxation:** For each neighbor v of u :

- If $\text{dist}[u] + \text{weight}(u, v) < \text{dist}[v]$:
- Update $\text{dist}[v] = \text{dist}[u] + \text{weight}(u, v)$.
- Update $\text{prev}[v] = u$ (to reconstruct the path).
- Update v 's priority in the priority queue (if using a mutable priority queue, otherwise re-insert and handle duplicates).

Example of Relaxation:

Suppose you are at vertex u with $\text{dist}[u] = 5$. You have an edge (u, v) with $\text{weight}(u, v) = 3$. If $\text{dist}[v]$ is currently 10, then $5 + 3 < 10$ is true. You would 'relax' the edge by updating $\text{dist}[v]$ to 8. This means a shorter path to v has been found through u .

Time Complexity:

With a binary heap as the priority queue, Dijkstra's algorithm has a time complexity of $O(E \log V)$ or $O(E + V \log V)$, where V is the number of vertices and E is the number of edges. If a Fibonacci heap is used, it can achieve $O(E + V \log V)$, which is faster for very dense graphs.

Limitations:

Dijkstra's algorithm **does not work correctly with negative edge weights**. For graphs with negative weights (but no negative cycles), algorithms like Bellman-Ford are used. Understanding Dijkstra's precise steps and the role of the priority queue is crucial. Practice tracing its execution on small weighted graphs to see how distances are iteratively updated and finalized.

KEY TAKEAWAYS

- Binary Search Trees provide efficient ordered data storage, but can degrade to $O(n)$ performance if unbalanced.
- AVL trees are self-balancing BSTs that use rotations to guarantee $O(\log n)$ height and operation times.
- Binary heaps are complete binary trees satisfying the heap property, efficiently implemented with arrays for priority queues.
- Heap operations like insertion and extraction take $O(\log n)$ time, while building a heap from an array takes $O(n)$ time.
- Breadth-First Search (BFS) explores graphs level by level, finding shortest paths in unweighted graphs.
- Depth-First Search (DFS) explores deeply before backtracking, useful for cycle detection and topological sorting.
- Dijkstra's algorithm finds single-source shortest paths in weighted graphs with non-negative edge weights using a greedy approach and a priority queue.

CHECK YOUR UNDERSTANDING

Answer these before moving on. Model answers are in the Answer Key at the back of this guide.

1. Explain why AVL trees are preferred over basic Binary Search Trees in many applications.
2. Describe the main difference between a min-heap and a max-heap, and how they are typically stored in memory.
3. When would you choose Breadth-First Search (BFS) over Depth-First Search (DFS) for a graph problem?
4. What is the primary limitation of Dijkstra's algorithm, and what kind of graph problem would require a different shortest path algorithm?
5. Briefly explain the 'relaxation' step in Dijkstra's algorithm.

CHAPTER 3

Shortest Paths and Dynamic Programming

This chapter explores fundamental algorithms for finding shortest paths in graphs and introduces the powerful technique of dynamic programming. You will learn how to compute the shortest path from a single source to all other vertices, handle graphs with negative edge weights, and find shortest paths between all pairs of vertices. The chapter then transitions to dynamic programming, a method for solving complex problems by breaking them down into simpler, overlapping subproblems.

Mastering these concepts provides essential tools for analyzing network routing, resource allocation, and sequence alignment problems. We will cover the core principles of each algorithm, walk through concrete examples, and discuss their respective strengths and limitations.

Single-Source Shortest Paths: Bellman-Ford and Dijkstra's Algorithms

Covers: Problem Session 5 · 12. Bellman-Ford · 13. Dijkstra

Finding the shortest path from a single source vertex to all other vertices in a graph is a foundational problem in computer science. Two primary algorithms address this: Bellman-Ford and Dijkstra's.

Bellman-Ford Algorithm

The Bellman-Ford algorithm computes single-source shortest paths even when edge weights are negative. It can also detect the presence of negative cycles, which would make shortest paths undefined. The algorithm works by iteratively relaxing edges. In each iteration, it updates distance estimates if a shorter path is found. This process repeats a fixed number of times.

Here are the steps:

1. **Initialization:** Set the distance to the source vertex s as 0. Set distances to all other vertices as infinity. Initialize predecessor pointers for path reconstruction.
2. **Relaxation:** Repeat $V - 1$ times (where V is the number of vertices): For every edge (u, v) with weight $w(u, v)$ in the graph, if $\text{dist}[u] + w(u, v) < \text{dist}[v]$, update $\text{dist}[v] = \text{dist}[u] + w(u, v)$ and set $\text{pred}[v] = u$.
3. **Negative Cycle Detection:** After $V - 1$ iterations, perform one more pass over all edges. If any edge (u, v) still satisfies $\text{dist}[u] + w(u, v) < \text{dist}[v]$, then a negative cycle exists and the algorithm reports this. Otherwise, the computed distances are the true shortest path distances.

The time complexity of Bellman-Ford is $O(VE)$, where V is the number of vertices and E is the number of edges. This makes it suitable for graphs with negative weights, especially when E is not too large.

Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm that finds the single-source shortest paths in graphs with non-negative edge weights. It is significantly faster than Bellman-Ford for such graphs. It maintains a set of visited vertices and a priority queue of unvisited vertices, always extracting the vertex with the smallest current shortest path estimate.

Here are the steps:

1. **Initialization:** Set the distance to the source vertex s as 0. Set distances to all other vertices as infinity. Create a min-priority queue and insert all vertices with their current distance estimates.
2. **Extraction and Relaxation:** While the priority queue is not empty:
 - Extract the vertex u with the minimum distance estimate from the priority queue.
 - Mark u as visited.
 - For each neighbor v of u :
 - If v has not been visited and $\text{dist}[u] + w(u, v) < \text{dist}[v]$, update $\text{dist}[v] = \text{dist}[u] + w(u, v)$ and update v 's priority in the queue.

The time complexity of Dijkstra's algorithm depends on the priority queue implementation. With a binary heap, it is $O(E \log V)$. With a Fibonacci heap, it can be improved to $O(E + V \log V)$. Dijkstra's algorithm fails if the graph contains negative edge weights because its greedy choice might not lead to the optimal global solution.

Example: Dijkstra's on a simple graph

Consider a graph with vertices A, B, C, D and edges: A-B (weight 1), A-C (weight 4), B-C (weight 2), B-D (weight 5), C-D (weight 1). Source is A.

1. Initialize: $\text{dist} = \{A:0, B:\text{inf}, C:\text{inf}, D:\text{inf}\}$, $\text{PQ} = \{(A,0), (B,\text{inf}), (C,\text{inf}), (D,\text{inf})\}$.
2. Extract A (dist 0).
 - Relax A-B: $\text{dist}[B] = 0 + 1 = 1$. $\text{PQ} = \{(B,1), (C,\text{inf}), (D,\text{inf})\}$.
 - Relax A-C: $\text{dist}[C] = 0 + 4 = 4$. $\text{PQ} = \{(B,1), (C,4), (D,\text{inf})\}$.
3. Extract B (dist 1).
 - Relax B-C: $\text{dist}[B] + w(B,C) = 1 + 2 = 3$. Since $3 < \text{dist}[C]$ (which is 4), update $\text{dist}[C] = 3$. $\text{PQ} = \{(C,3), (D,\text{inf})\}$.
 - Relax B-D: $\text{dist}[B] + w(B,D) = 1 + 5 = 6$. Update $\text{dist}[D] = 6$. $\text{PQ} = \{(C,3), (D,6)\}$.
4. Extract C (dist 3).
 - Relax C-D: $\text{dist}[C] + w(C,D) = 3 + 1 = 4$. Since $4 < \text{dist}[D]$ (which is 6), update $\text{dist}[D] = 4$. $\text{PQ} = \{(D,4)\}$.
5. Extract D (dist 4).

Final shortest distances from A: A:0, B:1, C:3, D:4.

All-Pairs Shortest Paths: Floyd-Warshall and Johnson's Algorithms

Covers: Problem Session 7 · 14. APSP and Johnson · Quiz 2 Review

The all-pairs shortest path (APSP) problem seeks to find the shortest path between every pair of vertices in a graph. One straightforward approach is to run a single-source shortest path algorithm, like Dijkstra's or Bellman-Ford, from each vertex. However, dedicated algorithms often offer better performance or handle specific graph properties more efficiently.

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is a dynamic programming approach to solve the APSP problem. It can handle graphs with negative edge weights, but it cannot handle negative cycles. If a negative cycle is present, the algorithm will detect it by finding a path from a vertex to itself with a negative length.

The algorithm works by considering intermediate vertices. It computes $\text{dist}[i][j]$ as the shortest path from i to j using only vertices from 1 to k as intermediate vertices. The core recurrence is:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$$

Here are the steps:

1. **Initialization:** Create a $V \times V$ distance matrix. Initialize $\text{dist}[i][j]$ with the direct edge weight $w(i, j)$ if an edge exists, 0 if $i == j$, and infinity otherwise.
2. **Iteration:** For each vertex k from 1 to V (considered as an intermediate vertex):
 - For each source vertex i from 1 to V :
 - For each destination vertex j from 1 to V :
 - Update $\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$.

The time complexity of Floyd-Warshall is $O(V^3)$. It is simple to implement and efficient for dense graphs or when V is small.

Johnson's Algorithm

Johnson's algorithm is another APSP algorithm that efficiently handles graphs with negative edge weights. It is particularly advantageous for sparse graphs, where E is much smaller than V^2 . It works by reweighting the edges so that all edge weights become non-negative, allowing Dijkstra's algorithm to be used.

Here are the steps:

1. **Add a new source:** Create a new vertex s and add zero-weight edges from s to all other vertices in the graph.
2. **Run Bellman-Ford:** Execute the Bellman-Ford algorithm from s on the modified graph. Let $h(v)$ be the shortest path distance from s to v found by Bellman-Ford. If Bellman-Ford detects a negative

cycle, then the original graph contains a negative cycle, and shortest paths are undefined. The algorithm terminates.

