Learning Algorithms

A Programmer's Guide to Writing Better Code

https://chools.in/ https://choolsgroup.com/ https://choolskool.com/

Table of Contents

Preface

Who This Book Is For

If you are reading this book, I assume you already have a working knowledge of a programming language, such as Python. If you have never programmed before, I encourage you to first learn a programming language and then come back! I use Python in this book because it is accessible to programmers and nonprogrammers alike.

Algorithms are designed to solve common problems that arise frequently in software applications. When teaching algorithms to undergraduate students, I try to bridge the gap between the students' background knowledge and the algorithm concepts I'm teaching. Many textbooks have carefully written—but always too brief—explanations. Without having a guide to explain how to navigate this material, students are often unable to learn algorithms on their own.

In one paragraph and in Figure P-1, let me show you my goal for the book. I introduce a number of data structures that explain how to organize information using primitive fixed-size types, such as 32-bit integer values or 64-bit floating point values. Some algorithms, such as Binary Array Search, work directly on data structures. More complicated algorithms, especially graph algorithms, rely on a number of fundamental abstract data types, which I introduce as needed, such as *stacks* or *priority queues*. These data types provide fundamental operations that can be efficiently implemented by choosing the right data structure. By the end of this book, you will understand how the various algorithms achieve their performance. For these algorithms, I will either show full implementations in Python or refer you to third-party Python packages that provide efficient implementation.

If you review the associated code resources provided with the book, you will see that for each chapter there is a `book.py` Python file that can be executed to reproduce all tables within the book. As they say in the business, "your mileage may vary," but the overall trends will still appear.

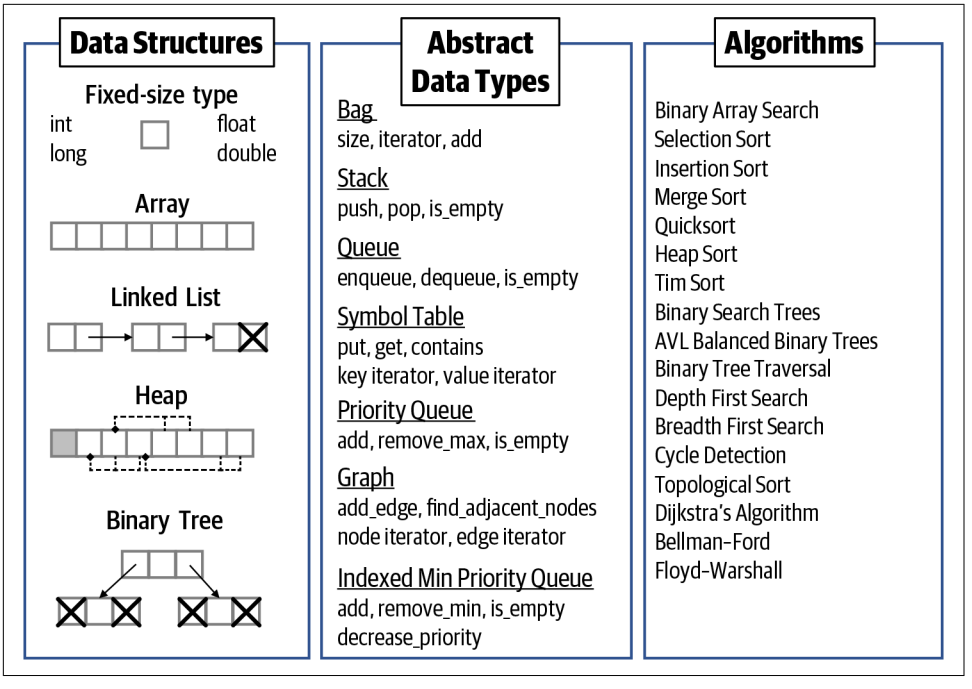


Figure P-1. Summary of the technical content of the book

At the end of every chapter in the book are challenge exercises that give you a chance to put your new knowledge to the test. I encourage you to try these on your own before you review my sample solutions, found in the code repository for the book.

