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/.

What is this book about?

The Algorithm Codex is designed as a multifaceted resource for a broad spectrum of the computational community. **Students** will find it to be a vital bridge between theoretical university lectures and the concrete reality of modern implementations, serving as a complement to standard academic curricula.

For **professors**, the book provides a library of clean, pedagogical code that emphasizes structural clarity over complexity. **Researchers** can utilize these implementations as a reliable reference for standard algorithmic logic when prototyping new ideas, while **developers** will gain a deeper understanding of the underlying abstractions that power their daily tools.

Ultimately, even **enthusiasts** driven by curiosity will find the logical elegance of these algorithms accessible through the lens of literate programming.

It is equally important to clarify what this project represents by stating that this is **not a programming book**; we assume that the reader is already proficient in Python 3 and understands the fundamental principles of software development. We deliberately avoid **pseudo-code** in favor of actual, runnable Python to ensure that no logic is lost in translation.

Furthermore, this work is **not an algorithm design book** and should not be viewed as a replacement for comprehensive theoretical texts such as *Introduction to Algorithms*. We do not offer **formal proofs of correctness** or exhaustive mathematical analysis, choosing instead to build intuition through back-of-the-envelope cost estimates and prose.

Finally, the Codex is **not a production cookbook** meant for direct copy-pasting into high-stakes environments; our code is optimized for readability and educational insight rather than the micro-optimizations required for production-level software.

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.

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.

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.

What is an Algorithm?

At its simplest, an algorithm is a mechanical procedure that takes an input and produces an output. However, in this Codex, we view an algorithm as a **formal mathematical object**–a precise strategy that exploits the structure of data to achieve an outcome efficiently.

To be considered a valid algorithm in our context, a procedure must satisfy several key characteristics:

- **Finiteness**: The description of the algorithm itself must be finite. Furthermore, for any valid input, the algorithm must always finish within a finite amount of time, for any given input.
- **Correctness**: The algorithm must always produce the correct answer for every valid input within its problem class.
- **Definiteness**: An algorithm is a formal procedure. It must be described in a language that admits no ambiguity regarding the operations to be performed. Historically, this has been achieved through mathematical notation; in this book, we use the **Python programming language**.

Most academic texts rely on *pseudo-code*–a high-level, informal description of an algorithm. While pseudo-code is useful for broad strokes, it often hides subtle complexities and can be interpreted in multiple ways.

In **The Algorithm Codex**, we deliberately avoid pseudo-code in favor of actual, runnable **Python 3.13**. By using a real programming language, we ensure that every operation is precisely defined and that the implementations you see are ready to be tested, scrutinized, and executed. This approach removes the "translation layer" between theory and practice, making the logic transparent and absolute.

Analyzing Algorithms

Following the definition of what an algorithm is, we must establish a framework for evaluating them. Once an implementation is complete, the work of a computer scientist is only beginning. We must subject our solution to a rigorous three-step analysis to ensure it is not just a working program, but a complete solution to a computational problem.

When we finish writing an algorithm, we must ask ourselves three fundamental questions. Only when all three are answered can we consider our work in computer science truly satisfied.

Is it correct?

The first and most critical question is whether the algorithm always produces the expected output for any valid input. This is the property of **correctness**. While this book avoids exhaustive formal proofs, we will always strive to provide a deep, intuitive explanation of why the logic holds. We focus on the underlying mechanics of the algorithm and how it handles **corner cases**—those extreme or unusual inputs where many naive solutions fail.

How efficient is it?

Once we are certain the algorithm is correct, we must quantify its cost. We ask: **How efficient is this in terms of time and space?**. Using the scaling intuition of Big O notation, we analyze how the algorithm's resource requirements grow as the input size n increases . We look for the "bottlenecks" in the logic and determine whether the primary cost comes from the number of operations performed or the amount of memory consumed.

Is it optimal?

The final question is perhaps the most profound: **Is this the most efficient algorithm possible, or can there be a better one?** We are not just looking for a "fast" algorithm; we are looking for the theoretical limits of the problem itself. Throughout this Codex, we will try to provide intuitive proofs of **optimality**—explaining why a certain complexity (like O(nlogn) for comparison-based sorting) cannot be improved upon.

When we can prove—even intuitively—that we have reached the optimal efficiency for a correct algorithm, we have solved the problem completely. At that point, the computational task is no longer a mystery; it is a solved piece of the science of computation.

Measuring Efficiency

Once we have established that an algorithm is correct, we must ask how much it "costs" to run. In this book, we care about two primary resources: **Time** (the number of operations performed) and **Space** (the amount of memory required).

However, hardware changes so rapidly that it is rarely useful to talk about performance in terms of seconds or megabytes. To remain hardware-agnostic, we use an idealized computational model.

In this book, and in most algorithmic analysis, we utilize the **Random Access Machine (RAM) model**. This model provides a controlled environment where we can precisely describe the number of steps an algorithm takes by assuming a **unitary unit of cost** for basic operations.

In the RAM model, we assume:

- **Unitary Operation Cost**: Basic operations—such as arithmetic, variable assignment, and method calls—all cost exactly one unit of time.
- **Discrete Memory Cells**: Memory is divided into discrete cells, each capable of holding one unit of data (such as a number, a character, or sometimes a small string).
- **Constant Access Time**: We can access any memory cell directly with a unitary cost. This is the "Random Access" from which the model takes its name; we can jump to any random location in memory without paying a penalty for distance.

