Algorithms Video Lectures
ISBN: 9780134384436
August 2015

These video lectures are based on the "Algorithms" course that was developed at Princeton University by Robert Sedgewick and Kevin Wayne in conjunction with their textbook *Algorithms*, Fourth Edition, and their online content at algs4.cs.princeton.edu. The course has been taught for decades around the world and has been adapted as an exemplar course in the 2013 ACM-IEEE curriculum.

Recorded in a studio, these videos provide another perspective on the material. As with any lectures, they are intended to introduce the material and inspire further study. They make extensive use of visualizations and dynamic simulations that could not be included in printed media.

Students and professionals can use the lectures for self-study, and instructors can use them as the basis for teaching a course. Generally, the lectures cover the material in the order presented in the book, though some book topics have been combined, rearranged, or omitted in the lectures.

The following outline includes suggested readings from *Algorithms*, Fourth Edition, and links to online content on the companion site.

Introduction (9:22)

Lecture 1 suggested reading: Chapter 1, section 1.4
Online content: http://algs4.cs.princeton.edu/14analysis/

Lecture 1: Union-Find. We illustrate our basic approach to developing and analyzing algorithms by considering the dynamic connectivity problem. We introduce the union-find data type and consider several implementations (quick find, quick union, weighted quick union, and weighted quick union with path compression). Finally, we apply the union-find data type to the percolation problem from physical chemistry.

Dynamic Connectivity (10:22)
Quick Find (10:18)
Quick Union (7:50)
Quick-Union Improvements (13:02)
Union-Find Applications (9:22)

Lecture 2 suggested reading: Chapter 1, section 1.5
Online content: http://algs4.cs.princeton.edu/15uf/

Lecture 2: Analysis of Algorithms. The basis of our approach for analyzing the performance of algorithms is the scientific method. We begin by performing computational experiments to measure the running times of our programs. We use these measurements to develop hypotheses about performance. Next, we create mathematical models to explain their behavior. Finally, we consider analyzing the memory usage of our Java programs.

Introduction to Analysis of Algorithms (8:14)
Observations (10:05)
Mathematical Models (12:48)
Order-of-Growth Classifications (14:39)
Theory of Algorithms (11:35)
Memory (8:11)

Lecture 3 suggested reading: Chapter 1, section 1.3
Online content: http://algs4.cs.princeton.edu/13stacks/

Lecture 3: Stacks and Queues. We consider two fundamental data types for storing collections of objects: the stack and the queue. We implement each using either a singly-linked list or a resizing array. We introduce two advanced Java features—generics and iterators—that simplify client code. Finally, we consider various applications of stacks and queues ranging from parsing arithmetic expressions to simulating queueing systems.

Stacks (16:24)
Resizing Arrays (9:56)
Queues (4:33)
Generics (9:26)
Iterators (7:16)
Stack and Queue Applications (13:25)

Lecture 4 suggested reading: Chapter 2, section 2.1
Online content: http://algs4.cs.princeton.edu/21elementary/

Lecture 4: Elementary Sorts. We introduce the sorting problem and Java's Comparable interface. We study two elementary sorting methods (selection sort andinsertion sort) and a variation of one of them (shellsort). We also consider two algorithms for uniformly shuffling an array. We conclude with an application of sorting to computing the convex hull via the Graham scan algorithm.

Introduction to Sorting (14:43)

Selection Sort (6:59)
Insertion Sort (9:28)
Shellsort (10:48)
Shuffling (7:39)
Convex Hull (13:50)

Lecture 5 suggested reading: Chapter 2, section 2.2
Online content: http://algs4.cs.princeton.edu/22mergesort/

Lecture 5: Mergesort. We study the mergesort algorithm and show that it guarantees to sort any array of N items with at most NlgN compares. We also consider a nonrecursive, bottom-up version. We prove that any compare-based sorting algorithm must make at least ~NlgN compares in the worst case. We discuss using different orderings for the objects that we are sorting and the related concept of stability.

