The Algorithm Codex

Alejandro Piad-Morffis, Ph.D.

2026-01-18

Table of contents

Preface

Welcome to The Algorithm Codex.

This book is a repository of common algorithms used in all areas of computer science. It contains reference implementations in Python for many well-known (and some not-so-much) algorithms spanning from simple linear search to sorting, graphs, computational geometry, data structures, flow networks, game theory, number theory, optimization, and many other fields.

We wrote this book to serve as a complement for the main bibliography in a typical Computer Science major. You will not find comprehensive theory of algorithms in this book, or detailed analyses. However, we do present some basic intuitions into why most of the presented algorithms work and a back-of-the-envelope cost analysis whenever it makes sense.

The order in which algorithms are presented is our best attempt to build the most complex ideas on top of the simpler ones. We start with basic algorithms that everyone can understand and progressively move towards the more advanced.

The algorithms are presented in a literate programming format. The gist is that we combine code and prose in the best possible way to maximize understanding. Actually, the source code is generated from the book source, and not the other way around–that is what literate programming is, after all. Accompanying this book, you will find an open source repository with the exact implementations in this book.

You can read the book online at https://matcom.github.io/codex/.

Content of the Book

The Algorithm Codex is organized into several major parts, designed to take you from foundational concepts to specialized domains and the limits of computation:

- **Searching and Sorting**: We establish the core intuitions of algorithmic efficiency by exploring how to find and organize data in linear and logarithmic time.
- **Fundamental Data Structures**: We implement essential abstractions—including linked lists, stacks, queues, and hash tables—that serve as the building blocks for more complex systems.

- **Trees**: This part covers hierarchical data, from binary search trees to self-balancing structures and specialized variants like heaps and tries.
- **Graphs**: A significant section dedicated to relational data, covering traversals, shortest paths, spanning trees, and flow networks.
- **Dynamic Programming and Greedy Algorithms**: We delve into powerful paradigms for solving optimization problems by exploiting subproblem structure and local optimality.
- **Specialized Domains**: We explore deep subregions of Computer Science, including computational geometry, number theory, and game theory.
- **Advanced Complexity**: The book concludes with the frontiers of computation, exploring NP-completeness, approximation algorithms, and randomized approaches.

About the coding style

The code in this book is written in Python 3, specifically the 3.13 version.

We make extensive use of Python's generic syntax to write clean but fully typed methods that leverage the best and most modern practices in software development. Other than that, the code is often written in the simplest possible way that works. We don't make unnecessary optimizations like taking bounds out of a loop. On the other hand, our code is optimized in the algorithmic sense; it is fast because it exploits the inherent structure of the problem.

Since most of our code is pure, functional algorithms, we often rely on public, plain Python functions. We thus have very few classes, and the ones we have are very simple, often nothing but data stores. However, we do make heavy use of protocols and abstract classes, especially those in the Python standard library like sequences, maps, and queues.

Support The Algorithm Codex

This book is free, as in free beer and free speech, and it will always be.

The book content is licensed CC-BY-NC-SA, that means you are free to share the book in any format (HTML, ePUB, PDF) with anyone, and produce any derivatives you want, as long as you also share those freely for posterity.

The source code is licensed MIT, and thus you can use the algorithms implemented here for anything, including classes, academic work, but also writing commercial software.

The only thing you cannot do is resell the book itself or any derivative work like lecture notes, translations, etc.

If you want to support this effort, the best way to do is to buy the official PDF.

Stay in touch

Most of the chapters in this book are first published as standalone articles in The Computist Journal. Subscribing there is the best way to stay in the loop and get early access to most of the material.

Part I

Searching and Sorting

1 Basic Search

Searching is arguably the most important problem in Computer Science. In a very simplistic way, searching is at the core of critical applications like databases, and is the cornerstone of how the internet works.