Of course, this is an abstraction. In a real computer, multiplication is more expensive than addition, floating-point numbers carry additional costs, and memory is structured into complex layers of cache. However, the RAM model works exceptionally well for comparing algorithms in the abstract because it glosses over details that are often unimportant in the grand scheme of complexity. It only begins to break down in specialized areas, such as **numerical algorithms**, where the exact cost of multiplications versus additions or the precise layout of numbers in memory becomes critical to performance.

Furthermore, we rarely care about the absolute number of steps. Knowing that a specific sort takes exactly 1,024 operations is less useful than knowing how that cost grows as the input size n increases.

The core of algorithmic analysis is to look at how an algorithm time or memory cost *scales* with data. For example, an algorith that checks all items in a list exactly once scales *linearly*, which means if you double the size of the input, you expect the running time to double. However, an algorithm that scales *quadratically* with the input size–for example, if you compare each item in a list with all others–has a very different behavior: if you double the input size, that algorithm *quadruples* its runnign time.

The reason we care about scaling behavior rather than actual runtime cost is thus threefold. First, it lets us reason about the efficiency of two different algorithms regardless of the hardware. If my algorithm scales better than yours, they will both be faster on fast hardware, and slower on slow hardware, but mine will beat yours in every ocasion. No need to discuss which hardware to buy to decide here.

But more importantly, if my algorithm is written with poor optimizations or in a slower language–like Python–but yours is written in C++, you might get an edge on small instances

because you can run a tight loop in one milisecond while I need ten miliseconds to do the same. However, as the input data becomes larger and larger, there is a point after which your super optimized quadratic algorithm will always be worse than my lazy linear algorithm. This shouldn't be a justification to write lazy algorithms, but it does tells us we should focus on improving the high-level asymptotic complexity before low-level optimization tricks.

And finally, as time goes by, we expect hardware to improve, and thus we hope to tackle bigger and bigger problems with the same algorithms. If my algorithm scales linearly, next year when I get access to a twice-as-fast computer, I expect to solve a twice-as-big problem with the same resources (time and memory). However, if my algorithm scales quadratically, I have to wait until I get a computer four-times-as-fast to tackle a twice-as-big problem.

Formalizing scaling behavior

Thus, scaling is what we care about. To formalize this notion we use something called **asymptotic notation**, which looks like this. If we want to say an algorithm scales *roughly linearly* with input size, we say its running time (or memory) cost is $O(n)$.

Formally, this means the running time (or memory) can be expressed as some function $f(n)$ that grows as slow or slower than the linear function $g(n) = n$. In mathematical terms, we say there exists a constant c and an input size n_0 such that for all $n > n_0$ we have $f(n) <= c \cdot g(n)$.

The nice thing about this formulation is that it lets us gloss over all the tiny details of an algorithm and talk just about the rough growth rate. It is easy to prove–although we won't do it–that in asymptotic analysis we can throw away constants and lower order terms, and just keep the higher order function. For example, if some algorithm has a time cost of $f(n) = 3n+2$, that is still $O(n)$.

In this book, however, we won't concern ourselves too much with being strict at complexity analysis. For the most part, we will rely on intuitions like a single for loop is $O(n)$ and a double-nested loop is $O(n^2)$. However, for some algorithms we will need to perform a slightly more nuanced analysis to arrive at asymptotic cost functions like $O(n \log n)$ which are neither linear nor quadratic, and have their own and very interesting scaling behavior.

Final Words

Now that we have settled our expectations, you are ready to start the journey. It will be fast-paced but–I hope–really exciting. We will discover many algorithms, close to a hundred of them! And in each case, we will ask ourselves these same three questions. And, surprisingly often, we will be able to answer them pretty well!

Part I

Searching and Sorting

This first part of **The Algorithm Codex** serves as our entry point into the science of computation. We begin with the most fundamental of tasks: finding and organizing data. While these problems may seem elementary, they reveal the deepest truth of computer science: **structure is the primary driver of efficiency**.

In this part, we transition from the exhaustive "brute force" methods of linear search to the elegant, logarithmic precision of binary search, and from the quadratic complexity of basic sorting to the theoretical limits of divide-and-conquer algorithms.

What We Will Explore

Through these chapters, we move from simple observations to a rigorous structural analysis of search spaces and the "geometry of inversions":

- **Basic Search**: We start with the universal but expensive paradigm of linear search, establishing the baseline for what happens when we know nothing about our data.
- **Efficient Search**: We introduce binary search and bisection, demonstrating how an ordered search space allows us to gain the maximum possible information from every comparison.
- **Fundamental Sorting**: We analyze Selection, Insertion, and Bubble sort, learning why is the natural ceiling for algorithms that fix only one or two inversions at a time.
- **Efficient Sorting**: We break the quadratic barrier using Merge Sort and Quick Sort, exploring the power of recursion to fix multiple inversions simultaneously through divide-and-conquer strategies.
- **Order Statistics**: We solve the problem of selection—finding the -th smallest item—by leveraging partitioning logic to achieve linear time performance.
- **Linear Time Sorting**: We demonstrate that the limit can be bypassed entirely if we exploit domain-specific constraints, such as the discrete nature of integers, through Counting and Radix sort.