Mergesort (23:54)
Bottom-up Mergesort (3:20)
Sorting Complexity (9:05)
Comparators (6:43)
Stability (5:39)

Lecture 6 suggested reading: Chapter 2, section 2.3
Online content: http://algs4.cs.princeton.edu/23quicksort/

Lecture 6: Quicksort. We introduce and implement the randomized quicksort algorithm and analyze its performance. We also consider randomized quickselect, a quicksort variant which finds the kth smallest item in linear time. Finally, consider 3-way quicksort, a variant of quicksort that works especially well in the presence of duplicate keys.

Quicksort (19:33)
Selection (7:08)
Duplicate Keys (11:25)
System Sorts (11:50)

Lecture 7 suggested reading: Chapter 2, section 2.4
Online content: http://algs4.cs.princeton.edu/24pq/

Lecture 7: Priority Queues. We introduce the priority queue data type and an efficient implementation using the binary heap data structure. This implementation also leads to an efficient sorting algorithm known as heapsort.

We conclude with an applications of priority queues where we simulate the motion of N particles subject to the laws of elastic collision.

APIs and Elementary Implementations (12:52)
Binary Heaps (23:36)
Heapsort (14:29)
Event-Driven Simulation (22:38)

Lecture 8 suggested reading: Chapter 3, sections 3.1 and 3.2
Online content: http://algs4.cs.princeton.edu/31elementary/ and http://algs4.cs.princeton.edu/32bst/

Lecture 8: Elementary Symbol Tables. We define an API for symbol tables (also known as associative arrays) and describe two elementary implementations using a sorted array (binary search) and an unordered list (sequential search). When the keys are Comparable, we define an extended API that includes the additional methods min, max floor, ceiling, rank, and select. To develop an efficient implementation of this API, we study the binary search tree data structure and analyze its performance.

Symbol Table APIs (21:30)
Elementary Implementations (9:03)
Ordered Operations (6:26)
Binary Search Trees (19:56)
Ordered Operations in BSTs (10:31)
Deletion in BSTs (9:52)

Lecture 9 suggested reading: Chapter 3, section 3.3
Online content: http://algs4.cs.princeton.edu/33balanced/

Lecture 9: Balanced Search Trees. In this lecture, our goal is to develop a symbol table with guaranteed logarithmic performance for search and insert (and many other operations). We begin with 2-3 trees, which are easy to analyze but hard to implement. Next, we consider red-black binary search trees, which we view as a novel way to implement 2-3 trees as binary search trees. Finally, we introduce B-trees, a generalization of 2-3 trees that are widely used to implement file systems.

Search Trees (16:55)
Red-Black BSTs (35:30)
B-Trees (10:36)

Lecture 10: No suggested reading or online content references for this lecture

Lecture 10: Geometric Applications of BSTs. We start with 1d and 2d range searching, where the goal is to find all points in a given 1d or 2d interval. To accomplish this, we consider kd-trees, a natural generalization of BSTs when the keys are points in the plane (or higher dimensions). We also consider intersection problems, where the goal is to find all intersections among a set of line segments or rectangles.

Range Search (8:51)
Line Segment Intersection (5:46)
Kd-Trees (29:07)
Interval Search Trees (13:47)
Rectangle Intersection (8:10)

Lecture 11 suggested readings: Chapter 3, sections 3.4 and 3.5
Online content: http://algs4.cs.princeton.edu/34hash/ and
http://algs4.cs.princeton.edu/35applications/

Lecture 11: Hash Tables. We begin by describing the desirable properties of hash functions and how to implement them in Java, including a fundamental tenet known as the uniform hashing assumption that underlies the potential success of a hashing application. Then, we consider two strategies for implementing hash tables—separate chaining and linear probing. Both strategies yield constant-time performance for search and insert under the uniform hashing assumption. We conclude with applications of symbol tables including sets, dictionary clients, indexing clients, and sparse vectors.

