Bradfield
SCHOOL *of* COMPUTER SCIENCE

Problem Solving with Algorithms and Data Structures

August 2018 with Ozan Onay and Elliott Jin

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'll learn a number of important data structures and algorithms, which we're sure will prove useful throughout your career. Just as importantly, you'll 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.

Class Structure

We have carefully designed this course to maximize the time that we have available. To maximize its effectiveness, we will expect you to complete all necessary pre- and post-class work. We have scoped these with the expectation that each pre- or post-class obligation will take 1-1.5hrs each, and have included "stretch goals" for those able to devote more time.

Classes generally follow this structure:

1. Prework, for which you will be asked to read lecture notes or watch video lectures, then answer comprehension questions and/or solve a problem. Your answers and/or solution should be submitted to your instructor ideally 12-24 hours before class commences, so that they can adjust the class content accordingly and address any common questions;
2. In-class work, where your instructor will clarify any misunderstandings and help consolidate your understanding of the material, and guide you through a number of programming problems. Typically, you will solve the problems in pseudocode in class, often in pairs. To ensure that you've fully understood the problems, you should translate the pseudocode into an implementation in your language of choice and submit it to your instructor for feedback.
3. Post-class work, where you will be given a challenging extra problem or two to solve. These should ideally be submitted to the instructor within a day or two of class completion.

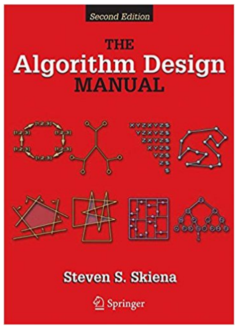
Work should be submitted to your instructor by slack, or if you prefer by asking your instructor to review a PR which incorporates your work into a github/gitlab repository. Solutions to each section will be available from this page after each class.

Recommended Resources

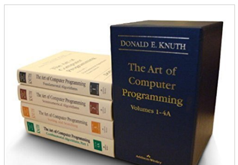
It's important that you complete the prework for each class. Doing so will enable you to derive much more value from classes, as we will mostly be solving problems using the ideas covered in the readings. We have done our best to keep the prework minimal, and distinguish "further study" resources for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

1. Technical Problem Solving and Analysis

Monday, 6 August 2018

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.

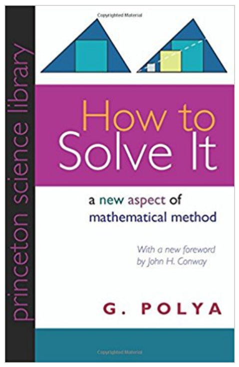
There are often many ways to "solve" any given technical problem, so we need a shared understanding of how we may assess one approach to be preferable over another. In this class we'll introduce algorithmic analysis with Big O notation, and practice by analyzing some algorithms.

The analysis of algorithms is a huge field—to which some computer scientists dedicate their entire lives—[3]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.

[3] One such devotee is Donald Knuth, who started writing *The Art of Computer Programming* in 1962 and has published approximately half of his planned volumes.

Pre-class Work

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. To prepare for our conversation on algorithmic analysis, please read the /algos section "Analysis".


princeton science library
How to Solve It
a new aspect of mathematical method
With a new foreword by John H. Conway
G. POLYA

Then, test your understanding with the following problem: a pangram is a phrase which contains every letter at least once, such as "the quick brown fox jumps over the lazy dog". Write a function which determines if a given string is a pangram.

This should be an easy problem, so it's a good excuse to practice Polya's method and algorithmic analysis. So, please find at least three different strategies for solving this problem, proceeding through Polya's method each time. In each case, make a note of how you came to that strategy and what the time and space complexity are. Please submit your solutions by Slack to your instructor prior to the start of class.

Post-class Work

For more practice at problem solving and analysis, select 5 problems from exercism.io in your language of choice. Some of these problems are much easier than others; you should select problems that seem challenging but realistic to you.

