Sorting and Searching

A Gentle Introduction

Jeremy Evert

November 10, 2025

Contents

Chapter 1

Introduction

1.1 Why Sorting and Searching Matter

Hey friends! In this book we are going to explore two of the most common jobs we ask computers to do:

- **Searching**: “Is this value in my data? If so, where?”
- **Sorting**: “Please put this data in order.”

At first glance, both tasks sound simple. You have a bunch of numbers, and you either want to find one of them or line them up from smallest to largest. Easy, right?

The fun begins when we realize there are many different ways to search and sort, and some of those ways are dramatically faster than others as our data sets grow. That is where ideas like:

- **Big-O notation**,
- **algorithmic complexity**, and
- **careful performance analysis**

start to matter. By the end of this book, you will not only know how to write sorting and searching code, but also how to reason about *why* one approach is better than another.

In this first chapter, we will keep things very simple. We will:

1. write a basic **linear search** algorithm that looks for a number in a list of numbers (without sorting first), and

2. write a classic **bubble sort** algorithm that reorders a list so that later searching can be more structured.

We will gently hint at runtime complexity, but save the deeper Big-O discussion for later chapters.

1.2 A First Search: Linear Search

Imagine you have a small list of numbers on a sticky note:

$$[9,\ 3,\ 7,\ 2,\ 10]$$

and you want to know whether the number 7 is in the list. One straightforward strategy is:

1. Start at the first number.
2. Compare it to 7.
3. If it matches, you are done.
4. If it does not, move one step to the right and repeat.

You keep walking through the list *linearly*, one element at a time. This strategy is called **linear search**.

Here is a simple Python implementation:

Listing 1.1: A simple linear search in Python.

```
def linear_search(data, target):
    """
    Return the index of 'target' in the list 'data',
    or -1 if the target is not found.
    """
    for index, value in enumerate(data):
        if value == target:
            return index
    return -1


if __name__ == "__main__":
    numbers = [9, 3, 7, 2, 10]
    target = 7

```

```
    position = linear_search(numbers, target)
    if position != -1:
        print(f"Found {target} at index {position}.")
    else:
        print(f"{target} was not found.")
```

A few quick observations (we will formalize these ideas later):

- In the *best* case, the target is at the first position, so we only do one comparison.
- In the *worst* case, the target is at the very end of the list or not present at all, so we check every element.
- As the list gets longer, the number of checks grows roughly in proportion to the length of the list.

That "grows in proportion to the length" idea is the heart of what we will later call *linear time*, or $\mathcal{O}(n)$ time.

1.3 Sorting to Help Searching

Linear search works on any list, even if the elements are in a completely random order. The downside is that it can be slow for very large lists, because we may have to check every single element.

If, however, we put the data into *sorted order* first, we can sometimes use much faster searching techniques. For example, binary search (which we will meet soon) can find values in a sorted list in a way that scales much more efficiently than linear search.

So there is a trade-off:

- Sorting the data takes extra work up front.
- After sorting, searching can become much faster.

In this chapter, we will not yet optimize that trade-off. Instead, we will simply learn a very basic way to sort: bubble sort.

1.4 Our First Sort: Bubble Sort

Bubble sort is one of the simplest sorting algorithms to understand and implement, even though it is *not* the most efficient choice for large data sets. We study it because it gives us a clear, concrete example of how a sorting algorithm works.

The idea:

1. Look at neighboring pairs of elements in the list.
2. If a pair is out of order, swap them.
3. Keep sweeping through the list, pushing larger values toward the end, like bubbles rising to the surface.
4. Repeat these passes until no more swaps are needed.

Instead of writing the full code directly in this chapter, we store it in a separate Python file inside a `scripts` folder. This keeps our project organized and makes it easier to rerun experiments or change the code later.

Listing 1.2 shows the contents of `scripts/bubble_sort_basic.py`. This version of bubble sort does two important things for us:

- It prints the list after each full "bubble pass" so that we can see how the numbers move over time.
- It writes the same information to a CSV file (`data/bubble_sort_basic_trace.csv`) so that we can load it into a spreadsheet, plot graphs, or quote the exact output later in this book.