About the Code

All the code for this book can be found in the associated GitHub repository, *http://github.com/heineman/LearningAlgorithms*. The code conforms to Python 3.4 or higher. Where relevant, I conform to Python best practices using double underscore methods, such as `__str()__` and `__len()__`. Throughout the code examples in the book, I use two-space indentation to reduce the width of the code on the printed page; the code repository uses standard four-space indentation. In a few code listings, I format code using an abbreviated one-line `if` statement like `if j == lo: break`.

The code uses three externally available, open source Python libraries:

- NumPy version 1.19.5
- SciPy version 1.6.0
- NetworkX version 2.5

NumPy and SciPy are among the most commonly used open source Python libraries and have a wide audience. I use these libraries to analyze empirical runtime performance. NetworkX provides a wide range of efficient algorithms for working with graphs, as covered in Chapter 7; it also provides a ready-to-use graph data type implementation. Using these libraries ensures that I do not unnecessarily reinvent the wheel. If you do not have these libraries installed, you will still be fine since I provide workarounds.

All timing results presented in the book use the `timeit` module using repeated runs of a code snippet. Often the code snippet is run a repeated number of times to ensure it can be accurately measured. After a number of runs, the minimum time is used as the timing performance, *not* the average of all runs. This is commonly considered to be the most effective way to produce an accurate timing measurement because averaging a number of runs can skew timing results when some performance runs are affected by external factors, such as other executing tasks from the operating system.

When the performance of an algorithm is highly sensitive to its input (such as Insertion Sort in Chapter 5), I will clearly state that I am taking the average over all performance runs.

The code repository contains over 10,000 lines of Python code, with scripts to execute all test cases and compute the tables presented in the book; many of the charts and graphs can also be reproduced. The code is documented using Python docstring conventions, and code coverage is 95%, using *https://coverage.readthedocs.io*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning Algorithms: A Programmer's Guide to Writing Better Code* by George T. Heineman (O'Reilly). Copyright 2021 George T. Heineman, 978-1-492-09106-6."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, filenames, file extensions, and points I want to emphasize.

`Constant width`
: Used for program listings as well as within paragraphs to refer to program elements such as variable or function names, data types, statements, and keywords.

This element, identified by an image of a ring-tailed lemur, is a tip or suggestion. I use this image because lemurs have a combined visual field of up to 280°, which is a wider visual field than anthropoid primates (such as humans). When you see this tip icon, I am literally asking you to open your eyes wider to learn a new fact or Python capability.

This element, identified by an image of a crow, signifies a general note. Numerous researchers have identified crows to be intelligent, problem-solving animals—some even use tools. I use these notes to define a new term or call your attention to a useful concept that you should understand before advancing to the next page.

This element, identified by an image of a scorpion, indicates a warning or caution. Much like in real life, when you see a scorpion, stop and look! I use the scorpion to call attention to key challenges you must address when applying algorithms.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/learn-algorithms*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://youtube.com/oreillymedia*

Acknowledgments

For me, the study of algorithms is the best part of computer science. Thank you for giving me the opportunity to present this material to you. I also want to thank my wife, Jennifer, for her support on yet another book project, and my two sons, Nicholas and Alexander, who are now both old enough to learn about programming.

My O'Reilly editors—Melissa Duffield, Sarah Grey, Beth Kelly, and Virginia Wilson—improved the book by helping me organize the concepts and its explanations. My technical reviewers—Laura Helliwell, Charlie Lovering, Helen Scott, Stanley Selkow, and Aura Velarde—helped eliminate numerous inconsistencies and increase the quality of the algorithm implementations and explanations. All defects that remain are my responsibility.

CHAPTER 1

Problem Solving

In this chapter, you will learn:

- Multiple algorithms that solve an introductory problem.
- How to consider an algorithm's performance on problem instances of size N.
- How to count the number of times a key operation is invoked when solving a given problem instance.
- How to determine order of growth as the size of a problem instance doubles.
- How to estimate *time complexity* by counting the number of key operations an algorithm executes on a problem instance of size N.
- How to estimate *space complexity* by determining the amount of memory required by an algorithm on a problem instance of size N.

Let's get started!

What Is an Algorithm?

Explaining how an algorithm works is like telling a story. Each algorithm introduces a novel concept or innovation that improves upon ordinary solutions. In this chapter I explore several solutions to a simple problem to explain the factors that affect an algorithm's performance. Along the way I introduce techniques used to analyze an algorithm's performance *independent of its implementation*, though I will always provide empirical evidence from actual implementations.