Please employ Polya's method to solve these problems, and submit each version of your solutions as you (hopefully) iteratively improve them. In either the exercism interface or inline comments, please also note the time and space complexity for each solution. When you are ready, send the URLs of your submitted solutions to your instructor, for code review.

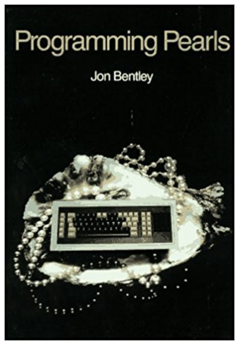
If we can leave you with just one insight from this entire course, we hope it is that rigorous problem solving leads to better programs. So now may be a good time to ask yourself: did your exploration and planning help you arrive at a neater solution than where your first instincts would have lead you? Did your solutions improve when you reminded yourself to revise them after you'd "finished" them?

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[5] about why it's important for programmers to spend time in deep thought, and *Programming Pearls*,[6] an old book by Jon Bentley that surveys some neat programming tricks and describes how they were developed.

[5] Rich Hickey is the creator of the Clojure programming language and Datomic database management system. Many of his talks are worth watching; we particularly like Simple Made Easy, The Value of Values and The Language of the System

[6] Since *Programming Pearls* is quite old, you may need some open mindedness to appreciate the relevance to appreciate the relevance of some of the techniques, but we assure you that your patience will be well rewarded!



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 Structure Implementation

Thursday, 9 August 2018

Most modern programming languages come with useful data structures built in, such as dynamic arrays[7] and maps.[8] 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.

[7] You may know these as `list` in Python, `Array` in Ruby or JavaScript, or `ArrayList` in Java. What distinguishes them from a true array is that they are designed to be able to grow and shrink as elements are added. Often, they will also allow for elements of various types.

[8] You may know these as `dict` in Python, `HashMap` in Ruby or `Object` or `Map` in JavaScript. While it's most common to implement a map as a hashmap, it is also possible (and in some contexts preferable) to use a tree under the hood instead. This is the case with Clojure for instance, and in Java we are given the choice between `HashMap` and `TreeMap`.

Pre-class Work

Please arrive with a good understanding of the semantics of stacks, queues, double-ended queues and linked lists by reading the corresponding sections of /algos. For hashmaps, read the /algos section on hashing.

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, and how their implementation might impact their performance.

While reading, you may wish to ask yourself these questions:

- What is the difference between an abstract data type and an implementation of a data structure?
- Consider two implementations of the queue abstract data type, one implemented using a dynamic array and the other as a doubly linked list. What is the Big O complexity of the push and pop operations? Other than Big O complexity, can you think of any potential performance differences between the two?
- What are the trade-offs being made between chaining and linear probing? Which of these do you think is more commonly used?
- Considering a hash map that uses chaining, when should the underlying array grow? What about for one that uses linear probing?

To confirm your understanding, solve the two problems implement a queue using stacks and implement a stack using queues, and as usual submit your solutions to your instructor.

In-class Exercise

One of the exercises for this class will be to implement and profile a fast queue.

Post-class Work

After class, please complete and submit your final queue implementation.

Further Resources

ADM covers the relevant data structures in sections 3.1-3.3. Skiena's lecture elementary data structures does the same.

For those seeking more practice, we suggest Leetcode problems tagged stack, queue or linked list.

For another introductory view of hashing, with an additional peek at some potential alternatives based on machine learning, see this Bradfield article. For more detail, see this paper comparing hashing strategies. To take the red pill, see Chapter 6.4 of *The Art of Computer Programming, Volume 3*.

For a better sense of the trade-offs between chaining and open addressing, see this lively Ruby core conversation surrounding their switch from the former to the latter in 2016.

3. Divide and Conquer, Sorting and Searching

Monday, 13 August 2018

Divide and conquer is a broadly applicable problem solving strategy, renowned for producing astonishing results.[9] We'll explore the idea generally, as well as in the context of binary search and sorting.