3. **Reweight edges:** For every edge (u, v) in the original graph with weight $w(u, v)$, compute a new reweighted edge weight $w'(u, v) = w(u, v) + h(u) - h(v)$. A key property of this reweighting is that all $w'(u, v)$ are non-negative, and shortest paths in the reweighted graph correspond to shortest paths in the original graph.
4. **Run Dijkstra's:** For each vertex u in the original graph, run Dijkstra's algorithm from u using the reweighted edge weights w' . This gives $d'(u, v)$, the shortest path distance from u to v in the reweighted graph.
5. **Convert back:** The actual shortest path distance $d(u, v)$ in the original graph is $d'(u, v) - h(u) + h(v)$.

The time complexity of Johnson's algorithm is $O(VE + V^2 \log V)$ when using a binary heap for Dijkstra's. This is often better than Floyd-Warshall for sparse graphs, where E is closer to V than V^2 .

Dynamic Programming Fundamentals and Applications

Covers: 15. Dynamic Programming, Part 1: SRTBOT, Fib, DAGs, Bowling · 16. Dynamic Programming, Part 2: LCS, LIS, Coins

Dynamic programming (DP) is a powerful algorithmic technique for solving optimization problems. It applies to problems that exhibit two key properties: optimal substructure and overlapping subproblems.

Core Concepts

- **Optimal Substructure:** An optimal solution to a problem contains optimal solutions to its subproblems. This means you can build a global optimal solution from locally optimal subproblem solutions.
- **Overlapping Subproblems:** The same subproblems are encountered and solved repeatedly. DP addresses this by storing the results of subproblems to avoid recomputing them.

There are two main approaches to dynamic programming:

- **Memoization (Top-down):** This is a recursive approach. When a subproblem is encountered, its solution is computed and stored. If the subproblem is encountered again, the stored result is returned directly. This avoids redundant computations.
- **Tabulation (Bottom-up):** This is an iterative approach. Solutions to subproblems are computed and stored in a table, typically in a specific order, starting from the smallest subproblems and building up to the main problem solution.

Classic Examples

1. **Fibonacci Sequence:** The n -th Fibonacci number $F(n)$ is defined as $F(n) = F(n-1) + F(n-2)$, with $F(0)=0$ and $F(1)=1$. A naive recursive solution recomputes $F(k)$ many times. Dynamic programming solves this efficiently.
- **Memoized (Python-like pseudocode):**

```
memo = {}
def fib(n):
    if n in memo: return memo[n]
    if n <= 1: return n
    result = fib(n-1) + fib(n-2)
    memo[n] = result
    return result
```

• **Tabulated (Python-like pseudocode):**

```
dp = [0] * (n + 1)
dp[1] = 1
for i in range(2, n + 1):
    dp[i] = dp[i-1] + dp[i-2]
return dp[n]
```

2. **Rod Cutting (SRTBOT):** Given a rod of length n and a table of prices $p[i]$ for pieces of length i , find the maximum profit obtainable by cutting the rod and selling the pieces. The optimal substructure is that the maximum profit for a rod of length n is found by considering all possible first cuts i and then adding $p[i]$ to the maximum profit for the remaining rod of length $n-i$.

```
max_profit[n] = max(p[i] + max_profit[n-i]) for i in 1..n
```

3. **Shortest Paths in Directed Acyclic Graphs (DAGs):** For DAGs, shortest paths can be found in linear time using dynamic programming. First, topologically sort the graph. Then, iterate through the vertices in topological order, relaxing their outgoing edges. This ensures that when a vertex u is processed, all paths leading to u have already been considered.
4. **Longest Common Subsequence (LCS):** Given two sequences, X and Y , find the longest subsequence common to both. A subsequence does not require consecutive elements. The DP state $LCS[i][j]$ represents the length of the LCS of $X[1..i]$ and $Y[1..j]$.
- If $X[i] == Y[j]$: $LCS[i][j] = 1 + LCS[i-1][j-1]$
 - Else: $LCS[i][j] = \max(LCS[i-1][j], LCS[i][j-1])$

This results in an $O(MN)$ algorithm, where M and N are the lengths of the sequences.

5. **Longest Increasing Subsequence (LIS):** Given a sequence of numbers, find the length of the longest subsequence such that all elements of the subsequence are sorted in increasing order. A common DP approach uses $dp[i]$ to store the length of the LIS ending at index i .

```
dp[i] = 1 + max(dp[j] for j < i if arr[j] < arr[i])
```

If no such j exists, $dp[i] = 1$. This approach has $O(N^2)$ complexity. An optimized $O(N \log N)$ solution uses a different DP state and binary search.

6. **Coin Change Problem:** This problem has two common variations:

- **Minimum number of coins:** Given a set of coin denominations and a target amount, find the minimum number of coins to make that amount. $dp[i]$ stores the minimum coins for amount i . $dp[i] = \min(1 + dp[i - \text{coin_val}])$ for all coin_val in the set.
- **Number of ways to make change:** Given a set of coin denominations and a target amount, find the number of distinct ways to make that amount. $dp[i]$ stores the number of ways for amount i . Iterate through coins, and for each coin c , $dp[j] += dp[j - c]$ for j from c to amount .