An *algorithm* is a step-by-step problem-solving method implemented as a computer program that returns a correct result in a predictable amount of time. The study of algorithms is concerned with both correctness (will this algorithm work for all input?) and performance (is this the most efficient way to solve this problem?).

Let's walk through an example of a problem-solving method to see what this looks like in practice. What if you wanted to find the largest value in an unordered list? Each Python list in Figure 1-1 is a *problem instance*, that is, the input processed by an algorithm (shown as a cylinder); the correct answer appears on the right. How is this algorithm implemented? How would it perform on different problem instances? Can you predict the time needed to find the largest value in a list of one million values?

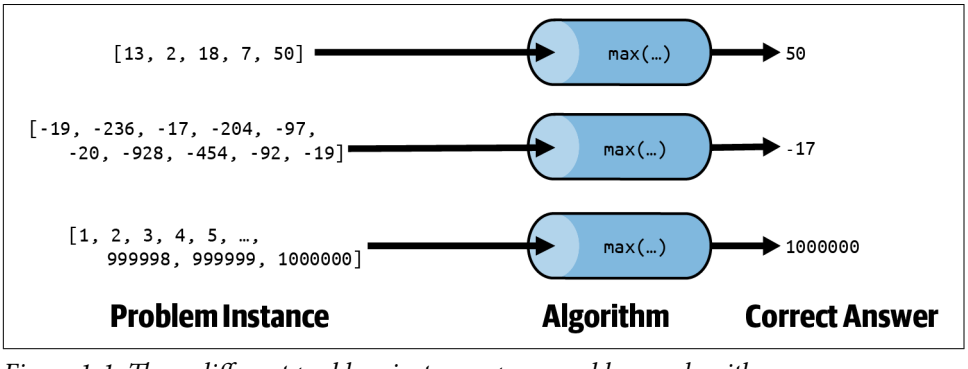


Figure 1-1. Three different problem instances processed by an algorithm

An algorithm is more than just a problem-solving method; the program also needs to complete in a predictable amount of time. The built-in Python function `max()` already solves this problem. Now, it can be hard to predict an algorithm's performance on problem instances containing random data, so it's worth identifying problem instances that are carefully constructed.

Table 1-1 shows the results of timing `max()` on two kinds of problem instances of size N: one where the list contains ascending integers and one where the list contains descending integers. While your execution may yield different results in the table, based on the configuration of your computing system, you can verify the following two statements:

- The timing for `max()` on ascending values is always slower than on descending values once N is *large enough*.
- As N increases ten-fold in subsequent rows, the corresponding time for `max()` also appears to increase ten-fold, with some deviation, as is to be expected from live performance trials.

For this problem, the maximum value is returned, and the input is unchanged. In some cases, the algorithm updates the problem instance directly instead of computing a new value—for example, sorting a list of values, as you will see in Chapter 5. In this book, N represents the size of a problem instance.

Table 1-1. Executing `max()` on two kinds of problem instances of size N (time in ms)

N	Ascending values	Descending values
100	`0.001`	`0.001`
1,000	`0.013`	`0.013`
10,000	`0.135`	`0.125`
100,000	`1.367`	`1.276`
1,000,000	`14.278`	`13.419`

When it comes to timing:

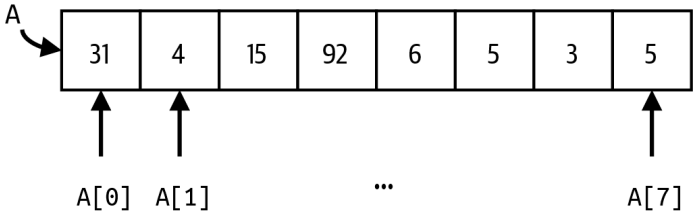
- You can't predict *in advance* the value of `T`(100,000)—that is, the time required by the algorithm to solve a problem instance of size 100,000—because computing platforms vary, and different programming languages may be used.
- However, once you empirically determine `T`(10,000), you can predict `T`(100,000)—that is, the time to solve a problem instance ten times larger—though the prediction will inevitably be inaccurate to an extent.