Hash Functions (18:13)
Separate Chaining (7:28)
Linear Probing (14:37)
Context (10:09)
Sets (5:04)
Dictionary Clients (5:40)
Indexing Clients (8:57)
Sparse Vectors (7:41)

Lecture 12 suggested reading: Chapter 4, section 4.1
Online content: http://algs4.cs.princeton.edu/41undirected/

Lecture 12: Undirected Graphs. We define an undirected graph API and consider the adjacency-matrix and adjacency-lists representations. We introduce two classic algorithms for searching a graph—depth-first search and

breadth-first search. We also consider the problem of computing connected components and conclude with related problems and applications.

Introduction to Graphs (9:32)
Graph API (14:47)
Depth-First Search (26:22)
Breadth-First Search (13:34)
Connected Components (18:56)
Graph Challenges (14:29)

Lecture 13 suggested reading: Chapter 4, section 4.2
Online content: http://algs4.cs.princeton.edu/42directed/

Lecture 13: Directed Graphs. In this lecture we study directed graphs. We begin with depth-first search and breadth-first search in digraphs and describe applications ranging from garbage collection to web crawling. Next, we introduce a depth-first search based algorithm for computing the topological order of an acyclic digraph. Finally, we implement the Kosaraju-Sharir algorithm for computing the strong components of a digraph.

Introduction to Digraphs (8:30)
Digraph API (4:56)
Digraph Search (20:56)
Topological Sort (12:54)
Strong Components (20:22)

Lecture 14 suggested reading: Chapter 4, section 4.3
Online content: http://algs4.cs.princeton.edu/43mst/

Lecture 14: Minimum Spanning Trees. In this lecture we study the minimum spanning tree problem. We begin by considering a generic greedy algorithm for the problem. Next, we consider and implement two classic algorithms for the problem—Kruskal's algorithm and Prim's algorithm. We conclude with some applications and open problems.

Introduction to MSTs (4:04)
Greedy Algorithm (12:56)
Edge-Weighted Graph API (11:15)
Kruskal's Algorithm (12:28)
Prim's Algorithm (33:15)
MST Context (10:34)

Lecture 15 suggested reading: Chapter 4, section 4.4
Online content: http://algs4.cs.princeton.edu/44sp/

Lecture 15: Shortest Paths. In this lecture we study shortest-paths problems. We begin by analyzing some basic properties of shortest paths and a generic algorithm for the problem. We introduce and analyze Dijkstra's algorithm for shortest-paths problems with nonnegative weights. Next, we consider an even faster algorithm for DAGs, which works even if the weights are negative. We conclude with the Bellman-Ford-Moore algorithm for edge-weighted digraphs with no negative cycles. We also consider applications ranging from content-aware fill to arbitrage.

Shortest Paths APIs (10:51)
Shortest Path Properties (14:46)
Dijkstra's Algorithm (18:58)
Edge-Weighted DAGs (19:23)
Negative Weights (21:01)

Lecture 16 suggested reading: Chapter 6, section 6.4 (pp. 886-902)
Online content: http://algs4.cs.princeton.edu/64maxflow/

Lecture 16: Maximum Flow and Minimum Cut. In this lecture we introduce the maximum flow and minimum cut problems. We begin with the Ford-Fulkerson algorithm. To analyze its correctness, we establish the maxflow-mincut theorem. Next, we consider an efficient implementation of the Ford-Fulkerson algorithm, using the shortest augmenting path rule. Finally, we consider applications, including bipartite matching and baseball elimination.

Introduction to Maxflow (10:33)
Ford-Fulkerson Algorithm (6:32)
Maxflow-Mincut Theorem (9:38)
Running Time Analysis (8:49)
Java Implementation (14:29)
Maxflow Applications (22:20)

Lecture 17 suggested reading: Chapter 5, section 5.1
Online content: http://algs4.cs.princeton.edu/51radix/

Lecture 17: Radix Sorts. In this lecture we consider specialized sorting algorithms for strings and related objects. We begin with a subroutine to sort integers in a small range. We then consider two classic radix sorting algorithms—LSD and MSD radix sorts. Next, we consider an especially efficient variant, which is a hybrid of MSD radix sort and quicksort known as 3-way radix quicksort. We conclude with suffix sorting and related applications.