[9] One astonishing result is the Karatsuba algorithm for fast multiplication of two n-digit numbers. The renowned complexity theorist Kolmogorov had conjenctured that the fastest possible multiplication algorithm was $O(n^2)$, and presented this conjecture at a seminar in 1960 attended by the then 23-year-old Karatsuba. Within a week, Karatsuba had disproved the conjecture, finding an $O(n^{\log_2 3})$ algorithm using an approach which would later come to be named divide and conquer.

This caused Kolmogorov to become wildly excited, terminating the seminar and publishing and article on the result in Karatsuba's name, kick starting a period of intense interest in the area.

The history of the result is detailed in Karatsuba's later paper The Complexity of Computations. Pictured below is Karatsuba as a high school graduate, a few years before his remarkable discovery.

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".

Turning our attention to sorting, we'll explore the problem until we're convinced that it can be done at no better than $O(n^2)$, and be duly impressed by two divide and conquer based algorithms that manage to run at the *much* better O(n log n)! It's uncommon—though a rare pleasure—to implement your own sorting algorithms from scratch, so instead we'll focus our discussion on the overall strategy, as it can be applied in main domains outside of searching and sorting.

Pre-class Work

For binary search, read the corresponding section in /algos. For sorting, Read this article on sorting and use this visualization tool to deepen your understanding.

While you're reading, you may wish to ask yourself these questions:

- Hashing and binary search both address the same problem: finding an item in a collection. What are some trade-offs between the two strategies? When might you want to pick one over the other?

- If sorting takes (at minimum) O(n log n) time, and binary search takes O(log n) time, under what circumstances might it worth it to sort a collection in order to perform binary search?
- For each of bubble, insertion, and selection sort, what are the best and worse case scenarios?
- Why is the sorting problem a good candidate for divide and conquer?

Then, when you're finished, 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.[10]

[10] A quote from the linked Wikipedia article:

"When Jon Bentley assigned binary search as a problem in a course for professional programmers, he found that ninety percent failed to provide a correct solution after several hours of working on it, and another study published in 1988 shows that accurate code for it is only found in five out of twenty textbooks. Furthermore, Bentley's own implementation of binary search, published in his 1986 book Programming Pearls, contained an overflow error that remained undetected for over twenty years. The Java programming language library implementation of binary search had the same overflow bug for more than nine years."

Post-class Work

At the very least, please solve the problems Search a 2D Matrix and Different Ways to Add Parentheses and submit your solutions to your instructor.

Many of the Leetcode problems tagged divide and conquer, sort or binary search are also worth attempting. In particular we recommend Maximum Subarray, Majority Element and Sort Colors.

Further Resources

Tim Roughgarden in particular does a great job of covering divide and conquer, including the astonishing Strassen matrix multiplication algorithm.

Skiena covers divide and conquer in the context of searching, in chapter 4 of ADM and these videos on mergesort/quicksort.

Finally, for a bit of fun, see these European folk dancing sort videos. The inefficient algorithms look tiring!

4. Graph Search

Thursday, 16 August 2018

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,[11] 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.

[11] The stereotypical examples are social networks, the web and the internet. But if we were to emphasize these examples, you're likely to come away with a narrow view of what is in fact a very broadly applicable tool.

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.[12]

[12] We will stick to unweighted graphs in this class, even though you are more likely to encounter graphs where edges are weighted. We will need the algorithms in our *next* class in order to be able to adequately deal with these.

Pre-class Work

Prior to class, you should be able to implement both breadth first and depth first search over a tree or graph. To achieve this, please first work through sections 2-4 of Trees and 1-5 of Graphs, in /algos.

While reading, you may want to focus on understanding how trees and graphs are implemented, what is required to search through a tree or graph, and the relationship between DFS/BFS and stacks/queues respectively.

