Problem Solving with Algorithms and Data Structures

> I have only one method that I recommend extensively—it's called think before you write.

Richard Hamming

> It is foolish to answer a question that you do not understand. It is sad to work for an end that you do not desire.

George Polya

Problem Solving with Algorithms and Data Structures is one of our favorite courses to teach, because it drives at the core of computer programming: solving difficult problems.

Over the coming weeks, you will learn a number of important ways to model data (data structures) and answer interesting questions about them (algorithms). We are sure these will prove useful throughout your career. More importantly however, you will develop a stronger ability to understand, break down and solve novel problems, whether inventing your own techniques or repurposing those which you learn with us.

RECOMMENDED RESOURCES

It's important that you complete the prework for each class. Doing so will enable you to derive much more value from classes, as we will mostly be solving problems using the ideas covered in the readings. We have done our best to keep the prework minimal, and distinguish "further study" resources for those who may wish to explore topics in further depth.

We have a short online text at bradfieldcs.com/algos which is our distillation of what we feel to be the most practical aspects of a traditional treatment of algorithms text. Most of our assigned prework will be from /algos.

Our recommended next stop is Steven Skiena's video lectures and book: *The Algorithm Design Manual* (ADM below). Another great source of video content is from Tim Roughgarden of Stanford, available here. Either of these will serve you well if /algos fails to answer any of your questions.

Two common textbook recommendation are *Introduction to Algoritms* by Cormen, Leiserson, Rivest and Stein (CLRS) and Sedgewick's *Algorithms*. In our opinion, while these are both good books, they are essentially reference books, at which point you may as well consult Knuth's *The Art of Computer Programming*. Knuth is more accessible than you might imagine, and incredibly thorough. All of these are available in the Bradfield library, for when you'd like a different perspective on a topic.

For those who like interactive courseware, Khan Academy has a brief introductory algorithms course in collaboration with Tom Cormen, one of the authors of CLRS. It only surveys a handful of topics but is well produced and includes some short programming exercises.

Finally, we will be doing a lot of practice problems to reinforce the concepts we cover. We have chosen most of these from leetcode, but other good sources are hackerrank, topcoder and UVa Online Judge. We strongly encourage you to use problems from these websites to test and strengthen your understanding of the concepts we cover.

CLASSES

1 Technical Problem Solving and Analysis

We start by reminding ourselves why we're even here: to be able to solve challenging technical problems!

The bar for "challenging" is different for each of us, but no matter where you are on your journey there'll be problems that you won't be able to solve simply by staring at them! Our first class will give us a systematic approach to problem solving that includes exploration, planning, implementation and revision. We'll then practice this approach on a few interesting problems.

There are often many ways to "solve" any given technical problem, so we need a shared understanding of how we may assess one approach to be preferable over another. In this class we'll introduce algorithmic analysis with Big O notation, and practice by analyzing some short code snippets. The analysis of algorithms is a huge field (to which some computer scientists dedicate their entire lives) but by the end of this class you should be able to assess the time and space cost, in Big O terms, of most of the programs you'll encounter in the wild.

Prework

Our problem solving technique derives from George Pólya's *How to Solve It*. If you're able to locate a copy, please read the first section (you can stop before "A Short Dictionary of Heuristic"). If not, the Wikipedia article or this summary cover the main ideas. When you feel bought in to this 4 step approach, try it out on the first 10 exercism.io problems in your language of choice. Did your exploration and planning help you arrive at a neater solution than where your first instincts would have lead you? Did your solutions improve when you reminded yourself to revise them after you'd "finished" them?

To prepare for our conversation on algorithmic analysis, please read the /algos section "Analysis". Practice by coming up with 3 different ways to solve this problem: "how many handshakes occur when n people at a party greet each other"? What are the Big O time and space costs of each of your approaches?

Further Resources

We'd love to see more resources available around problem solving. Other than *How to Solve It*, we like Hammock Driven Development, a talk by Rich Hickey (the creator of Clojure and Datomic) about why it's important to think deeply as a programmer, and *Programming Pearls*, an old book by Jon Bentley that surveys some neat programming tricks and describes how they were developed.