What You Will Learn

By following this progression, you will develop the "algorithmic intuition" required to analyze and solve increasingly complex problems:

1. **The Information Gain Principle**: Why halving the search space leads to exponential efficiency gains.
2. **Divide and Conquer**: How to break a monolithic problem into independent subproblems that are easier to solve and combine.
3. **The Geometry of Inversions**: Understanding "unsortedness" as a structural property that can be measured and methodically reduced.

4. **Randomization as a Strategy**: How to use probabilistic approaches to avoid pathological cases and ensure robust average-case performance.
5. **The Power of Constraints**: Why knowing the range or type of your input allows for optimizations that are mathematically impossible in a generic context.

Searching and sorting are not just utility functions; they are the playground where we learn the rules of algorithmic negotiation. We are learning how much effort we must expend to impose order, and how much that order pays us back in search speed.

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.

```
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
```

Analyzing Linear Search

Once we have an implementation, we must subject it to the three-step analysis established in our foundations.

Is it correct?

The property of **correctness** ensures that for any valid input, the algorithm produces the expected output. For linear search, we can verify this through three increasingly formal lenses:

- **The Exhaustive Argument**: Suppose an element x exists in the sequence. By definition, there is some index i such that `items[i] == x`. Since the algorithm performs an equality test over every single index in the sequence without exception, it is logically impossible to miss the item if it is there.
- **The Inductive Argument**: We can reason about the algorithm's correctness across different input sizes. For a sequence of length 0, the loop never executes, and the algorithm correctly returns `False`. Assume the algorithm works for a sequence of length n. For a sequence of length $n+1$, the target x is either in the first n elements—where the inductive hypothesis ensures we find it—or it is the $n+1$-th element, which we check in the final iteration. If it is in neither, the algorithm correctly concludes it is not present.

- **The Loop Invariant**: We can define a formal invariant for the `for` loop: *At the start of iteration i, the element x has not been found in the first $i-1$ elements of the sequence.* By the time the loop completes at iteration n, if the function hasn't returned `True`, we know with certainty that x is not in the first n elements, which constitutes the entire sequence.

How efficient is it?

We analyze linear search using the **RAM model**, assuming each comparison and iteration step has a unitary cost.

- **Time Complexity**: In the worst-case scenario (the item is at the very end or not present at all), we must perform n comparisons for a sequence of size n. This gives us a growth rate of $O(n)$, or **linear time**.
- **Space Complexity**: The algorithm only requires a constant amount of extra memory to store the loop variable and the target, regardless of the input size, resulting in $O(1)$ **space complexity**.

Is it optimal?

Intuitively, linear search must be **optimal for unstructured data**. If we know *nothing* about the order or distribution of the elements, we are mathematically forced to look at every single item at least once to be certain x is not there. Any algorithm that skipped an element could be "fooled" if that specific element happened to be the one we were looking for. Thus, for a generic sequence, $O(n)$ is the best possible lower bound.

To prove this more formally, we employ an **adversarial argument**, a powerful technique in complexity theory where we imagine a game between our algorithm and a malicious adversary.

- **The Adversary's Strategy**: Suppose an algorithm claims to find an element x (or prove its absence) by examining fewer than n elements—say, $n-1$ elements. The adversary waits for the algorithm to finish its $n-1$ checks.
- **The "Trap"**: Because there is one element the algorithm did not inspect, the adversary is free to define that specific element as x if the algorithm concludes "False," or as something other than x if the algorithm concludes "True" without having seen it.
- **The Conclusion**: Since the adversary can always change the unexamined element to make the algorithm's answer wrong, any correct algorithm *must* inspect every element in the worst case.

This proves that the lower bound for searching an unstructured sequence is $\Omega(n)$. Linear search, which operates in $O(n)$, meets this lower bound exactly, making it a **tightly optimal** solution for the problem as defined. Unless we possess more information about the data's structure—the central theme of the next chapter—we simply cannot do better.

1.2 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
```

Analysis

We won't dwell too much in this section since the analysis is very similar to linear search–these are just specialized versions of it. Once more, we have $O(n)$ algorithms (with $O(1)$ memory cost) for a problem that is provable $\Omega(n)$. Thus, given our assumptions (that there is no intrinsic structure to the elements order), we have optimal algorithms.

1.3 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 correct 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] = default_order) -> T:
    m = items[0]

    for x in items:
        if 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] = default_order) -> T:
    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
```

The correctness, cost, and optimality analysis is very similar in these cases as well.

1.4 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.

2 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.

2.1 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
```

2.2 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
```

2.3 Binary Search on Predicates

The true power of binary search extends far beyond finding a number in a list. We can generalize the algorithm to find the "boundary" of any **monotonic predicate**.

A predicate p is monotonic if, once it becomes true for some index i, it remains true for all $j > i$. We can use binary search to find the smallest index i such that $p(i)$ is true. This is often called binary searching on the answer. Instead of searching through a physical collection of items, we are searching through an abstract **decision space**.

```
from typing import Callable

def find_first(
    low: int, high: int, p: Callable[[int], bool]
) -> int | None:
    """
    Finds the first index in [low, high] for which p(index) is True.
    Assumes p is monotonic: if p(i) is True, p(i+1) is also True.
    """
    ans = None
    l, r = low, high

    while l <= r:
        m = (l + r) // 2
        if p(m):
            ans = m
            r = m - 1
        else:
```

```
            l = m + 1

    return ans
```

Consider the problem of finding the integer square root of a very large number —that is, the largest integer such that . While we could use `math.sqrt`, binary search allows us to find this value using only integer arithmetic, which is vital in fields like cryptography or when dealing with arbitrary-precision integers.