Then, solve the problem Minimum Depth of Binary Tree.

Post-class Work

At a minimum, please complete the problems Perfect Squares and Number of Islands and as usual submit your solution to your instructor.

Graph search should really start to become intuitive after a few more problems. Any tagged depth first search or breadth first search on Leetcode should be a good candidate, but we particularly recommend Open the Lock, Pacific Atlantic Water Flow, Word Ladder II and Minesweeper.

Further Resources

The relevant Skiena lectures are graph data structures, breadth-first search and depth-first search. The relevant section of Tim Roughgarden's course is titled "Graph Search and Connectivity", and you should focus on the search aspects (stopping around "Topological Sort").

The *Algorithm Design Manual* coverage of graph traversal is excellent, and contained in chapter 5.

5. Weighted Graphs, Dijkstra's Algorithm and A*

Monday, 20 August 2018

Breadth-first and depth-first search are tremendously useful, but don't quite extend to finding shortest path over weighted graphs,[13] which turns out to be a common problem. In this class, we develop our understanding of graph search further with with two important algorithms: Dijkstra's algorithm[14] and A*.[15]

[13] Recall that a weighted graph is one where the strength of the relationship between two entities *matters*, which we model by assigning a "weight" value to edges.

[14] While we stick with the convention of calling this algorithm Dijkstra's, the better name—used more in the AI community—is "uniform cost search". Dijkstra was the first to develop many algorithms, but this one was well known to others before Dijkstra rediscovered it. Furthermore, "uniform cost search" begins to describe its approach: the frontier of the search is defined by points of equivalent cost.

[15] Pronounced "A-star", this algorithm was developed at Stanford Research Institute to help Shakey the Robot navigate obstacles in a room. At the time, Dijkstra's algorithm was the best path planning option available. A* was seen as a minor enhancement to the algorithm, but the improvement in efficiency was very significant.

Pictured below is Shakey on display at Computer History Museum.

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

Pre-class Work

Please read Shortest Paths in /algos, followed by this article tying together BFS, DFS, Dijkstra's Algorithm and the two *heuristic based* search algorithms "best first" (greedy) search and A*. During your reading, ask yourself what exactly are the differences between each of the algorithms, to what kind of problems they may apply, and where you might prefer one over another.

When you've finished reading, please solve the problem Network Delay Time.

In-class Exercise

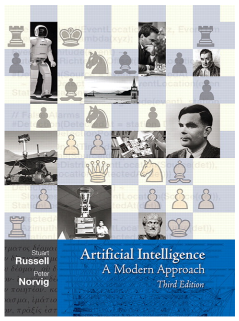
One of the problems we'll do in class will involve implementing a number of graph search algorithms over a grid. For those intending to solve this in Python, we have provided some scaffolding, which you may wish to review before class.

Post-class Work

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

Further Resources

The Berkeley AI lecture on uniform cost search and A* is excellent, as is the corresponding project, referenced in the post-class work.



One of the best resources for understanding uniform cost search and A* in the context of artificial intelligence is *Artificial Intelligence: A Modern Approach*. See chapter 3 of the 3rd edition: "Solving Problems by Searching".

6. Backtracking and Constraint Satisfaction

Thursday, 23 August 2018

Recursion is among the most powerful tactics that programmers can bring to bear on a problem. Done correctly, recursion can be used to solve complex problems in wonderfully simple code.

Unfortunately, the naive recursive approach will sometimes be too slow on large input. This and the next class present strategies to leverage the simplicity of recursion while avoiding unnecessary computation.

After refamiliarizing ourselves with recursion, we will use the back-tracking search technique to effectively share state between steps of a recursive (depth-first) search, avoiding unnecessary copying. We will then consider problems where it is particularly helpful to "prune the search tree" using a technique called constraint propagation.

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.