There are *far more* resources available for algorithmic analysis. Skiena provides a good introduction in chapters 1-2 of ADM as well as his videos introduction and asymptotic notation. We also like the short videos and notes that Tom Cormen put together for the Khan Academy section on asymptotic notation.

2 Data Structures

Most modern programming languages come with useful data structures built in, such as dynamic arrays (`list` in Python, `Array` in Ruby or JavaScript, `ArrayList` in Java, etc) and maps (`dict` in Python, `HashMap` in Ruby, `Object` in JavaScript, `HashMap` or `TreeMap` in Java, etc). Any experienced programmer knows how to use these, but by the end of this class you'll also know:

- Which operations are typically fast or slow, and under which conditions;
- When it's worth using more exotic (or writing your own!) versions of typically built in structures;
- What are the best underlying data structures to use for more abstract data types like queues and stacks;
- What you should ask about data structures when learning a new language; and,
- How to cope when your new favorite language is missing your old favorite data structures.

Prework

Please arrive with a good understanding of the semantics of stacks, queues, dequeues and lists by reading the corresponding sections of /algos, then testing yourself by implementing a queue using stacks and a stack using queues. If you come to class with a good understand how these data structures behave, we'll be able to focus on the more interesting question of how they're implemented.

For hashmaps, read the /algos section on hashing, then test your understanding by researching the hashmap implementation in your language of choice. If your language has an open source implementation you may wish to read the source, otherwise try to find sufficiently technical articles (and ask us if you get stuck). You should be able to answer questions such as "how are collisions resolved?" and "when does the underlying array grow?"

Further Resources

- Skiena videos on elementary data structures
- ADM 3.1-3.2
- Leetcode problems tagged stack, queue or linked list

3 Divide and Conquer, Sorting and Searching

"When would you ever write your implement your own sorting algorithm!?" The answer turns out to be… very rarely, if you're working in a high level language. But wait! Sorting algorithms are a terrific lens through which to appreciate the stupendous power of one of algorithms' Greatest Hits: the divide and conquer strategy. In this class we'll explore the problem of sorting until we're confident that it can be done at no better than speed X, then be amazed by two divide and conquer based algorithms that manage to run at the much better speed Y! We'll then practice divide and conquer on other problems.

"When would you ever need to write a binary search over an array of sorted integers!?" The answer turns out to be… very rarely. But wait! Binary search is more broadly applicable than most people realize, so we also spend some time on *constructing search spaces* over which we then search. We'll practice this on problems that don't look at all like "binary search" problems. This is also great practice for applying divide and conquer generally, since binary search can be thought of as "divide and ignore".

Prework

Please come *unprepared* for the sorting material, if it's not too late. Most people get stuck on the implementation details of particular sorting algorithms, which is counterproductive for our purposes.

For binary search, read the corresponding section in /algos. Test yourself by writing a plain binary search over a sorted array of integers. Watch for edge cases! Binary search is notorious for being tricky to implement at the edges.

Further Resources

- Skiena videos on mergesort/quicksort
- ADM chapter 4
- Roughgarden videos sections 2 and 13
- European folk dancing sort videos

Practice Problems

- Search a 2D Matrix
- Maximum Subarray
- Majority Element
- Sort Colors
- Different Ways to Add Parentheses
- Leetcode problems tagged sort or binary search

4 Graph Search

If we had to choose a single Greatest Hit of algorithms, it would be graph search. It's tempting to give examples where graph search shines, but by the end of this class you'll actually appreciate that *almost anything* can be modeled as a graph and *almost any problem* can be solved with graph search.

In this class we'll start with simple breadth first and depth first traversal over trees, and extend our understanding to breadth first search and depth first search over graphs. By the end, you should be able to confidently choose between these two strategies, and implement either one over graphs that you model yourself.