When designing an algorithm, the primary challenge is to ensure it is correct and works *for all input*. I will spend more time in Chapter 2 explaining how to analyze and compare the behavior of different algorithms that solve the exact same problem. The field of algorithm analysis is tied to the study of interesting, relevant problems that arise in real life. While the mathematics of algorithms can be challenging to understand, I will provide specific examples to always connect the abstract concepts with real-world problems.

The standard way to judge the efficiency of an algorithm is to count how many *computing operations* it requires. But this is exceptionally hard to do! Computers have a central processing unit (CPU) that executes *machine instructions* that perform mathematical computations (like add and multiply), assign values to CPU registers, and compare two values with each other. Modern programming languages (like C or C++) are compiled into machine instructions. Other languages (like Python or Java) are compiled into an intermediate *byte code* representation. The Python *interpreter* (which is itself a C program) executes the byte code, while built-in functions, such as `min()` and `max()`, are implemented in C and ultimately compiled into machine instructions for execution.

The Almighty Array

An *array* stores a collection of N values in a contiguous block of memory. It is one of the oldest and most dependable data structures programmers use to store multiple values. The following image represents an array of eight integers.



The array `A` has eight values indexed by their location. For example, `A[0]` = 31, and `A[7]` = 5. The values in `A` can be of any type, such as strings or more complicated objects.

The following are important things to know about an array:

- The first value, `A[0]`, is at index position 0; the last is `A[N-1]`, where N is the size of the array.
- Each array has a fixed length. Python and Java allow the programmer to determine this length at runtime, while C does not.
- One can read or update an individual location, `A[i]`, based on the *index* position, `i`, which is an integer in the range from 0 to N – 1.
- An array cannot be extended (or shrunk); instead, you allocate a new array of the desired size and copy old values that should remain.

Despite their simplicity, arrays are an extremely versatile and efficient way to structure data. In Python, `list` objects can be considered an array, even though they are more powerful because they can grow and shrink in size over time.

It is nearly impossible to count the total number of executed machine instructions for an algorithm, not to mention that modern day CPUs can execute *billions* of instructions per second! Instead, I will count the number of times a *key operation* is invoked for each algorithm, which could be "the number of times two values in an array are compared with each other" or "how many times a function is called." In this discussion of `max()`, the key operation is "how many times the *less-than* (<) operator is invoked." I will expand on this counting principle in Chapter 2.

Now is a good time to lift up the hood on the `max()` algorithm to see why it behaves the way it does.

Finding the Largest Value in an Arbitrary List

Consider the flawed Python implementation in Listing 1-1 that attempts to find the largest value in an arbitrary list *containing at least one value* by comparing each value in `A` against `my_max`, updating `my_max` as needed when larger values are found.

Listing 1-1. Flawed implementation to locate largest value in list

```
def flawed(A):
  my_max = 0          ❶
  for v in A:         ❷
    if my_max < v:
      my_max = v      ❸
  return my_max
```

❶ `my_max` is a variable that holds the maximum value; here `my_max` is initialized to 0.

❷ The `for` loop defines a variable `v` that iterates over each element in `A`. The `if` statement executes once for each value, `v`.

❸ Update `my_max` if `v` is larger.

Central to this solution is the less-than operator (<) that compares two numbers to determine whether a value is smaller than another. In Figure 1-2, as `v` takes on successive values from `A`, you can see that `my_max` is updated three times to determine the largest value in `A`. `flawed()` determines the largest value in `A`, invoking less-than six times, once for each of its values. On a problem instance of size N, `flawed()` invokes less-than N times.