Pre-class Work

First, read the recursion section of /algos if you have not already. On the topic of backtracking, please read this introductory article as well as Peter Norvig's fantastic articles Solving Every Sudoku Puzzle.[17]

While you're reading, you may wish to ask yourself how exactly

[17] Peter Norvig is both an accomplished researcher and excellent programmer, currently a director of research at Google. He is known for co-authoring the wildly successful textbook *Artificial Intelligence: A Modern Approach* as well as—in some circles—for his influential article Teach Yourself Programming in Ten Years. He has also written a number of highly enlightening essays and interactive

backtracking helps to improve performance over recursive brute force approaches, and in what situations backtracking will **not** be able to provide performance benefits.

After you've finished reading, please solve the problem Word Search and as usual submit your solution to your instructor prior to class.

Post-class Work

You should find that backtracking search problems become much easier with a little practice. At a minimum, please solve either Restore IP Addresses From a String or Generate Parentheses.

There are many more excellent Leetcode problems tagged backtracking, including the classic N-Queens and Sudoku Solver problems.

Further Resources

Skiena has great coverage of backtracking, as videos part one and part two, as well as chapter 7 "Combinatorial Search and Heuristic Methods" in ADM.

Russell and Norvig cover constraint satisfaction with considerable clarity in chapter 7 of *Artificial Intelligence: A Modern Approach*.

7. Dynamic Programming

Monday, 27 August 2018

This class ensures that your recursive thinking is not in vain, by introducing dynamic programming: an important technique to avoid unnecessary computation and render certain recursive problems tractable.[18] Dynamic programming has proved invaluable in applications varying from database query optimization to sequence alignment in bioinformatics.

[18] The name "dynamic programming" could certainly be more descriptive. Richard Bellman claims in his autobiography that he chose these words because it's impossible to use the adjective "dynamic" in a pejorative sense, and because the practical connotation of the word "programming" would appeal to the research-averse Secretary of Defense of the time (who was indirectly responsible for his funding).

This story may not quite be true; an alternative account suggests that he was trying to "upstage Dantzig's linear programming by adding dynamic". For more, see the dynamic programming Wikipedia article.

Pre-class Work

Please read the /algos section on dynamic programming. While reading, you may wish to ask yourself:

- In what scenarios can it be beneficial to apply dynamic programming?
- What are the differences between the memoization (top-down) and tabulation (bottom-up) approaches?
- Is there a relationship between the parameters of a recursive function and the corresponding memo table? What about between the time and space complexities of a top-down approach?

When you've finished reading, solve the problem Maximum Subarray on Leetcode, and submit your solution to your instructor.

Post-class Work

Dynamic programming won't feel comfortable until you've done quite a lot of practice: we would suggest at least *ten* problems. At the very least, please solve Triangle and Best Time to Buy and Sell Stock with Cooldown.

There are a large number of fun, challenging problems tagged dynamic programming on Leetcode. In particular, we recommend continuing with the "buy and sell stock" series of problems, as well as Decode Ways, Coin Change, Word Break and Edit Distance.

Further Resources

Skiena covers dynamic programming in some depth, in his videos introduction, edit distance example and applications of dynamic programming as well as chapter 8 of ADM.

Tim Roughgarden also covers dynamic programming, within part 2 of his course on Lagunita.

8. Rapid Fire Problem Solving

Thursday, 30 August 2018

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.

Epilogue

As demonstrated by Knuth, it's possible to spend a lifetime studying algorithms and data structures. While few of us will become researchers or develop novel algorithms, we will all find many opportunities to use these wonderful tools, if only we know where to look!

We hope that this course has helped you take a step or two forward on this long path, and provided a clearer map of the road ahead. We would suggest continuing to explore some of the "further resources" listed above, practicing more problems, and most importantly fostering a playful and positive attitude toward your relationship with the material. In our experience, those who value algorithms and data structures—even without a specific objective—more often find themselves in situations where they can apply their knowledge, enjoy the process, and complete the virtuous cycle by deriving motivation for further study!