KEY TAKEAWAYS

- Bellman-Ford's algorithm computes single-source shortest paths in graphs with negative edge weights and can detect negative cycles.
- Dijkstra's algorithm is faster for single-source shortest paths but requires all edge weights to be non-negative.
- The Floyd-Warshall algorithm solves the all-pairs shortest path problem using dynamic programming in $O(V^3)$ time.
- Johnson's algorithm solves all-pairs shortest paths for graphs with negative weights by reweighting edges to be non-negative, then running Dijkstra's from each vertex.
- Dynamic programming is an algorithmic technique for problems exhibiting optimal substructure and overlapping subproblems.
- Memoization is a top-down, recursive DP approach that stores computed subproblem solutions to avoid recomputation.
- Tabulation is a bottom-up, iterative DP approach that systematically fills a table of subproblem solutions.

CHECK YOUR UNDERSTANDING

Answer these before moving on. Model answers are in the Answer Key at the back of this guide.

1. When would you use Bellman-Ford's algorithm instead of Dijkstra's algorithm for single-source shortest paths?
2. Explain the purpose of edge reweighting in Johnson's algorithm.
3. What are the two fundamental properties a problem must exhibit to be solvable by dynamic programming?
4. Describe the difference between memoization and tabulation in dynamic programming.
5. How does the presence of negative cycles affect shortest path algorithms?

CHAPTER 4

Advanced Dynamic Programming and Computational Complexity

This chapter concludes your study of algorithms by exploring advanced applications of dynamic programming and introducing the fundamental concepts of computational complexity theory. You will learn to tackle complex optimization problems using sophisticated dynamic programming techniques, including those with pseudopolynomial time complexity.

We will then shift our focus to the theoretical limits of computation, understanding what makes problems inherently hard. This includes defining the complexity classes P and NP, and grasping the significance of NP-completeness. Finally, we will review the entire course content, connecting various algorithmic paradigms and preparing you for further exploration in the field of algorithms and computer science.

Advanced Dynamic Programming Techniques

Covers: Problem Session 8 · 17. Dynamic Programming, Part 3: APSP, Parens, Piano · 18. Dynamic Programming, Part 4: Rods, Subset Sum, Pseudopolynomial

Dynamic programming (DP) is a powerful method for solving problems by breaking them into smaller, overlapping subproblems and storing the solutions to avoid recomputation. This section extends your understanding of DP to more complex scenarios, including all-pairs shortest paths, matrix chain multiplication, and problems with pseudopolynomial time complexity.

All-Pairs Shortest Path (APSP)

Finding the shortest path between every pair of vertices in a graph is known as the All-Pairs Shortest Path problem. While running Dijkstra's algorithm from each vertex works for non-negative edge weights, the **Floyd-Warshall algorithm** provides a simpler DP approach that handles negative edge weights (but no negative cycles). The core idea is to consider intermediate vertices. Let `dist[i][j]` be the shortest path from `i` to `j`. The DP recurrence is:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$$

This update is performed for all `i`, `j`, and for `k` from 1 to `V` (number of vertices), where `k` is the intermediate vertex. The outer loop iterates `k` through all possible intermediate vertices. The time complexity is $O(V^3)$.

Matrix Chain Multiplication

Given a sequence of matrices, we want to find the most efficient way to multiply them. Matrix multiplication is associative, meaning $(A * B) * C$ is the same as $A * (B * C)$, but the number of scalar multiplications can differ significantly based on the parenthesization. This problem exhibits optimal

substructure and overlapping subproblems. Let $m[i][j]$ be the minimum scalar multiplications needed to compute the product of matrices A_i through A_j . The recurrence is:

$$m[i][j] = \min(m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]) \text{ for } i \leq k < j$$

Here, p is an array where $p[i-1]$ by $p[i]$ are the dimensions of matrix A_i . The base case is $m[i][i] = 0$. This problem often requires careful indexing and loop order to build the solution table correctly.

Rod Cutting Problem

Given a rod of length n and a table of prices $p[i]$ for rods of length i , determine the maximum revenue obtainable by cutting up the rod and selling the pieces. You can make as many cuts as you want. The recurrence relation for maximum revenue $r[n]$ is:

$$r[n] = \max(p[n], r[1] + r[n-1], r[2] + r[n-2], \dots, r[n-1] + r[1])$$

This can be simplified to $r[n] = \max(p[i] + r[n-i])$ for $1 \leq i \leq n$. The base case is $r[0] = 0$. This is a classic example of a one-dimensional DP problem.

Subset Sum Problem

Given a set of non-negative integers S and a target sum T , determine if there is a subset of S that sums to T . This can be solved using a 2D DP table, $dp[i][j]$, which is true if a sum j can be formed using the first i elements of S . The recurrence is:

$$dp[i][j] = dp[i-1][j] \text{ OR } dp[i-1][j - S[i]] \text{ (if } j \geq S[i])$$

The base cases are $dp[0][0] = \text{true}$ and $dp[0][j] = \text{false}$ for $j > 0$. The time complexity is $O(n * T)$, where n is the number of elements and T is the target sum. Notice that T is not necessarily polynomial in the *number of bits* required to represent T . This leads to the concept of **pseudopolynomial time**.