However, beyond this simple, superficial view of searching as an end in itself, you can also view search as means for general-purpose problem solving. When you are, for example, playing chess, what your brain is doing is, in a very fundamental way, *searching* for the optimal move–the only that most likely leads to winning.

In this sense, you can view almost all of Computer Science problems as search problems. In fact, a large part of this book will be devoted to search, in one way or another.

In this first chapter, we will look at the most explicit form of search: where we are explicitly given a set or collection of items, and asked to find one specific item.

We will start with the simplest, and most expensive kind of search, and progress towards increasingly more refined algorithms that exploit characteristics of the input items to minimize the time required to find the desired item, or determine if it's not there at all.

1.1 Linear Search

Let's start by analyzing the simplest algorithm that does something non-trivial: linear search. Most of these algorithms work on the simplest data structure that we will see, the sequence.

A sequence (`Sequence` class) is an abstract data type that represents a collection of items with no inherent structure, other than each element has an index.

```
from typing import Sequence
```

Linear search is the most basic form of search. We have a sequence of elements, and we must determine whether one specific element is among them. Since we cannot assume anything at all from the sequence, our only option is to check them all.

```python
def find[T](x:T, items: Sequence[T]) -> bool:
    for y in items:
        if x == y:
            return True

    return False
```

Our first test will be a sanity check for simple cases:

```python
from codex.search.linear import find

def test_simple_list():
    assert find(1, [1,2,3]) is True
    assert find(2, [1,2,3]) is True
    assert find(3, [1,2,3]) is True
    assert find(4, [1,2,3]) is False
```

The `find` method is good to know if an element exists in a sequence, but it doesn't tell us *where*. We can easily extend it to return an *index*. We thus define the `index` method, with the following condition: if `index(x,l) == i` then `l[i] == x`. That is, `index` returns the **first** index where we can find a given element `x`.

```python
def index[T](x: T, items: Sequence[T]) -> int | None:
    for i,y in enumerate(items):
        if x == y:
            return i

    return None
```

When the item is not present in the sequence, we return `None`. We could raise an exception instead, but that would force a lot of defensive programming.

Let's write some tests!

```python
from codex.search.linear import index

def test_index():
    assert index(1, [1,2,3]) == 0
    assert index(2, [1,2,3]) == 1
    assert index(3, [1,2,3]) == 2
    assert index(4, [1,2,3]) is None
```

As a final step in the linear search paradigm, let's consider the problem of finding not the first, but *all* occurrences of a given item. We'll call this function `count`. It will return the number of occurrences of some item `x` in a sequence.

```python
def count[T](x: T, items: Sequence[T]) -> int:
    c = 0

    for y in items:
        if x == y:
            c += 1

    return c
```

Let's write some simple tests for this method.

```python
from codex.search.linear import count

def test_index():
    assert count(1, [1,2,3]) == 1
    assert count(2, [1,2,2]) == 2
    assert count(4, [1,2,3]) == 0
```

1.2 Min and Max

Let's now move to a slightly different problem. Instead of finding one specific element, we want to find the element that ranks minimum or maximum. Consider a sequence of numbers in an arbitrary order. We define the minimum (maximum) element as the element `x` such as `x <= y` (`x >= y`) for all `y` in the sequence.

Now, instead of numbers, consider some arbitrary total ordering function `f`, such that `f(x,y) <= 0` if and only if `x <= y`. This allows us to extend the notion of minimum and maximum to arbitrary data types.