Our predicate p is: "Is $x^2 > n$?" This is monotonic: if $x^2 > n$, then $(x+1)^2$ is certainly greater than n. By finding the *first* x where $x^x > n$, we know that $x - 1$ is our desired integer square root.

```
def integer_sqrt(n: int) -> int:
    if n < 0:
        raise ValueError("Square root not defined for negative numbers")
    if n < 2:
        return n

    # Find the first x such that x*x > n
    first_too_big = find_first(1, n, lambda x: x * x > n)

    return first_too_big - 1
```

This approach reveals a deep connection between **searching and optimization**. Many problems that ask for a "minimum possible x such that $p(x)$ is possible" can be solved by binary searching over the value of x, provided that the possibility p is monotonic relative to x.

Whenever you encounter a problem where a "yes" answer for a value x implies a "yes" for all values larger than x, you are no longer looking for an item—you are looking for a **threshold**. Binary search is the most efficient way to discover it.

Verification

We can verify this generalized search and its application to the integer square root problem with the following tests.

```
from codex.search.binary import find_first, integer_sqrt

def test_find_first():
    # Predicate: is the number >= 7?
    nums = [1, 3, 5, 7, 9, 11]
```

```
    # find_first returns the index
    idx = find_first(0, len(nums) - 1, lambda i: nums[i] >= 7)
    assert idx == 3
    assert nums[idx] == 7

def test_integer_sqrt():
    assert integer_sqrt(16) == 4
    assert integer_sqrt(15) == 3
    assert integer_sqrt(17) == 4
    assert integer_sqrt(0) == 0
    assert integer_sqrt(1) == 1
    assert integer_sqrt(10**20) == 10**10
```

2.4 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**.

3 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.

3.1 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 $O(n^2)$.

3.2 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."

3.3 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.

3.4 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
```

3.5 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]
```

3.6 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 $O(n \log n)$, 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**.

4 Efficient Sorting

This chapter marks our departure from the quadratic barrier. As we established previously, sorting is the process of eliminating inversions. While basic algorithms like Selection or Insertion sort remove inversions somewhat haphazardly or one at a time, **efficient sorting** relies on a structured, recursive strategy known as **Divide and Conquer**.

By imposing a rigid structure on how we approach these inversions, we can reduce the computational cost from $O(n^2)$ to the theoretical optimum of $O(n \log n)$.

To break the quadratic limit, we must find ways to fix multiple inversions with a single operation. The most effective way to do this is to stop looking at the sequence as a monolithic block and start viewing it as a composition of smaller, more manageable sub-problems.

4.1 Merge Sort: Intuition through Order

The fundamental insight behind **Merge Sort** is that it is remarkably easy to combine two sequences that are *already sorted*. If we have two sorted lists, we can merge them into a single sorted list in linear time by simply comparing the heads of each list and picking the smaller one.

The "conquer" part of the algorithm is this linear merge. The "divide" part is the recursive leap: if we don't have two sorted halves, we simply split our current list in two and call Merge Sort on each piece until we reach the base case—a list of a single element, which is sorted by definition.

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

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

    if len(items) <= 1:
        return
```

```
    mid = len(items) // 2
    left = items[:mid]
    right = items[mid:]

    merge_sort(left, f)
    merge_sort(right, f)

    # Merge the sorted halves back into items
    i = j = k = 0
    while i < len(left) and j < len(right):
        if f(left[i], right[j]) <= 0:
            items[k] = left[i]
            i += 1
        else:
            items[k] = right[j]
            j += 1
        k += 1

    while i < len(left):
        items[k] = left[i]
        i += 1
        k += 1

    while j < len(right):
        items[k] = right[j]
        j += 1
        k += 1
```

In Merge Sort, the structural strategy is to **fix inversions within each half first**. We recursively dive deep into the sequence, sorting smaller and smaller sub-sequences. Only when the sub-sequences are internally free of inversions do we perform the merge step to fix the "global" inversions between the two halves.

Because the list is halved at each step, the recursion depth is $log(n)$. Since we perform $O(n)$ work at each level of the tree to merge the results, the total complexity is $O(nogn)$.

4.2 Quick Sort: Sorting by Partitioning

While Merge Sort is elegant, it requires extra space to store the temporary halves during the merge process. **Quick Sort** offers a narrative shift: instead of splitting the list blindly in the middle and merging later, we **rearrange the items first** so that no merge is ever necessary.

This is achieved through **partitioning**. We pick an element called a **pivot** and rearrange the sequence so that every element smaller than the pivot moved to its left, and every element larger than the pivot moved to its right.

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

    def _quick_sort(low: int, high: int):
        if low < high:
            p = _partition(low, high)
            _quick_sort(low, p)
            _quick_sort(p + 1, high)

    def _partition(low: int, high: int) -> int:
        pivot = items[(low + high) // 2]
        i = low - 1
        j = high + 1
        while True:
            i += 1
            while f(items[i], pivot) < 0:
                i += 1
            j -= 1
            while f(items[j], pivot) > 0:
                j -= 1
            if i >= j:
                return j
            items[i], items[j] = items[j], items[i]

    _quick_sort(0, len(items) - 1)
```

The striking difference here is the order of operations. Quick Sort **fixes inversions between both halves first**. By partitioning, we ensure that there are zero inversions between the left "half" and the right "half" (no item in the left is greater than an item in the right). Once this global structure is established, we recursively fix the remaining inversions **within** each half.