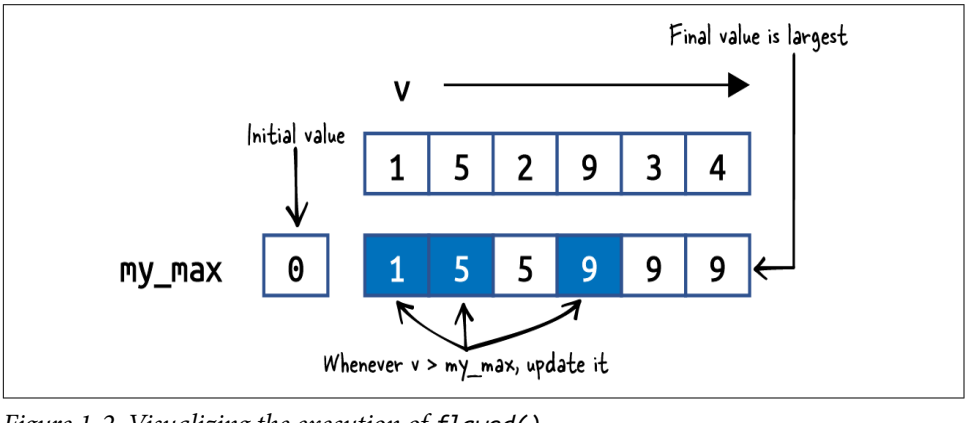


Figure 1-2. Visualizing the execution of `flawed()`

This implementation is flawed because it assumes that at least one value in `A` is greater than 0. Computing `flawed([-5,-3,-11])` returns 0, which is incorrect. One common fix is to initialize `my_max` to the smallest possible value, such as `my_max = float('-inf')`. This approach is still flawed since it would return this value if `A` is the empty list `[]`. Let's fix this defect now.

The Python statement `range(x,y)` produces the integers from `x` up to, but not including, `y`. You can also request `range(x,y,-1)`, which produces the integers from `x` counting down to, but not including, `y`. Thus `list(range(1,7))` produces `[1,2,3,4,5,6]`, and `list(range(5,0,-1))` produces `[5,4,3,2,1]`. You can count by arbitrary increments, thus `list(range(1,10,2))` produces `[1,3,5,7,9]` using a difference of 2 between values.

Counting Key Operations

Since the largest value must actually be contained in `A`, the correct `largest()` function in Listing 1-2 selects the first value of `A` as `my_max`, checking other values to see if any value is larger.

Listing 1-2. Correct function to find largest value in list

```
def largest(A):
  my_max = A[0]                  ❶
  for idx in range(1, len(A)):   ❷
    if my_max < A[idx]:
      my_max = A[idx]            ❸
  return my_max
```

❶ Set `my_max` to the first value in `A`, found at index position 0.

❷ `idx` takes on integer values from 1 up to, but not including, `len(A)`.

❸ Update `my_max` if the value in `A` at position `idx` is larger.

If you invoke `largest()` or `max()` with an empty list, it will raise a `ValueError: list index out of range` exception. These run-time exceptions are programmer errors, reflecting a failure to understand that `largest()` requires a `list` with at least one value.

Now that we have a correct Python implementation of our algorithm, can you determine how many times less-than is invoked in this new algorithm? Right! N – 1 times. We have fixed the flaw in the algorithm and improved its performance (admittedly, by just a tiny bit).

Why is it important to count the uses of less-than? This is the key operation used when comparing two values. All other program statements (such as `for` or `while` loops) are arbitrary choices during implementation, based on the program language used. We will expand on this idea in the next chapter, but for now counting key operations is sufficient.

Models Can Predict Algorithm Performance

What if someone shows you a different algorithm for this same problem? How would you determine which one to use? Consider the `alternate()` algorithm in Listing 1-3 that repeatedly checks each value in `A` to see if it is larger than or equal to all other values in the same list. Will this algorithm return the correct result? How many times does it invoke less-than on a problem of size N?

Listing 1-3. A different approach to locating largest value in `A`

```
def alternate(A):
  for v in A:
    v_is_largest = True          ❶
    for x in A:
      if v < x:
        v_is_largest = False     ❷
        break
    if v_is_largest:
      return v                   ❸
  return None                    ❹
```

❶ When iterating over `A`, assume each value, `v`, could be the largest.

❷ If `v` is smaller than another value, `x`, stop and record that `v` is not greatest.

❸ If `v_is_largest` is `true`, return `v` since it is the maximum value in `A`.

❹ If `A` is an empty list, return `None`.

`alternate()` attempts to find a value, `v`, in `A` such that no other value, `x`, in `A` is greater. The implementation uses two nested `for` loops. This time it's not so simple to compute how many times less-than is invoked, because the inner `for` loop over `x` stops as soon as an `x` is found that is greater than `v`. Also, the outer `for` loop over `v` stops once the maximum value is found. Figure 1-3 visualizes executing `alternate()` on our list example.