Pseudopolynomial Time

An algorithm runs in pseudopolynomial time if its running time is polynomial in the *numeric value* of the input, but not necessarily in the *number of bits* required to represent the input. For Subset Sum, if T is very large, even if n is small, $n*T$ can be exponential in the input size ($\log T$ bits). Problems like Subset Sum and Knapsack are common examples. This distinction is crucial when analyzing the efficiency of algorithms, especially for optimization problems where input values can be large.

Practice: Try to formulate the DP recurrence for the Optimal Binary Search Tree problem, which aims to minimize the expected search cost for a set of keys with given probabilities.

Computational Complexity Theory

Covers: 19. Complexity

Computational complexity theory classifies problems based on the resources required to solve them. We primarily focus on time complexity, distinguishing between problems that can be solved efficiently (in polynomial time) and those that appear to be inherently hard. This section introduces the fundamental complexity classes P and NP, and the concept of NP-completeness.

Decision Problems

Complexity theory often deals with **decision problems**, which are problems with a yes/no answer. For example, 'Is there a subset of S that sums to T?' is a decision problem, while 'Find a subset of S that sums to T' is an optimization problem. Many optimization problems have corresponding decision versions.

Class P

P (Polynomial time) is the class of decision problems that can be solved by a deterministic algorithm in polynomial time. This means the running time is bounded by $O(n^k)$ for some constant k , where n is the input size. Problems in P are generally considered 'tractable' or 'efficiently solvable'. Examples include sorting, searching, finding shortest paths in a graph (e.g., Dijkstra's), and matrix multiplication.

Class NP

NP (Nondeterministic Polynomial time) is the class of decision problems for which a *given solution* (a 'certificate' or 'witness') can be *verified* in polynomial time by a deterministic algorithm. This does not mean the problem can be *solved* in polynomial time. Many important problems fall into NP, such as the Traveling Salesperson Problem (TSP) decision version (Is there a tour of length at most K ?), the Satisfiability problem (SAT), and the Subset Sum problem. If someone hands you a tour, you can quickly check its length. If someone hands you a truth assignment, you can quickly check if it satisfies a Boolean formula.

It is clear that P is a subset of NP, because if a problem can be solved in polynomial time, its solution can certainly be verified in polynomial time (by simply solving it). The big open question in computer science is whether $P = NP$.

NP-Completeness

A decision problem L is **NP-complete** if it satisfies two conditions:

1. L is in NP.
2. L is NP-hard. This means that *every problem in NP* can be reduced to L in polynomial time. A **polynomial-time reduction** is an algorithm that transforms an instance of problem A into an instance of problem B such that solving B solves A , and the transformation takes polynomial time. If you can solve an NP-complete problem in polynomial time, then you can solve *all* problems in NP in polynomial time, implying $P = NP$.

The first problem proven to be NP-complete was SAT (Cook-Levin theorem). Since then, thousands of problems have been shown to be NP-complete by reducing known NP-complete problems to them. Examples include 3-SAT, Clique, Vertex Cover, and the Hamiltonian Cycle problem.

Implications of NP-Completeness

If a problem is NP-complete, it is highly unlikely that a polynomial-time algorithm exists for it. When faced with an NP-complete problem, algorithm designers typically turn to alternative strategies:

- **Approximation algorithms:** Algorithms that find a solution close to the optimal one within a guaranteed factor.
- **Heuristics:** Algorithms that find good solutions quickly, though without guarantees of optimality or performance bounds.
- **Exponential algorithms:** Algorithms that are efficient enough for small input sizes.

Understanding complexity classes helps you identify when to spend time searching for an exact polynomial-time algorithm versus when to explore other approaches. The P versus NP question remains one of the most important unsolved problems in theoretical computer science.

Practice: Consider the Independent Set problem. Is it in P or NP? How would you verify a proposed solution for it in polynomial time?

Course Synthesis and Future Directions

Covers: Problem Session 9 · Quiz 3 Review · 20. Course Review · 21. Algorithms, Next Steps

This final section provides a comprehensive review of the entire course, consolidating the fundamental concepts you have learned and offering guidance on how to continue your algorithmic path. We will revisit core ideas, discuss problem-solving strategies, and explore avenues for further study.

Review of Core Concepts

Throughout this course, you have built a strong foundation in algorithmic thinking. We began with **asymptotic notation** (Big-O, Omega, Theta) to analyze algorithm efficiency, a critical skill for comparing different approaches. You then mastered **core data structures**, including arrays, linked lists, stacks, queues, hash tables (with collision resolution), binary search trees (including balanced trees like AVL and Red-Black trees), and heaps (priority queues).

We explored **graph algorithms**, covering representations, graph traversal (BFS, DFS), topological sort, minimum spanning trees (Prim's, Kruskal's), and shortest path algorithms (Dijkstra's, Bellman-Ford, Floyd-Warshall). These provide frameworks for solving a wide range of network and connectivity problems.

Finally, you learned **dynamic programming**, a powerful technique for optimization problems with optimal substructure and overlapping subproblems. This included problems like rod cutting, matrix chain multiplication, and subset sum, culminating in an understanding of pseudopolynomial time.