This approach allows Quick Sort to operate **in-place**, requiring only $O(\log n)$ auxiliary space for the recursion stack. While its worst-case is $O(n^2)$, its average-case performance is a highly efficient $O(n \log n)$.

4.3 Verification

We can verify both algorithms using the same test suite we established for basic sorting.

```
import pytest
from codex.sort.efficient import merge_sort, quick_sort

@pytest.mark.parametrize("sort_fn", [merge_sort, quick_sort])
def test_efficient_sorting(sort_fn):
    items = [4, 2, 7, 1, 3]
    sort_fn(items)
    assert items == [1, 2, 3, 4, 7]

    # Handle duplicates and edge cases
    items = [2, 1, 2, 1]
    sort_fn(items)
    assert items == [1, 1, 2, 2]
```

4.4 Conclusion

Both Merge Sort and Quick Sort reaffirm our central theme: **structure matters**. By moving away from the unstructured swaps of Bubble Sort and adopting a rigorous divide-and-conquer strategy, we change how we interact with the geometry of inversions.

Whether we fix local inversions first (Merge Sort) or global ones first (Quick Sort), the result is a massive leap in efficiency. We have traded the simplicity of a double-loop for the power of recursion and structural partitioning.

5 Rankings and Selection

This chapter addresses the problem of selecting the -th smallest element in a sequence—a task that lies at the heart of calculating medians, percentiles, and other order statistics. While one could simply sort the entire collection in time and pick the element at index , we can do better by leveraging the same structural insights we gained from Quick Sort.

Finding the "rank" of an element—its position in a sorted version of the collection—is a fundamental operation in data analysis. The most common case is finding the **median**, the element that splits a set into two equal halves. By generalizing this, we look for the -th order statistic: the value that is greater than or equal to exactly elements.

5.1 Quick Select: Selection via Partitioning

The core of Quick Sort was the `partition` operation, which organized elements around a pivot. In Quick Sort, we recursively processed both sides of the partition. However, for selection, we only care about the side that contains our target index .

This leads to **Quick Select**. Because we discard one-half of the search space at every step, we aren't performing a "divide and conquer" so much as a "prune and search" strategy.

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

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

    if not 0 <= k < len(items):
        raise IndexError("Rank k is out of bounds")

    return _select(items, 0, len(items) - 1, k, f)
```

```
def _select[T](
    items: MutableSequence[T], low: int, high: int, k: int, f: Ordering[T]
) -> T:
    if low == high:
        return items[low]

    # Randomized pivot selection to ensure good average performance
    pivot_idx = random.randint(low, high)
    items[pivot_idx], items[high] = items[high], items[pivot_idx]

    p = _partition(items, low, high, f)

    if k == p:
        return items[p]
    elif k < p:
        return _select(items, low, p - 1, k, f)
    else:
        return _select(items, p + 1, high, k, f)

def _partition[T](
    items: MutableSequence[T], low: int, high: int, f: Ordering[T]
) -> int:
    pivot = items[high]
    i = low
    for j in range(low, high):
        if f(items[j], pivot) <= 0:
            items[i], items[j] = items[j], items[i]
            i += 1
    items[i], items[high] = items[high], items[i]
    return i
```

The probabilistic analysis of Quick Select is striking. In the worst case—where we consistently pick the worst possible pivot—the complexity is still $O(n^2)$. However, on average, the size of the search space follows a geometric series which converges to $O(n)$. This means that on average, Quick Select finds the k-th order statistic in $O(n)$ time.

5.2 Median of Medians: Deterministic Selection

While randomization is practically robust, we can achieve a guaranteed worst-case time using a clever, ad hoc approach to pivot selection known as the **Median of Medians** algorithm.

The goal is to find a pivot that is guaranteed to be "good enough"—meaning it is not too close to either end of the sorted sequence. We do this by:

1. Dividing the list into groups of five.
2. Finding the median of each small group.
3. Recursively finding the median of these medians.

This "median of medians" is then used as the pivot for a standard partition.

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

    def _get_pivot(sub_items: MutableSequence[T]) -> T:
        if len(sub_items) <= 5:
            return sorted(sub_items, key=lambda x: x)[len(sub_items) // 2]

        chunks = [sub_items[i:i + 5] for i in range(0, len(sub_items), 5)]
        medians = [sorted(c, key=lambda x: x)[len(c) // 2] for c in chunks]
        return _get_pivot(medians)

    # TODO: Use the median of medians to partition and select
    # (Implementation follows standard select logic using the calculated pivot)
    # ...
```

The structural beauty of this algorithm lies in the constant `5`. It is the smallest odd number that ensures the recursive step prunes enough of the search space to maintain a linear recurrence.