Listing 1.2: Bubble sort with a pass-by-pass trace, stored in `scripts/bubble_sort_basic.py`.

```
#!/usr/bin/env python3
"""
bubble_sort_basic.py

Bubble sort with a simple trace:

- Prints the list after each full bubble pass so you can
  see the numbers "bubbling" toward the end.
- Appends a CSV row for each pass, including a timestamp,
  so you can track when the data was generated.
"""

import csv
from pathlib import Path
from datetime import datetime
```

```


def bubble_sort_with_trace(data):
    """
    Perform bubble sort and record the list state after each pass.

    Returns:
        sorted_list: the sorted copy of the input list
        trace: a list of (pass_number, state_list) snapshots

    pass_number = 0 is the initial state before any passes.
    """
    arr = data[:]  # copy so we don't mutate the original list
    n = len(arr)
    trace = []

    # Record the initial (unsorted) state
    trace.append((0, arr[:]))

    for i in range(n):
        swapped = False

        # One "bubble pass"
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # Swap out-of-order neighbors
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        # Record the state of the list *after* this pass
        trace.append((i + 1, arr[:]))

        # If no swaps were made, the list is already sorted
        if not swapped:
            break

    return arr, trace


def print_trace(trace):
```

```python
    """
    Print a friendly view of how the list changes after each pass.
    """
    print("Bubble sort trace (state after each full pass):")
    for pass_number, state in trace:
        if pass_number == 0:
            label = "Start"
        else:
            label = f"Pass {pass_number}"
        print(f"{label:>6}: {state}")


def append_trace_to_csv(trace, csv_path: Path):
    """
    Append the bubble sort trace to a CSV file.

    Columns:
        timestamp, pass_number, state_list

    - timestamp: when this run was recorded
    - pass_number: 0 for initial state, 1, 2, ... for later passes
    - state_list: space-separated string version of the list
    """
    csv_path.parent.mkdir(parents=True, exist_ok=True)

    file_exists = csv_path.exists()
    run_timestamp = datetime.now().isoformat(timespec="seconds")

    with csv_path.open("a", newline="") as f:
        writer = csv.writer(f)

        # Write header only if this is a new file
        if not file_exists:
            writer.writerow(["timestamp", "pass_number", "state_list"])

        for pass_number, state in trace:
            state_str = " ".join(str(x) for x in state)
            writer.writerow([run_timestamp, pass_number, state_str])
```

```python


if __name__ == "__main__":
    # Example data; later chapters can experiment with other lists.
    numbers = [9, 3, 7, 2, 10]

    print("Original list:", numbers)
    sorted_numbers, trace = bubble_sort_with_trace(numbers)

    print()
    print_trace(trace)

    print()
    print("Sorted list:  ", sorted_numbers)

    # Write CSV file into ../data relative to this script
    base_dir = Path(__file__).resolve().parent
    csv_file = base_dir.parent / "data" / "bubble_sort_basic_trace.csv"

    append_trace_to_csv(trace, csv_file)
    print(f"\nTrace appended to {csv_file}")
```

1.5 Quoting the Data: Bubble Sort Trace CSV

Because the script writes its trace to a CSV file in the `data` directory, we can include that data directly in our book. This makes the book feel more like a living lab notebook: the text, the code, and the data all match each other.

Listing 1.3 is taken directly from `data/bubble_sort_basic_trace.csv` and shows how the list changes after each pass of bubble sort.

Listing 1.3: Trace data produced by `bubble_sort_basic.py`, stored in `data/bubble_sort_basic_trace.csv`.

```
pass_number,state_list
0,9 3 7 2 10
1,3 7 2 9 10
2,3 2 7 9 10
3,2 3 7 9 10
4,2 3 7 9 10
```

```
2025-11-10T18:17:12,0,9 3 7 2 10
2025-11-10T18:17:12,1,3 7 2 9 10
2025-11-10T18:17:12,2,3 2 7 9 10
2025-11-10T18:17:12,3,2 3 7 9 10
2025-11-10T18:17:12,4,2 3 7 9 10
```

Some early complexity intuition:

- In the worst case, bubble sort compares many pairs of elements over and over.
- As the number of elements n grows, the number of comparisons grows roughly like n^2.
- Later we will describe this more formally as $\mathcal{O}(n^2)$ time.

1.6 Where We Are Going Next

In this chapter we have:

- introduced the basic ideas of searching and sorting,
- written a simple linear search that works on unsorted data,
- implemented bubble sort to put data into order, and
- connected the code to actual trace data stored in a CSV file.

Next, we will:

- dig deeper into **Big-O notation** and what it means to say an algorithm runs in $\mathcal{O}(n)$ or $\mathcal{O}(n^2)$ time,
- compare different sorting algorithms, and
- explore faster search strategies that take advantage of sorted data.

For now, make sure you can trace both the linear search and the bubble sort by hand on a small list. Being able to follow each step is the first move toward truly understanding algorithmic complexity.

Chapter 2

Bubble Sort in the Wild: Timing and Big-O

In Chapter 1.2 we met bubble sort as a simple, easy-to-read sorting algorithm. In this chapter we turn it loose on larger inputs and watch how its running time grows.

Our goal is to connect three things:

1. the *code* that performs bubble sort and measures its running time,
2. the *data* we collect in a CSV file, and
3. the *Big-O* story that explains why the graph of time vs. input size curves upward like an n^2 function.

2.1 From Algorithm to Experiment

The bubble sort algorithm itself has not changed. What we are doing now is treating it like a lab experiment:

- pick a list size n (for example, 100, 200, 400, ...),
- generate a random list of n integers,
- sort that list using bubble sort, and
- measure how long the sort took.