Algorithmic Paradigms and Problem Solving

Beyond specific algorithms, you have gained exposure to several key algorithmic paradigms:

- **Brute Force:** Trying all possible solutions.

- **Greedy Algorithms:** Making locally optimal choices in the hope of finding a global optimum. (e.g., Kruskal's, Prim's, Dijkstra's).
- **Divide and Conquer:** Breaking a problem into smaller subproblems, solving them independently, and combining their solutions (e.g., Merge Sort, Quick Sort).
- **Dynamic Programming:** Solving overlapping subproblems once and storing their results.

When approaching a new problem, consider which paradigm might apply. Analyze the problem's structure: Does it have optimal substructure? Overlapping subproblems? Can a greedy choice always lead to an optimal solution? Is there a way to break it down into independent parts?

Correctness and Computational Tractability

Two recurring themes have been proving algorithm **correctness** and analyzing **computational tractability**. Correctness ensures your algorithm works as intended for all valid inputs. Tractability, informed by complexity theory, tells you if an efficient solution is even possible. Understanding P, NP, and NP-completeness helps you manage expectations and choose appropriate strategies when faced with hard problems.

Preparing for Assessments and Beyond

Problem sessions and quiz reviews are invaluable opportunities to apply what you have learned and identify areas for improvement. Practice solving a variety of problems, paying attention to edge cases and constraints. Clearly articulate your algorithm's steps, prove its correctness, and analyze its time and space complexity.

Next Steps in Algorithms

Your path in algorithms does not end here. Consider exploring:

- **Advanced Data Structures:** Suffix trees, segment trees, treaps, skip lists.
- **Advanced Algorithm Design:** Randomized algorithms, approximation algorithms, online algorithms, parallel algorithms.
- **Specialized Areas:** Computational geometry, computational biology, cryptography.
- **Competitive Programming:** Apply your skills to solve challenging algorithmic puzzles.
- **Machine Learning:** Many machine learning algorithms rely heavily on efficient data structures and optimization techniques.

Continuous learning and practice are key to mastering algorithms. The principles and techniques learned in this course are foundational for many areas of computer science and will serve you well in your future endeavors.

Practice: Choose a problem from a past problem session or quiz. Outline how you would approach it, identifying the algorithmic paradigm, relevant data structures, and a plan for complexity analysis.

KEY TAKEAWAYS

- Floyd-Warshall algorithm efficiently computes all-pairs shortest paths in $O(V^3)$ time, handling negative edge weights but not negative cycles.
- Dynamic programming solves optimization problems like Matrix Chain Multiplication and Rod Cutting by breaking them into overlapping subproblems.
- Pseudopolynomial time algorithms, such as for Subset Sum, have running times polynomial in the input values but potentially exponential in the input's bit length.
- The complexity class P includes problems solvable in polynomial time, while NP includes problems whose solutions can be verified in polynomial time.
- NP-complete problems are the 'hardest' problems in NP; finding a polynomial-time solution for one implies $P=NP$, a major open question.
- Algorithmic paradigms like greedy, divide and conquer, and dynamic programming offer structured approaches to problem-solving.
- Continuous practice, rigorous complexity analysis, and correctness proofs are essential for mastering algorithms and preparing for advanced topics.

CHECK YOUR UNDERSTANDING

Answer these before moving on. Model answers are in the Answer Key at the back of this guide.

1. Explain the core idea behind the Floyd-Warshall algorithm for All-Pairs Shortest Path.
2. What distinguishes a pseudopolynomial time algorithm from a truly polynomial time algorithm?
3. Define the complexity classes P and NP, and provide an example of a problem in each class.
4. What does it mean for a problem to be NP-complete, and why is this concept important?
5. Name three major algorithmic paradigms covered in the course and briefly describe when each might be applied.

REFERENCE

Glossary

Algorithm

A well-defined computational procedure that takes input and produces output through a finite sequence of steps.

Amortized analysis

A method of analyzing an algorithm's running time over a sequence of operations, averaging out the cost of expensive operations.

Avl tree

A self-balancing binary search tree where the height difference between the left and right subtrees of any node is at most 1.

Big-o notation

An asymptotic notation that provides an upper bound on the growth rate of a function, commonly used for worst-case time complexity.

Binary heap

A complete binary tree that satisfies the heap property, meaning the parent node's value is always greater than or equal to (max-heap) or less than or equal to (min-heap) its children's values.

Binary search tree (bst)

A tree data structure where for each node, all keys in the left subtree are smaller than the node's key, and all keys in the right subtree are larger.

Breadth-first search (bfs)

A graph traversal algorithm that explores all the neighbor nodes at the present depth before moving on to nodes at the next depth level, typically using a queue.

Class np

The set of all decision problems for which a given 'yes' solution can be verified in polynomial time by a deterministic algorithm.

Class p

The set of all decision problems that can be solved by a deterministic algorithm in polynomial time.

Collision

Occurs in a hash table when two different keys map to the same index.

Decision problem

A type of computational problem that can be posed as a yes/no question for a given input.

Depth-first search (dfs)

A graph traversal algorithm that explores as far as possible along each branch before backtracking, typically using a stack or recursion.