5.3 Verification

To ensure our selection logic holds, we test it by finding various ranks in both random and edge-case sequences.

```
import pytest
from codex.search.rank import quick_select

def test_selection():
    items = [3, 1, 2, 4, 0]
    # Finding the median (rank 2)
```

```
    assert quick_select(items[:], 2) == 2
    # Finding the minimum (rank 0)
    assert quick_select(items[:], 0) == 0
    # Finding the maximum (rank 4)
    assert quick_select(items[:], 4) == 4

def test_duplicates():
    items = [1, 2, 1, 2, 1]
    assert quick_select(items, 0) == 1
    assert quick_select(items, 4) == 2
```

5.4 Conclusion

With the implementation of selection algorithms, we have completed our initial survey of searching and sorting. The progression from $O(n)$ linear search to $O(n \log n)$ sorting, and finally back to $O(n)$ for selection, demonstrates the power of structured thinking. By understanding how to manipulate inversions and search spaces, we can find specific needles in increasingly large haystacks with mathematical precision.

Before moving on, we will briefly touch on one more subject: linear sorting. We will see how digging further down into specific structural constraints of the of the input data we can furhter refine our algorithms and make them even faster.

6 Linear Time Sorting

This chapter concludes our exploration of sorting by demonstrating that the limit we established is not a universal law, but a specific constraint of **comparison-based algorithms**. If we possess even deeper knowledge about the structure of our data—specifically that it consists of discrete values within a known range—we can bypass comparisons entirely and achieve true linear time performance.

In the previous chapters, we operated under the assumption that the only way to gain information about our items was to compare them. However, when our data has a predictable, bounded structure, we can use the values themselves as **indices**. This shifts the problem from "which is bigger?" to "how many of each value do we have?".

6.1 Counting Sort

Counting Sort is the purest expression of this idea. If we know that every element in a sequence is an integer in the range , we can simply count the occurrences of each integer. By calculating the cumulative sum of these counts, we determine the exact position each element should occupy in the final sorted array.

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

def counting_sort(
    items: MutableSequence[int], k: int
) -> List[int]:
    """
    Sorts a sequence of integers in the range [0, k].
    This implementation is stable.
    """
    counts = [0] * (k + 1)
    for x in items:
        counts[x] += 1

    # Transform counts into starting indices
    for i in range(1, k + 1):
```

```
        counts[i] += counts[i - 1]

    output = [0] * len(items)
    # Iterate backwards to maintain stability
    for x in reversed(items):
        counts[x] -= 1
        output[counts[x]] = x

    return output
```

Counting Sort performs exactly two passes over the input and one pass over the range $O(k)$. This results in a time complexity of $O(n+k)$. When $k = O(n)$, the algorithm is strictly linear. The cost, however, is **space complexity**: we require an auxiliary array of size k. If k is significantly larger than n, this becomes impractical.

6.2 Radix Sort: Sorting by Digits

To handle larger ranges without massive memory overhead, we use **Radix Sort**. The intuition here is to view each number (or string) as a sequence of "digits." We sort the collection multiple times, once for each digit position, starting from the **least significant digit** (LSD).

Crucially, each pass must be a **stable sort**. By using Counting Sort as the stable subroutine for each digit, we can sort numbers in any range by breaking them into digits.

```
def radix_sort(items: MutableSequence[int], base: int = 10) -> List[int]:
    if not items:
        return items

    max_val = max(items)
    exp = 1
    output = list(items)

    while max_val // exp > 0:
        output = _counting_sort_by_digit(output, exp, base)
        exp *= base

    return output

def _counting_sort_by_digit(items: List[int], exp: int, base: int) -> List[int]:
    counts = [0] * base
    for x in items:
```

```
        digit = (x // exp) % base
        counts[digit] += 1

    for i in range(1, base):
        counts[i] += counts[i - 1]

    res = [0] * len(items)
    for x in reversed(items):
        digit = (x // exp) % base
        counts[digit] -= 1
        res[counts[digit]] = x
    return res
```

Radix Sort runs in $O(d(n+k)) time, where d is the number of digits and k is the base. Because d is constant for a fixed word size (e.g., 32-bit or 64-bit integers), the algorithm remains linear relative to n. This is the preferred method for sorting large sets of integers or fixed-length strings where memory is constrained compared to the range of values.

6.3 Verification

We verify these linear approaches by testing them against various integer ranges and sequences.

```
import pytest
from codex.sort.linear import counting_sort, radix_sort

def test_counting_sort():
    items = [4, 1, 3, 4, 3]
    # Range is [0, 4]
    assert counting_sort(items, 4) == [1, 3, 3, 4, 4]