Let's formalize this notion as a Python type alias. We will define an `Ordering` as a function that has this signature:

```python
from typing import Callable

type Ordering[T] = Callable[[T,T], int]
```

Now, to make things simple for the simplest cases, let's define a default ordering function that just delegates to the items own `<=` implementation. This way we don't have to reinvent the wheel with numbers, strings, and all other natively comparable items.

```
def default_order(x, y):
    if x < y:
        return -1
    elif x == y:
        return 0
    else:
        return 1
```

Let's write the `minimum` method using this convention. Since we have no knowledge of the structure of the sequence other than it supports partial ordering, we have to test all possible items, like before. But now, instead of returning as soon as we find the "correcOf course, we t" item, we simply store the minimum item we've seen so far, and return at the end of the `for` loop. This guarantees we have seen all the items, and thus the minimum among them must be the one we have marked.

```
from codex.types import Ordering, default_order

def minimum[T](items: Sequence[T], f: Ordering[T] = None) -> T:
    if f is None:
        f = default_order

    m = None

    for x in items:
        if m is None or f(x,m) <= 0:
            m = x

    return m
```

The `minimum` method can fail only if the `items` sequence is empty. In the same manner, we can implement `maximum`. But instead of coding another method with the same functionality, which is not very DRY, we can leverage the fact that we are passing an ordering function that we can manipulate.

Consider an arbitrary ordering function `f` such `f(x,y) <= 0`. This means by definition that `x <= y`. Now we want to define another function `g` such that `g(y,x) <= 0`, that is, it *inverts* the result of `f`. We can do this very simply by swaping the inputs in `f`.

```python
def maximum[T](items: Sequence[T], f: Ordering[T] = None) -> T:
    if f is None:
        f = default_order

    return minimum(items, lambda x,y: f(y,x))
```

We can easily code a couple of test methods for this new functionality.

```python
from codex.search.linear import minimum, maximum

def test_minmax():
    items = [4,2,6,5,7,1,0]

    assert minimum(items) == 0
    assert maximum(items) == 7
```

1.3 Binary Search

Now that we have started considering ordered sets, we can introduce what is arguably the most beautiful algorithm in the history of Computer Science: binary search. A quintessential algorithm that shows how a well-structured search space is exponentially easier to search than an arbitrary one.

To build some intuition for binary search, let's consider we have an *ordered* sequence of items; that is, we always have that if `i < j`, then `l[i] <= l[j]`. This simple constraint introduces a very powerful condition in our search problem: if we are looking for `x`, and `x < y`, then we know no item after `y` in the sequence can be `x`.

Convince yourself of this simple truth before moving on.

This fundamentally changes how fast we can search. Why? Because now every test that we perform–every time we ask whether `x < y` for some `y`–we gain *a lot* of information, not only about `y`, but about *every other item greater or equal than* `y`.

This is the magical leap we always need in order to write a fast algorithm–fast as in, it doesn't need to check *every single thing*. We need a way to gather more information from every operation, so we have to do less operations. Let's see how we can leverage this powerful intuition to make search not only faster, but *exponentially faster* when items are ordered.

Consider the set of items $x_1, ..., y, ..., x_n$. We are searching for item x, and we choose to test whether $x \leq y$. We have two choices, either $x \leq y$, or, on the contrary, $x > y$. We want to gain the maximum amount of information in either case. The question is, how should we pick y?

If we pick y too close to either end, we can get lucky and cross off a large number of items. For example if y is in the last 5% of the sequence, and it turns out $x > y$, we have just removed the first 95% of the sequence without looking at it! But of course, we won't get *that* lucky too often. In fact, if x is a random input, it could potentially be anywhere in the sequence. Under the fairly mild assumption that x should be uniformly distributed among all indices in the sequences, we will get *this* lucky exactly 5% of the time. The other 95! we have almost as much work to do as in the beginning.

It should be obvious by now that the best way to pick y either case is to choose the middle of the sequence. In that way I always cross off 50% of the items, regardless of luck. This is good, we just removed a huge chunk. But it gets even better.

Now, instead of looking for x linearly in the remaining 50% of the items, we do the exact same thing again! We take the middle point of the current half, and now we can cross off another 25% of the items. If we keep repeating this over and over, how fast will we be left with just one item? Keep that thought in mind.