Dijkstra's algorithm

A greedy algorithm that finds the shortest paths from a single source vertex to all other vertices in a graph with non-negative edge weights.

Dynamic programming

An algorithmic technique that solves complex problems by breaking them into simpler, overlapping subproblems and storing the results of these subproblems.

Floyd-warshall algorithm

A dynamic programming algorithm used to find shortest paths between all pairs of vertices in a weighted graph, capable of handling negative edge weights but not negative cycles.

Graph traversal

The process of visiting every vertex and edge in a graph in a systematic manner.

Hash function

A function that maps keys to indices in a hash table.

Load factor

The ratio of the number of elements to the table size in a hash table, indicating how full it is.

Memoization

A top-down dynamic programming technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again.

Negative cycle

A cycle in a graph where the sum of the weights of its edges is negative, making shortest paths involving the cycle undefined.

Np-complete

A decision problem that is in NP and is NP-hard, meaning every problem in NP can be reduced to it in polynomial time.

Optimal substructure

A property of problems where an optimal solution to the problem contains optimal solutions to its subproblems.

Overlapping subproblems

A property of problems where the same subproblems are encountered and solved multiple times during a recursive solution.

Polynomial-time reduction

A transformation of an instance of one problem into an instance of another problem, computable in polynomial time, such that solving the second problem solves the first.

Priority queue

An abstract data type that provides access to the element with the highest or lowest priority, often used in graph algorithms like Dijkstra's.

Pseudopolynomial time

An algorithm's running time that is polynomial with respect to the numeric value of the input, but exponential with respect to the number of bits required to represent that input value.

Ram model

A computational model where basic operations and memory access take unit time, used for algorithm analysis.

Relaxation

A core step in shortest path algorithms where an estimated shortest distance to a vertex is updated if a shorter path is found through an adjacent vertex.

Shortest path

A path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.

Stable sort

A sorting algorithm that preserves the relative order of elements with equal values.

REFERENCE

Answer Key

Chapter 1: Algorithms and Data Structures Fundamentals

1. Explain the difference between Big-O, Big-Omega, and Big-Theta notation.

Big-O (O) provides an upper bound on a function's growth, meaning it grows no faster than a given function. Big-Omega (Ω) provides a lower bound, meaning it grows at least as fast. Big-Theta (Θ) provides a tight bound, indicating the function grows at the same rate both above and below.

2. How does a dynamic array achieve $O(1)$ amortized time for appends, even though resizing involves copying all elements?

When a dynamic array resizes, it typically doubles its capacity. This expensive $O(N)$ operation occurs infrequently. Over a sequence of N appends, the total cost of all appends and resizes sums to $O(N)$, making the average cost per append $O(1)$.

3. What is a collision in the context of hash tables, and name two common methods for resolving them?

A collision occurs when two distinct keys are mapped to the same index by the hash function. Two common resolution methods are chaining, where each table slot points to a linked list of colliding elements, and open addressing, where probing for an empty slot is used.

4. What are the main assumptions or conditions under which linear sorting algorithms like Counting Sort or Radix Sort can achieve $O(N)$ time complexity?

Linear sorting algorithms achieve $O(N)$ by not relying solely on comparisons. Counting Sort assumes keys are small non-negative integers within a limited range. Radix Sort assumes keys have a fixed number of digits and uses a stable sorting algorithm for each digit pass.

5. Why is the stability of Counting Sort important when it is used as a subroutine in Radix Sort?

Stability means that elements with equal values maintain their relative order after sorting. In Radix Sort, when sorting by a more significant digit, elements that have the same value for that digit must retain the order established by previous passes on less significant digits. Counting Sort's stability ensures this property is preserved across passes.

Chapter 2: Advanced Data Structures and Graph Algorithms

1. Explain why AVL trees are preferred over basic Binary Search Trees in many applications.

AVL trees are preferred because they guarantee $O(\log n)$ time complexity for search, insertion, and deletion operations. Basic BSTs can degenerate into a linked list in the worst case, leading to $O(n)$ performance. AVL trees maintain balance through rotations, ensuring consistent logarithmic performance.

2. Describe the main difference between a min-heap and a max-heap, and how they are typically stored in memory.

A min-heap ensures the smallest element is at the root, with all parent nodes smaller than or equal to their children. A max-heap ensures the largest element is at the root, with all parent nodes greater than or equal to their children. Both are typically stored in a contiguous array due to their complete binary tree property, allowing efficient access to parent and child nodes via index arithmetic.

3. When would you choose Breadth-First Search (BFS) over Depth-First Search (DFS) for a graph problem?

You would choose BFS when you need to find the shortest path in an unweighted graph, or when you need to explore nodes level by level. BFS is also suitable for problems like finding all nodes within a certain distance from a source. DFS is better for problems involving deeper exploration, such as cycle detection or topological sorting.

4. What is the primary limitation of Dijkstra's algorithm, and what kind of graph problem would require a different shortest path algorithm?

The primary limitation of Dijkstra's algorithm is that it cannot correctly handle graphs with negative edge weights. If a graph contains negative edge weights, a different algorithm like Bellman-Ford would be required to find the shortest paths, as Dijkstra's greedy approach fails in such scenarios.