def test_radix_sort():
    items = [170, 45, 75, 90, 802, 24, 2, 66]
    expected = [2, 24, 45, 66, 75, 90, 170, 802]
    assert radix_sort(items) == expected
```

6.4 Conclusions

The algorithms in this chapter serve as a powerful reminder that **theoretical limits are often tied to specific constraints**. By moving away from the "black box" of comparison—where

we know nothing about items except their relative order—to a model where we exploit the internal structure of keys, we successfully broke the $O(n \log n)$ barrier.

Through **Counting Sort**, we saw how integers can be used directly as indices to map values to their final positions in $O(n + k)$ time. With **Radix Sort**, we extended this principle to larger ranges by decomposing keys into digits, maintaining linearity through multiple stable passes.

However, this efficiency is not a "free lunch." We have traded mathematical generality for **physical memory**, as these algorithms require auxiliary space proportional to the range of values ($O(k)$) or the base used for digits. Furthermore, they are only applicable to discrete, bounded data types.

This chapter concludes our survey of sorting by reaffirming the Codex's central thesis: **the more you know about the structure of your data, the more effectively you can master its complexity**. Having mastered the logic of search and order, we are now ready to explore how these abstract processes are grounded in the physical topology of memory in **Part II: Fundamental Data Structures**.

Lessons from Part I

As we conclude our exploration of searching and sorting, we move away from specific implementations to look at the meta-principles of algorithm design. The algorithms we have studied are more than just tools; they are demonstrations of how a programmer can negotiate with the laws of logic to extract performance.

The most profound lesson of this part is that **structure matters**. In an unstructured sequence, finding an item is a linear struggle against entropy. By imposing **order**, we transform the search space into a hierarchy of information where every comparison halves our remaining work.

This theme repeated in our transition from basic to efficient sorting. By recognizing the "Geometry of Inversions," we moved from haphazardly swapping neighbors to structured partitioning and merging. The lesson is universal: to make a process efficient, you must first organize the environment in which it operates.

We discovered that the "best" algorithm often depends on how much we are willing to assume about our data. The barrier for sorting is only a law for **comparison-based** logic. By imposing stricter conditions—such as requiring inputs to be integers in a known range—we unlocked **Counting Sort** and **Radix Sort**, achieving true linear time.

In algorithm design, **constraints are not limitations; they are opportunities**. The more you know about the structure of your input, the more aggressively you can optimize your solution.

Throughout this part, **recursion** has been our primary tool for managing complexity. Whether in the binary halving of a search space or the divide-and-conquer strategy of Merge Sort, recursion allows us to solve a problem by defining what it means to be "partially solved." It is the mathematical expression of delegation, allowing us to focus on the logic of a single step while the structure of the call stack handles the global coordination.

Finally, we introduced **randomization** as a strategic weapon. In **Quick Select**, we saw that while a deterministic worst-case can be expensive, a randomized approach can offer performance with a probability so high it effectively becomes a certainty. This represents a pragmatic shift in computational thinking: sometimes, the most "rational" path is to embrace the roll of the dice to avoid the pathological cases that break deterministic logic.

Part II

Fundamental Data Structures

In the first part of this Codex, we treated data largely as an abstract sequence—a collection of items that we could index, compare, and rearrange at will. We discovered that by imposing logical order, we could achieve remarkable algorithmic efficiency. However, algorithms do not exist in a vacuum; they must inhabit the physical reality of a computer's memory.

In this part, we shift our focus from the logic of the process to the topology of the data. Here, we explore how the way we organize information in memory—whether as a contiguous block or a web of pointers—determines the physical limits of the algorithms we write.

What We Will Explore

In this part, we move beyond the high-level abstractions provided by Python's built-in types to implement the essential structures that form the foundation of all modern software systems:

- **Memory and Sequences**: We begin by examining the trade-offs between contiguous memory (arrays) and linked structures, understanding why the physical layout of data is the primary driver of performance.
- **Linked Lists**: We implement the most fundamental non-contiguous structure, exploring how pointers allow for dynamic growth and efficient insertions at the cost of random access.
- **Stacks and Queues**: We build the primary abstractions for managing order and flow, implementing LIFO (Last-In, First-Out) and FIFO (First-In, First-Out) behaviors that are critical for everything from expression evaluation to task scheduling.
- **Hashing and Hash Tables**: We investigate the "magic" of constant-time access, learning how to bridge the gap between an arbitrary value and its physical location in memory through hash functions and collision resolution strategies.

What You Will Learn

By the end of this part, you will have moved from a theoretical understanding of algorithms to a practical mastery of their physical containers. Specifically, you will learn:

1. **The Physicality of Data**: Why the distinction between a pointer and an index is the most important decision in low-level system design.
2. **Trade-off Analysis**: How to weigh the benefits of fast insertion against the necessity of fast retrieval, moving beyond simple Big O notation to consider cache locality and memory overhead.
3. **Abstractions and Protocols**: How to use Python's modern type system and protocols to implement these structures in a way that is generic, reusable, and mathematically sound.

4. **Building Blocks**: How these simple structures—lists, stacks, and queues—serve as the indispensable components for the more complex trees and graphs we will encounter later in the Codex.

In the same way that a builder must understand the properties of wood, steel, and stone, a computer scientist must understand the properties of data structures. We are moving from the design of the strategy to the selection of the material.

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

A A Python Primer

This appendix provides a concise overview of the Python 3.13 features and syntax used throughout **The Algorithm Codex**. While this book is not an introductory programming text, this primer serves as a reference for the specific idioms and modern type-system features that enable our clean, algorithmic implementations.

Python is a high-level, interpreted language that prioritizes readability and expressiveness. In this Codex, we treat Python as a executable notation for algorithms, leveraging its modern type system to ensure our code is both correct and self-documenting.

A.1 Variables and Basic Types

Variables in Python are names that point to objects in memory. Unlike many lower-level languages, variables do not have fixed types; however, the objects they point to do.

```
# Integers and Floats
n: int = 42
pi: float = 3.14159

# Strings and Booleans
name: str = "Codex"
is_active: bool = True
```

In the Codex, we always provide type hints for variables in global or class scopes to maintain clarity.

A.2 Expressions and Statements

An **expression** is a piece of code that evaluates to a value (e.g., `2 + 2`), while a **statement** is an instruction that performs an action (e.g., an assignment or a function call).

Python supports standard arithmetic operators (`+`, `-`, `*`, `/`) and a specific operator for integer division (`//`), which we use extensively in binary search and partitioning algorithms to find middle indices.

A.3 Control Flow: Loops and Conditionals

We rely on two primary looping constructs to traverse data structures: `for` loops for iterating over sequences and `while` loops for processes that continue until a specific logical condition is met.