Strings in Java (17:43)
Key-Indexed Counting (12:06)
LSD Radix Sort (15:00)
MSD Radix Sort (13:41)
3-way Radix Quicksort (7:22)
Suffix Arrays (19:25)

Lecture 18 suggested reading: Chapter 5, section 5.2
Online content: http://algs4.cs.princeton.edu/52trie/

Lecture 18: Tries. In this lecture we consider specialized algorithms for symbol tables with string keys. Our goal is a data structure that is as fast as hashing and even more flexible than binary search trees. We begin with multiway tries; next we consider ternary search tries. Finally, we consider character-based operations, including prefix match and longest prefix, and related applications.

R-way Tries (32:19)
Ternary Search Tries (22:42)
Character-Based Operations (20:03)

Lecture 19 suggested reading: Chapter 5, section 5.3
Online content: http://algs4.cs.princeton.edu/53substring/

Lecture 19: Substring Search. In this lecture we consider algorithms for searching for a substring in a piece of text. We begin with a brute-force algorithm, whose running time is quadratic in the worst case. Next, we consider the ingenious Knuth-Morris-Pratt algorithm whose running time is guaranteed to be linear in the worst case. Then, we introduce the Boyer-Moore algorithm, whose running time is sublinear on typical inputs. Finally, we consider the Rabin-Karp fingerprint algorithm, which uses hashing in a clever way to solve the substring search and related problems.

Introduction to Substring Search (6:42)
Brute-Force Substring Search (10:11)
Knuth-Morris-Pratt (33:14)
Boyer-Moore (8:36)
Rabin-Karp (16:13)

Lecture 20 suggested reading: Chapter 5, sections 5.4
Online content: http://algs4.cs.princeton.edu/54regexp/

Lecture 20: Regular Expressions. A regular expression is a method for specifying a set of strings. Our topic for this lecture is the famous grep algorithm that determines whether a given text contains any substring from the set. We examine an efficient implementation that makes use of our digraph reachability implementation from Lectures 1 and 2.

Regular Expressions (20:03)
REs and NFAs (13:14)
NFA Simulation (18:27)
NFA Construction (11:42)
Regular Expression Applications (20:09)

Lecture 21 suggested reading: Chapter 5, section 5.5
Online content: http://algs4.cs.princeton.edu/55compression/

Lecture 21: Data Compression. We study and implement several classic data compression schemes, including run-length coding, Huffman compression, and LZW compression. We develop efficient implementations from first principles using a Java library for manipulating binary data that we developed for this purpose, based on priority queue and symbol table implementations from earlier lectures.

Introduction to Data Compression (22:27)
Run-Length Coding (5:59)
Huffman Compression (24:14)
LZW Compression (27:33)

Lectures 22, 23, and 24: No suggested readings or online content references for these lectures

Lecture 22: Reductions. In this lecture our goal is to develop ways to classify problems according to their computational requirements. We introduce the concept of reduction as a technique for studying the relationship among problems. People use reductions to design algorithms, establish lower bounds, and classify problems in terms of their computational requirements.

Introduction to Reductions (9:25)
Designing Algorithms (8:13)
Establishing Lower Bounds (9:16)
Classifying Problems (12:45)

Lecture 23: Linear Programming. The quintessential problem-solving model is known as linear programming, and the simplex method for solving it is one of the

most widely used algorithms. In this lecture, we give an overview of this central topic in operations research and describe its relationship to algorithms that we have considered.

Brewer's Problem (21:15)
Simplex Algorithm (11:49)
Simplex Implementations (16:21)
Linear Programming Reductions (11:46)

Lecture 24: Intractability. Is there a universal problem-solving model to which all problems that we would like to solve reduce and for which we know an efficient algorithm? You may be surprised to learn that we do not know the answer to this question. In this lecture we introduce the complexity classes P, NP, and NP-complete; pose the famous P = NP question; and consider implications in the context of algorithms that we have treated in this course.

Introduction to Intractability (17:00)
Search Problems (10:56)
P vs. NP (16:29)
Classifying Problems (13:43)
NP-Completeness (12:38)
Coping with Intractability (14:01)