The Algorithm Codex

Alejandro Piad-Morffis, Ph.D.

2026-01-18

Table of contents

Preface **4**
Content of the Book 4
About the coding style 5
Support The Algorithm Codex 5
Stay in touch 6

1 Foundations **7**
1.1 What is an Algorithm? 7
1.2 Measuring Efficiency 7

I Searching and Sorting **8**

2 Basic Search **9**
2.1 Linear Search 9
2.2 Indexing and Counting 10
2.3 Min and Max 11
2.4 Conclusion 13

3 Efficient Search **14**
3.1 Binary Search 15
3.2 Bisection 16
3.3 Conclusion 18

4 Basic Sorting **19**
4.1 Selection Sort 19
4.2 Insertion Sort 20
4.3 The Geometry of Inversions 21
4.4 Bubble Sort 21
4.5 Verification 22
4.6 Conclusion 22

II Fundamental Data Structures **23**

III Trees **24**

IV String Matching **25**

V Graphs **26**

VI Dynamic Programming and Greedy Algorithms **27**

VII Specialized Domains **28**

VIIIAdvanced Complexity **29**

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.
- **String Algorithms**: We focus on pattern matching and text processing, covering algorithms from exact matching (KMP, Boyer-Moore) to advanced suffix structures.
- **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.

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.

1 Foundations

Before we begin our journey through specific algorithms, we must establish the ground on which we stand. To study algorithms is to study the limits of what can be computed and the cost of doing so.

1.1 What is an Algorithm?

At its simplest, an algorithm is a finite sequence of well-defined, computer-implementable instructions to solve a class of specific problems. However, in this Codex, we view an algorithm as a **mathematical object**–a strategy that exploits the structure of data to achieve an outcome efficiently.

Every algorithm we implement must satisfy three fundamental properties: 1. **Finiteness**: It must terminate after a finite number of steps. 2. **Definiteness**: Each step must be precisely defined. 3. **Effectiveness**: Each operation must be basic enough to be performed exactly. 4. **Correctness**: The algorithm must always produce the correct output for every valid input.

1.2 Measuring Efficiency

In Computer Science, we rarely care about how many seconds an algorithm takes on a specific machine. Instead, we care about how the effort scales as the input size n grows. This is the essence of **Big O Notation**.

- $O(1)$ **- Constant Time**: The "speed of light." No matter how big the universe gets, the time remains the same.
- $O(\log n)$ **- Logarithmic Time**: The "magical" scaling. Even if you double the input, you only add one extra step. This is the gold standard for search.
- $O(n)$ **- Linear Time**: The "honest day's work." To know everything about n items, you must look at n items.
- $O(n^2)$ **- Quadratic Time**: The "barrier." As we will see in the sorting chapter, this often arises when we compare every item with every other item.

Part I

Searching and Sorting

2 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.

2.1 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.

```
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:

```
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
```

2.2 Indexing and Counting

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`.

```
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!

```
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.

```
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.

```
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
```

2.3 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:

```
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`.

```
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.

```
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
```

2.4 Conclusion

Linear search is a powerful paradigm precisely because it is universal. Whether we are checking for the existence of an item, finding its index, or identifying the minimum or maximum element in a collection, the exhaustive approach provides absolute certainty. No matter the nature of the data, if we test every single element and skim through every possibility, the problem will be solved.

The primary drawback of this certainty is the cost: some search spaces are simply too vast to be traversed one item at a time. To achieve better performance, we must move beyond the assumption of an unstructured sequence. We need to know more about the search space and impose some level of structure.

In the next chapter, we will explore the most straightforward structure we can impose: order. We will see how knowing the relative position of items allows us to implement what is arguably the most efficient and beautiful algorithm ever designed.

3 Efficient 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.

3.1 Binary Search

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.

```
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.

```
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
```

3.2 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`.

```
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]

    # 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
```

3.3 Conclusion

Searching is arguably the most important problem in Computer Science. In this first chapter, we have only scratched the surface of this vast field, but in doing so, we have discovered one of the fundamental truths of computation: structure matters–a lot.

When we know nothing about the structure of our problem or the collection of items we are searching through, we have no choice but to rely on exhaustive methods like linear search. In these cases, we must check every single item to determine if it is the one we care about.

However, as soon as we introduce some structure–specifically, some *order*–the landscape changes completely. Binary search allows us to exploit this structure to find an element as fast as is theoretically possible, reducing our workload from a linear progression to a logarithmic one.

This realization that we can trade a bit of organizational effort for a massive gain in search efficiency is the perfect segue for our next chapter. If searching is easier when items are ordered, then we must understand the process of establishing that order. We must talk about **sorting**.

4 Basic Sorting

As we discovered in the previous chapter, structure is the secret ingredient that makes computation efficient. Searching through an unordered collection is a tedious, linear process, but searching through an ordered one is exponentially faster.

Sorting is the process of establishing this order. Formally, we want to take a sequence of items and rearrange them into a new sequence where, for any two indices i and j, if $i < j$, then $x_i < x_j$. In this chapter, we explore the most fundamental ways to achieve this, building our intuition from simple observation to a deeper structural analysis of the sorting problem itself.

4.1 Selection Sort

The most intuitive way to sort a list is to think about the destination. If we are building a sorted sequence, the very first element (index 0) *must* be the minimum element of the entire collection. Once that is in place, the second element (index 1) *must* be the minimum of everything that remains, and so on.

In Selection Sort, we stand at each position in the array and ask: **"Which element belongs here?"** To answer, we scan the unsorted portion of the list, find the minimum, and swap it into our current position.