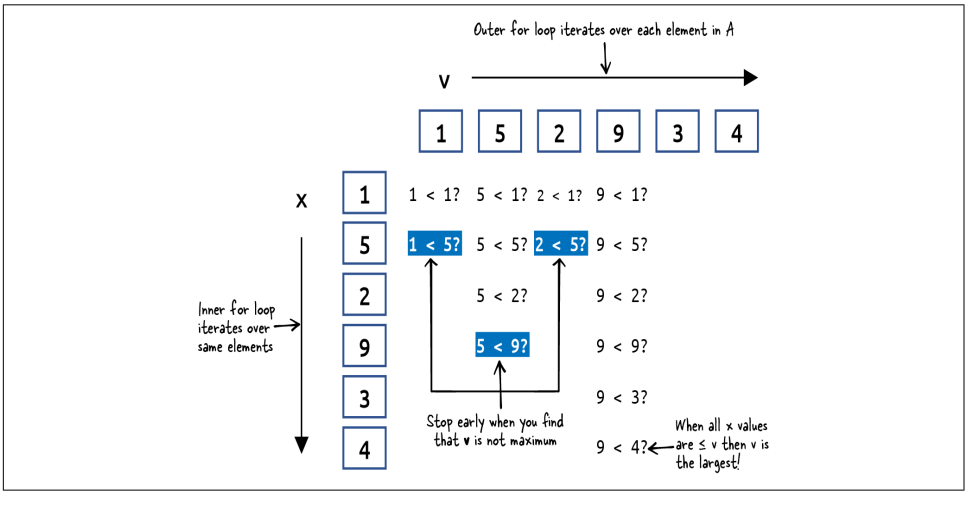


Figure 1-3. Visualizing the execution of `alternate()`

For this problem instance, less-than is invoked 14 times. But you can see that this total count depends on the specific values in the list `A`. What if the values were in a different order? Can you think of an arrangement of values that requires the least number of less-than invocations? Such a problem instance would be considered a *best case* for `alternate()`. For example, if the first value in `A` is the largest of all N values, then the total number of calls to less-than is always N. To summarize:

Best case
: A problem instance of size N that requires the least amount of work performed by an algorithm

Worst case
: A problem instance of size N that demands the most amount of work

Let's try to identify a *worst case* problem instance for `alternate()` that requires the most number of calls to less-than. More than just ensuring that the largest value is the last value in `A`, in a *worst case* problem instance for `alternate()`, the values in `A` must appear in ascending order.

Figure 1-4 visualizes a *best case* on the top where `p = [9,5,2,1,3,4]` and a *worst case* on the bottom where `p = [1,2,3,4,5,9]`.