```
# Iterating over a sequence
for item in [1, 2, 3]:
    print(item)

# Conditional logic
if n > 0:
    # Do something
elif n < 0:
    # Do something else
else:
    # Default case
```

In our implementations of Selection Sort and Bubble Sort, we use `range()` to generate indices for controlled iteration.

A.4 Functions

Functions are the primary unit of work in this book. We prefer "pure" functions that take inputs, perform a transformation, and return a result without side effects.

```
def square(x: int) -> int:
    return x * x
```

A.5 Classes and Objects

While the Codex favors a functional style, we use classes as simple data containers or to implement specific protocols. Python 3.13 allows for clean class definitions that integrate seamlessly with the type system.

```
class Point:
    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y
```

A.6 Type Annotations and Modern Generics

The most important feature of the Codex's coding style is the use of **PEP 695 generics**, introduced in Python 3.12 and refined in 3.13. This allows us to write algorithms that work for any data type `T` while maintaining full type safety.

Instead of older `TypeVar` syntax, we use the elegant bracket notation:

```
# A generic function that works for any type T
def identity[T](value: T) -> T:
    return value

# Using the 'type' alias for complex definitions
type Ordering[T] = Callable[[T, T], int]
```

This syntax is used throughout our searching and sorting implementations to ensure that an algorithm designed for integers works just as correctly for strings or custom objects, provided they satisfy the required protocols.

A.7 The `Sequence` Protocol

In almost every chapter, we use `Sequence` or `MutableSequence` from the `typing` module. These are **Protocols**–they define what a type can *do* (like being indexed or having a length) rather than what it *is*. By using `Sequence[T]` instead of `list[T]`, our algorithms remain generic enough to work with lists, tuples, or custom array-like structures.

B Asymptotic Notation and Analysis

This appendix provides a formal grounding for the "scaling intuition" introduced in the foundations of the Codex. In the study of algorithms, we are less concerned with exact cycle counts and more with **asymptotic behavior**: how the resource requirements of an algorithm grow as the input size n approaches infinity.

To analyze algorithms rigorously, we use a set of mathematical notations that allow us to ignore constant factors and lower-order terms, focusing instead on the **rate of growth**.

B.1 The Big-O Family

Let $f(n)$ and $g(n)$ be functions mapping natural numbers to non-negative real numbers.

Big-O (Upper Bound)

We say that $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that:

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

This notation provides an asymptotic upper bound. If an algorithm is $O(n^2)$, it will perform no worse than quadratic time for large n.

Big-Ω (Lower Bound)

We say that $f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that:

$$0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0$$

Ω provides a lower bound, representing the minimum amount of work an algorithm must perform.

Big-Θ (Tight Bound)

We say that $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. This means there exist positive constants c_1, c_2, and n_0 such that:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0$$

Θ is the most descriptive notation, as it tells us the algorithm grows exactly like $g(n)$.

Little-o and Little-ω (Strict Bounds)

- $f(n) = o(g(n))$: The growth of $f(n)$ is strictly less than $g(n)$. Formally, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- $f(n) = \omega(g(n))$: The growth of $f(n)$ is strictly greater than $g(n)$. Formally, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

B.2 The Master Theorem

Many of the most efficient algorithms in this book (like Merge Sort) use a divide-and-conquer strategy. Their complexity is often expressed as a recurrence of the form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ is the number of subproblems, $b > 1$ is the factor by which the subproblem size is reduced, and $f(n)$ is the cost of work done outside the recursive calls. The Master Theorem provides a "cookbook" solution for such recurrences:

1. **If** $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. (Work is dominated by the leaves).
2. **If** $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$. (Work is balanced across levels).
3. **If** $f(n) = \Omega(n^{\log_b a + \epsilon})$ and satisfies the regularity condition ($af(n/b) \leq cf(n)$), then $T(n) = \Theta(f(n))$. (Work is dominated by the root).

B.3 The Hierarchy of Growth

Understanding the relative order of complexity functions is essential for selecting the right algorithm for a given problem size. Below is the standard hierarchy from slowest to fastest growth:

1. $O(1)$ — **Constant**: Accessing an array element, simple arithmetic.
2. $O(\log n)$ — **Logarithmic**: Binary search. The "gold standard."
3. $O(n)$ — **Linear**: Sequential scan.
4. $O(n \log n)$ — **Linearithmic**: Optimal comparison-based sorting (Merge Sort).
5. $O(n^2)$ — **Quadratic**: Nested loops (Bubble Sort).
6. $O(n^k)$ — **Polynomial**: General algorithmic complexity.
7. $O(2^n)$ — **Exponential**: Exhaustive search (Traveling Salesperson).
8. $O(n!)$ — **Factorial**: Permutations of a set.

As n grows, the gaps between these classes become astronomical. While a $O(n^2)$ algorithm might be acceptable for $n = 1,000$, it becomes entirely impractical for $n = 1,000,000$, where an $O(n \log n)$ or $O(n)$ solution remains trivial for modern hardware.