Prework

Please read the first sections 1-4 of Trees and 1-5 of Graphs, in /algos. The goal is to be generally comfortable implementing a depth first or breadth first traversal over a tree, at least, with a sense of how these are extended to graphs. We'll help you solidify your understanding in class, but please attempt to implement BFS and/or DFS a couple of times before arriving, for instance for the problems Path Sum or Maximum Depth of Binary Tree.

Further Resources

- Skiena videos on graph data structures, breadth first search and depth-first-search
- ADM chapter 5
- Roughgarden videos section 5

Practice Problems

- Number of Islands
- Pacific Atlantic Water Flow
- Perfect Squares
- Further Leetcode problems tagged depth first search or breadth first search

5 Advanced Graph Search: Uniform Cost Search and A*

BFS and DFS are tremendously powerful, but provide little help on problems over graphs with weighted edges (in other words, where the strength of a relationship between two entities matters). In practice, most interesting graphs are weighted, so we use this class to extend our understanding of graph search with two important algorithms: uniform cost search (also known as Dijkstra's algorithm) and A* (pronounced A-star) search.

Using A* search in practice is all about finding suitable "heuristic functions": measures of how close we are to our goal. We'll practice this skill on problems where the choice of heuristic function makes a big difference.

Prework:

Read Shortest Paths in /algos. Then read this article that will tie together BFS, DFS, Dijkstra's Algorithm and introduce two *heuristic based* search algorithms called "Best First (greedy)" and "A*" (pronounced A-star). Then attempt to solve this problem on HackerRank. (The main point is not how to do the conversion to binary notation, most languages have a library function to do this such as the bin() function in Python)

Further Resources:

The Berkeley AI lecture on uniform cost search and A* is excellent, as is the corresponding project, below.

Practice Problems

Implement BFS, DFS, UCS, and A* in the context of a Pacman grid using the Berkeley AI Search Lab.

Word Ladder. You may wish to solve this problem without a heuristic – if you do so consider what heuristics you might add in order to improve the speed of your search.

6 Constraint Satisfaction Problems

This class introduces backtracking and constraint propagation as two powerful techniques for addressing a class of problems known as “constraint satisfaction”. Typical textbook examples of these problems tend to be small games like “N Queens” or Sudoku, but we’ll see that we can apply these strategies to most problems where we’d like a computer to help us search through an enormous number of possible choices, but where we have some extra information about which search paths are viable.

Prework

Read chapter 5 of *Artificial Intelligence: A Modern Approach (2nd edition)* kindly hosted by UC Berkeley. Specifically, look at sections 5.1 and 5.2, then skim the rest of the chapter. Pay particular attention to the definition of a “constraint satisfaction problem” and the two algorithms *Backtracking-Search* and *Recursive-Backtracking* defined on page 142. Then attempt to Restore IP Addresses From a String.

Practice Problems

- Generate Parentheses
- Word Search
- Sudoku Solver
- Further Leetcode problems tagged backtracking

Further Resources

The Skiena lectures on backtracking Day one and Day Two

7 Dynamic Programming

Recursion is one of the most powerful concepts in algorithm design. This class ensures that your recursive thinking is not in vain, by introducing dynamic programming: an important technique to avoid unnecessary calculation and make certain recursive problems tractable. Dynamic programming has proved invaluable in applications varying from database query optimization to genomics, and at time of writing it's a popular topic of question for technical interviews :).

Prework

Please read the /algos section on recursion and dynamic programming, and test yourself by attempting to solve this problem: Best Time to Buy and Sell Stock.

Further Resources

- ADM chapter 8
- Roughgarden videos section 15
- Skiena videos on DP: introduction, edit distance example and applications of dynamic programming

Practice Problems

- Triangle
- Best Time to Buy and Sell Stock with Cooldown
- Decode Ways
- Further Leetcode problems tagged dynamic programming

8 Rapid Fire Problem Solving

This class will be an opportunity to put your problem solving techniques to good practice. We will speed through a large number of problems, aiming to develop a reasonable plan or pseudocode implementation for each.

Further Resources

The Art of Computer Programming