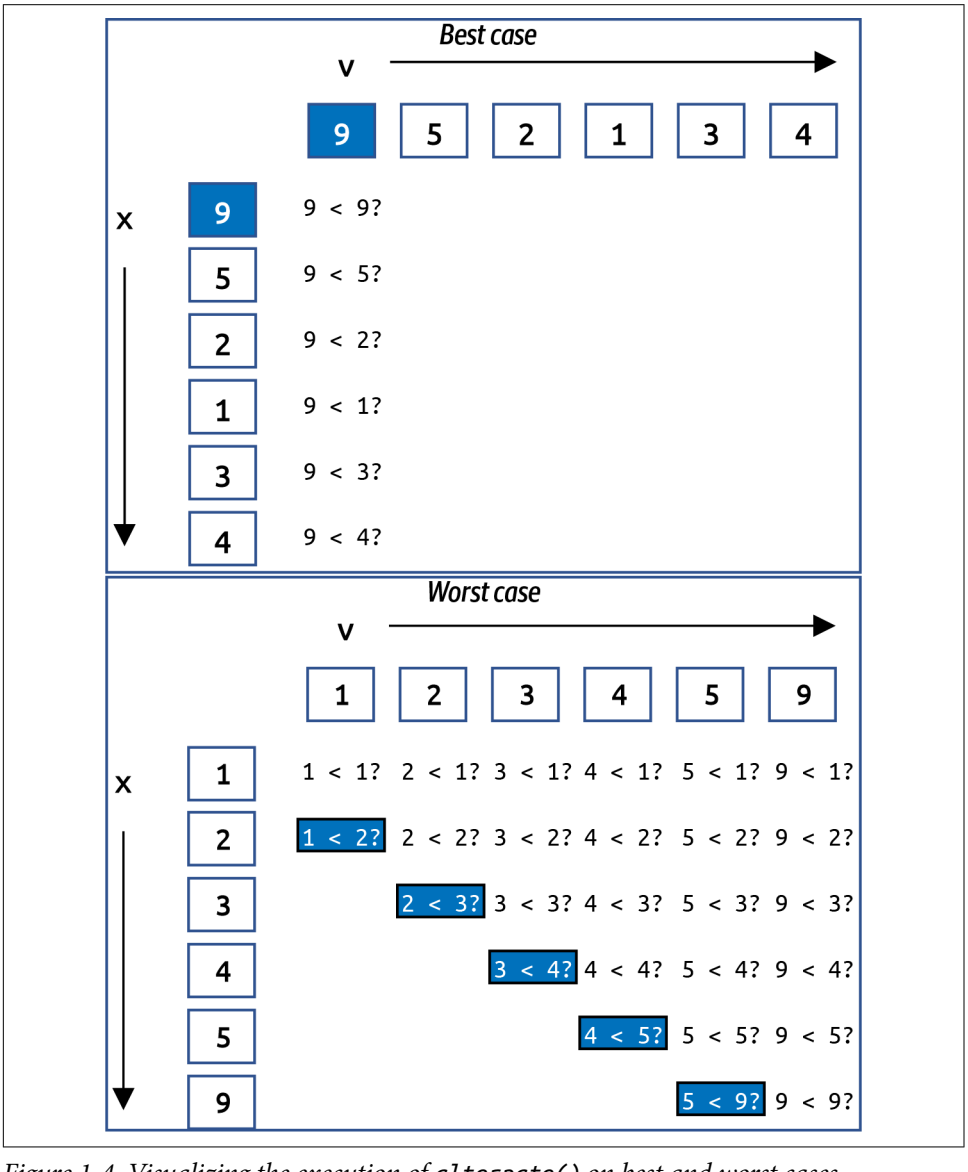


Figure 1-4. Visualizing the execution of `alternate()` on best and worst cases

In the *best case*, there are six calls to less-than; if there were N values in a *best case*, then the total number of invocations to less-than would be N. It's a bit more complicated for the *worst case*. In Figure 1-4 you can see there are a total of 26 calls to less-than when the list of N values is in ascending sorted order. With a little bit of mathematics, I can show that for N values, this count will always be $(N^2 + 3N - 2)/2$.

Table 1-2 presents empirical evidence on `largest()` and `alternate()` on *worst case* problem instances of size N.

Table 1-2. Comparing `largest()` with `alternate()` on worst case problem instances

N	Largest (# less-than)	Alternate (# less-than)	Largest (time in ms)	Alternate (time in ms)
8	7	43	0.001	0.001
16	15	151	0.001	0.003
32	31	559	0.002	0.011
64	63	2,143	0.003	0.040
128	127	8,383	0.006	0.153
256	255	33,151	0.012	0.599
512	511	131,839	0.026	2.381
1,024	1,023	525,823	0.053	9.512
2,048	2,047	2,100,223	0.108	38.161

For small problem instances, it doesn't seem bad, but as the problem instances double in size, the number of less-than invocations for `alternate()` essentially quadruples, far surpassing the count for `largest()`. The next two columns in Table 1-2 show the performance of these two implementations on 100 random trials of problem instances of size N. The completion time for `alternate()` quadruples as well.

I measure the time required by an algorithm to process random problem instances of size N. From this set of runs, I select the quickest completion time (i.e., the smallest). This is preferable to simply averaging the total running time over all runs, which might skew the results.

Throughout this book, I am going to present tables, like Table 1-2, containing the total number of executions of a key operation (here, the less-than operator) as well as the runtime performance. Each row will represent a different problem instance size, N. Read the table from top to bottom to see how the values in each column change as the problem instance size doubles.