```
from typing import MutableSequence
from codex.types import Ordering, default_order

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

    n = len(items)
    for i in range(n):
        # Find the index of the minimum element in the unsorted suffix
        min_idx = i
```

```
        for j in range(i + 1, n):
            if f(items[j], items[min_idx]) < 0:
                min_idx = j

        # Swap it into the current position
        items[i], items[min_idx] = items[min_idx], items[i]
```

We implement this by directly searching for the minimum index in each iteration rather than calling a helper function, keeping the logic self-contained. Because we must scan the remaining items for every single position in the list, we perform roughly $1+2+3+...(n-1) = n(n-1)/2$ comparisons, leading to a time complexity of $O(n^2)$.

4.2 Insertion Sort

We can flip the narrative of Selection Sort. Instead of standing at a position and looking for the right element, we can take an element and look for its right position. This is how most people sort a hand of cards.

We assume that everything behind our current position is already sorted. We take the next element and ask: **"How far back must I move this element so that the sequence remains sorted?"** We shift it backward, swapping it with its predecessor, until it finds its rightful place.

```
def insertion_sort[T](
    items: MutableSequence[T], f: Ordering[T] = None
) -> None:
    if f is None:
        f = default_order

    for i in range(1, len(items)):
        j = i
        # Move the element backward as long as it is smaller than its predecessor
        while j > 0 and f(items[j], items[j-1]) < 0:
            items[j], items[j-1] = items[j-1], items[j]
            j -= 1
```

The first element is sorted by definition. The second element either stays put or moves before the first. The third moves until it is in the correct spot relative to the first two. In the worst case (a reverse-sorted list), this also results again in $O(n^2)$ operations, but it is remarkably efficient for lists that are already "nearly sorted."

4.3 The Geometry of Inversions

To understand why these algorithms are all quadratic in complexity, we need to look at the structure of "unsortedness." Whenever a list in unsorted, is because we can find at least a couple of elements that are out of place. This means some $x_i > x_j$ where $i < j$. We define an **inversion** as any pair of such items. A sorted list has zero inversions.

With this idea in place, we can see sorting as "just" the problem of reducing the number of inversions down to zero. Any algorithm that does progress towards reducing the number of inversions is actually sorting. And a crucial insight is that there can be *at most* $O(n^2)$ inversions in any sequence of size n.

Selection sort reduces up to n inversions with each swap, that is, all inversions relative to the current minimum element. But every swap requires up to n comparisons, so we get $O(n^2)$ steps. Insertion sort reduces at most one inversion each step, by moving one item forward, thus it will require $O(n^2)$ steps to eliminate that many inversions.

4.4 Bubble Sort

Let's now build an algorithm based on this idea of eliminating inversions directly. A first guess could be, lets try to find a pair of inverted items and swap them. But we must be careful, if we do indiscrimantely, we might end up fixing one inversion but creating other inversions.

However, a powerful idea that we won't formally proof is that if a list has *any* inversions, there must be at least one inversion between two *consecutive* elements. If we fix these local inversions, we eventually fix them all. And fixing an inversion between consecutive elements cannot create new inversions. This is the heart of **Bubble Sort**: we repeatedly step through the list and swap adjacent items that are out of order.

```
def bubble_sort[T](
    items: MutableSequence[T], f: Ordering[T] = None
) -> None:
    if f is None:
        f = default_order

    n = len(items)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            # If we find a consecutive inversion, fix it
            if f(items[j+1], items[j]) < 0:
                items[j], items[j+1] = items[j+1], items[j]
```

```
                swapped = True

        # If no swaps occurred, the list is already sorted
        if not swapped:
            break
```

4.5 Verification

To ensure these three fundamental sorting approaches work as intended, we can run them against a set of standard cases.

```
import pytest
from codex.sort.basic import selection_sort, insertion_sort, bubble_sort

@pytest.mark.parametrize("sort_fn", [selection_sort, insertion_sort, bubble_sort])
def test_sorting_algorithms(sort_fn):
    items = [4, 2, 7, 1, 3]
    sort_fn(items)
    assert items == [1, 2, 3, 4, 7]

    # Already sorted
    items = [1, 2, 3]
    sort_fn(items)
    assert items == [1, 2, 3]

    # Reverse sorted
    items = [3, 2, 1]
    sort_fn(items)
    assert items == [1, 2, 3]
```

4.6 Conclusion

Selection, Insertion, and Bubble sort are all $O(n^2)$ algorithms. The reason is structural: in the worst case, a list of size can have $O(n^2)$ inversions. Since each swap in these algorithms only fixes one inversion at a time–in the best case–we are forced to perform a quadratic number of operations.

To break this ceiling and reach the theoretical limit of $O(n^2)$, we need to be more clever. We need algorithms that can fix *many* inversions with a single operation. This "divide and conquer" approach will be the focus of our next chapter: **Efficient Sorting**.

Part II

Fundamental Data Structures

Part III

Trees

Part IV

String Matching

Part V

Graphs

Part VI

Dynamic Programming and Greedy Algorithms

Part VII

Specialized Domains

Part VIII

Advanced Complexity