Before doing the math, here is the most straightforward implemenation of binary search. We will use two indices, `l(eft)` and `r(ight)` to keep track of the current sub-sequence we are analyzing. As long as `l <= r` there is at least one item left to test. Once `l > r`, we must conclude `x` is not in the sequence.

Here goes the code.

```python
from typing import Sequence
from codex.types import Ordering, default_order

def binary_search[T](
    x: T, items: Sequence[T], f: Ordering[T] = None
) -> int | None:
    if f is None:
        f = default_order

    l, r = 0, len(items)-1

    while l <= r:
        m = (l + r) // 2
        res = f(x, items[m])

        if res == 0:
            return m
        elif res < 0:
            r = m - 1
        else:
            l = m + 1
```

Here is a minimal test.

```python
from codex.search.binary import binary_search

def test_binary_search():
    items = [0,1,2,3,4,5,6,7,8,9]

    assert binary_search(3, items) == 3
    assert binary_search(10, items) is None
```

1.4 Bisection

Standard binary search is excellent for determining if an element exists, but it provides no guarantees about which index is returned if the sequence contains duplicates. In many applications—such as range queries or maintaining a sorted list—we need to find the specific boundaries where an element resides or where it should be inserted to maintain order.

This is the problem of **bisection**. We define two variants: `bisect_left` and `bisect_right`.

The `bisect_left` function finds the first index where an element `x` could be inserted while maintaining the sorted order of the sequence. If `x` is already present, the insertion point will be before (to the left of) any existing entries. Effectively, it returns the index of the first element that is not "less than" `x`.

```python
def bisect_left[T](
    x: T, items: Sequence[T], f: Ordering[T] = None
) -> int:
    if f is None:
        f = default_order

    l, r = 0, len(items)

    while l < r:
        m = (l + r) // 2
        if f(items[m], x) < 0:
            l = m + 1
        else:
            r = m

    return l
```

The logic here is subtle: instead of returning immediately when an element matches, we keep narrowing the window until `l` and `r` meet. By setting `r = m` when `items[m] >= x`, we ensure the right boundary eventually settles on the first occurrence.

Conversely, `bisect_right` (sometimes called `bisect_upper`) finds the last possible insertion point. If `x` is present, the index returned will be after (to the right of) all existing entries. This is useful for finding the index of the first element that is strictly "greater than" `x`.

```
def bisect_right[T](
    x: T, items: Sequence[T], f: Ordering[T] = None
) -> int:
    if f is None:
        f = default_order

    l, r = 0, len(items)

    while l < r:
        m = (l + r) // 2
        if f(x, items[m]) < 0:
            r = m
        else:
            l = m + 1

    return l
```

In this variant, we only move the left boundary `l` forward if `x >= items[m]`, which pushes the search toward the end of a block of identical values.

Since both functions follow the same halving principle as standard binary search, their performance characteristics are identical:

- **Time Complexity**: $O(\log n)$, as we halve the search space in every iteration of the `while` loop.
- **Space Complexity**: $O(1)$, as we only maintain two integer indices regardless of the input size.

To ensure these boundaries are calculated correctly, especially with duplicate elements, we use the following test cases:

```
from codex.search.binary import bisect_left, bisect_right

def test_bisection_boundaries():
    # Sequence with a "block" of 2s
    items = [1, 2, 2, 2, 3]
```

```python
    # First index where 2 is (or could be)
    assert bisect_left(2, items) == 1

    # Index after the last 2
    assert bisect_right(2, items) == 4

    # If element is missing, both return the same insertion point
    assert bisect_left(1.5, items) == 1
    assert bisect_right(1.5, items) == 1

def test_bisection_extremes():
    items = [1, 2, 3]
    assert bisect_left(0, items) == 0
    assert bisect_right(4, items) == 3
```

Part II

Fundamental Data Structures

Part III

Trees

Part IV

Graphs

Part V

Dynamic Programming and Greedy Algorithms

Part VI

Specialized Domains

Part VII

Advanced Complexity