5. Briefly explain the 'relaxation' step in Dijkstra's algorithm.

The relaxation step in Dijkstra's algorithm involves checking if a shorter path to a neighbor vertex v can be achieved by going through the currently processed vertex u . If the sum of the distance to u and the weight of the edge from u to v is less than the current known distance to v , then v 's distance is updated to this new, shorter value.

Chapter 3: Shortest Paths and Dynamic Programming

1. When would you use Bellman-Ford's algorithm instead of Dijkstra's algorithm for single-source shortest paths?

You would use Bellman-Ford's algorithm when the graph contains negative edge weights. Dijkstra's algorithm assumes non-negative edge weights and will not produce correct results in the presence of negative weights. Bellman-Ford also has the advantage of detecting negative cycles, which render shortest paths undefined.

2. Explain the purpose of edge reweighting in Johnson's algorithm.

Edge reweighting in Johnson's algorithm transforms a graph with negative edge weights into an equivalent graph where all edge weights are non-negative. This transformation allows the use of Dijkstra's algorithm, which is faster, from each vertex. The reweighting ensures that shortest paths in the original graph correspond to shortest paths in the reweighted graph.

3. What are the two fundamental properties a problem must exhibit to be solvable by dynamic programming?

A problem must exhibit optimal substructure and overlapping subproblems to be solvable by dynamic programming. Optimal substructure means an optimal solution to the problem can be constructed

from optimal solutions to its subproblems. Overlapping subproblems means the same subproblems are solved repeatedly by a recursive algorithm.

4. Describe the difference between memoization and tabulation in dynamic programming.

Memoization is a top-down, recursive approach where solutions to subproblems are computed only when needed and stored in a cache. Tabulation is a bottom-up, iterative approach where solutions to subproblems are computed and stored in a table, typically in a specific order, building up from the smallest subproblems to the final solution.

5. How does the presence of negative cycles affect shortest path algorithms?

The presence of a negative cycle means that shortest paths involving that cycle are undefined, as one could traverse the cycle infinitely to achieve an arbitrarily small path weight. Algorithms like Bellman-Ford can detect negative cycles, indicating that no true shortest path exists. Algorithms like Dijkstra's, which do not handle negative weights, would fail or produce incorrect results if a negative cycle were present.

Chapter 4: Advanced Dynamic Programming and Computational Complexity

1. Explain the core idea behind the Floyd-Warshall algorithm for All-Pairs Shortest Path.

The Floyd-Warshall algorithm uses dynamic programming to find the shortest path between all pairs of vertices. It iteratively considers all possible vertices as intermediate points on a path. For each pair (i, j) , it updates the shortest path by checking if passing through an intermediate vertex k creates a shorter path than the current known path from i to j .

2. What distinguishes a pseudopolynomial time algorithm from a truly polynomial time algorithm?

A pseudopolynomial time algorithm's complexity is polynomial in the *numeric value* of its input. A truly polynomial time algorithm's complexity is polynomial in the *number of bits* required to represent its input. For large numeric inputs, a pseudopolynomial algorithm can be exponential in the input's bit length, making it inefficient.

3. Define the complexity classes P and NP, and provide an example of a problem in each class.

P is the class of decision problems solvable by a deterministic algorithm in polynomial time, like sorting or finding a shortest path. NP is the class of decision problems for which a 'yes' solution can be verified in polynomial time, such as the Subset Sum problem or the Traveling Salesperson Problem decision version.

4. What does it mean for a problem to be NP-complete, and why is this concept important?

An NP-complete problem is in NP and is NP-hard, meaning every problem in NP can be reduced to it in polynomial time. This concept is important because if a polynomial-time algorithm is found for any NP-complete problem, it implies $P=NP$, and all problems in NP could be solved efficiently. If no such algorithm exists, it guides us to seek approximation or heuristic solutions for these inherently hard problems.

5. Name three major algorithmic paradigms covered in the course and briefly describe when each might be applied.

Three major paradigms are Greedy algorithms, Divide and Conquer, and Dynamic Programming. Greedy algorithms are applied when locally optimal choices lead to a global optimum. Divide and Conquer is used for problems that can be broken into independent subproblems. Dynamic Programming is suitable for optimization problems with optimal substructure and overlapping subproblems.

CLOSING

Where to Go Next

Upon completing MIT 6.006, you will possess a strong foundation in algorithmic thinking. To consolidate your skills, apply the concepts learned to real-world problems. Consider implementing various data structures and algorithms from scratch to deepen your practical understanding.

Continue to build projects that require efficient problem-solving. This practice will help you recognize when and how to apply the algorithms you have studied. Actively seek out opportunities to optimize existing codebases or design new solutions with performance in mind.

This course prepares you for advanced topics in computer science, including complexity theory, cryptography, and specialized algorithm design. You are now well-equipped to explore more complex computational challenges.



MEGACOURSE

Learn Anything. Achieve Everything.

Track your progress, take the quizzes, and earn your certificate for this course at
megacourse.app/courses/mit-6-006-introduction-to-algorithms