We repeat this for several sizes n and several trials per size. Each run produces one data point: *"sorting n items took t seconds."* Those data points are written to a CSV file so that we can analyze and plot them later.

2.2 Timing Bubble Sort

Listing 2.1 shows the script `scripts/bubble_sort_timing.py`. This code focuses on running bubble sort as fast as it reasonably can and recording the total wall-clock time.

Listing 2.1: Timing bubble sort on growing input sizes.

```
#!/usr/bin/env python3
"""
bubble_sort_timing.py

Measure wall-clock time for bubble sort on lists of increasing size
and append the results to a CSV file.

Each row in the CSV has:
    timestamp, n_items, elapsed_seconds
"""

import csv
import random
import time
from datetime import datetime
from pathlib import Path


def bubble_sort(arr):
    """
    In-place bubble sort with early exit if the list is already sorted.
    No tracing, just sorting as fast as this simple algorithm allows.
    """
    n = len(arr)
    for i in range(n):
        swapped = False

        # After each pass, the largest element among the unsorted part
        # "bubbles" to the end of the list.
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

```

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


def time_bubble_sort(n, max_value=10**6):
    """
    Generate a random list of length n, sort it with bubble_sort,
    and return the elapsed wall-clock time in seconds.
    """
    data = [random.randint(0, max_value) for _ in range(n)]

    start = time.perf_counter()
    bubble_sort(data)
    end = time.perf_counter()

    return end - start


def append_result(csv_path: Path, n: int, elapsed: float):
    """
    Append a single timing result to the CSV file.

    Columns:
        timestamp, n_items, elapsed_seconds
    """
    csv_path.parent.mkdir(parents=True, exist_ok=True)
    file_exists = csv_path.exists()
    timestamp = datetime.now().isoformat(timespec="seconds")

    with csv_path.open("a", newline="") as f:
        writer = csv.writer(f)

        # Write a header row only if the file is new.
        if not file_exists:
            writer.writerow(["timestamp", "n_items", "elapsed_seconds"])

        writer.writerow([timestamp, n, f"{elapsed:.6f}"])
```

```python

def main():
    # Figure out where we are and where the data directory lives.
    base_dir = Path(__file__).resolve().parent
    csv_file = base_dir.parent / "data" / "bubble_sort_timing.csv"

    # List sizes to test. Feel free to tweak this for bigger/smaller runs.
    sizes = [100, 200, 400, 800, 1600, 3200, 6400, 10000, 20000, 30000, 40000]
    trials_per_size = 3  # Run multiple trials per size for smoother data.

    print("Bubble sort timing experiment")
    print(f"Results will be appended to: {csv_file}")
    print()

    for n in sizes:
        for trial in range(1, trials_per_size + 1):
            elapsed = time_bubble_sort(n)
            append_result(csv_file, n, elapsed)
            print(f"n = {n:5d}, trial = {trial}, time = {elapsed:.6f} seconds")

    print("\nDone. Data appended to CSV; ready for plotting in Chapter 2 and beyond.")


if __name__ == "__main__":
    main()
```

A few key points about this script:

- The function `bubble_sort` implements the same algorithm you saw in Chapter 1, but without any extra printing or tracing.
- The function `time_bubble_sort(n)` generates a random list of length n, sorts it, and returns the elapsed time.
- The `append_result` function adds a new row to `data/bubble_sort_timing.csv` with three fields: timestamp, number of items, and elapsed time in seconds.

- The `main()` function loops over a range of input sizes (100, 200, 400, ...) and runs several trials for each size.

This script is our experimental engine. Every time we run it, we append more timing data to the same CSV file.

2.3 Big-O Intuition for Bubble Sort

Before looking at the plot, let us reason about the shape we expect to see.

Bubble sort works by repeatedly sweeping through the list and comparing neighbor pairs:

- On the first pass, it may compare positions $(0, 1)$, $(1, 2)$, ..., up to $(n - 2, n - 1)$.
- On the second pass, it does almost as many comparisons, and so on.

If you imagine counting comparisons, the total number of neighbor comparisons is roughly:

$$(n - 1) + (n - 2) + (n - 3) + \cdots + 1$$

This is a classic triangular sum. Its exact value is $\frac{n(n-1)}{2}$, which behaves like $\frac{1}{2}n^2$ for large n. In Big-O notation we say:

$$T(n) \in \mathcal{O}(n^2)$$

because, up to constant factors, the running time grows like n^2.

So if we double n, we should expect the running time to grow by about a factor of four:

$$T(2n) \approx 4\,T(n).$$

Our CSV data from `bubble_sort_timing.py` lets us see whether the actual wall-clock time behaves the way this n^2 theory predicts.

2.4 Fitting an n^2 Curve

To make the connection concrete, we wrote a second script that reads the CSV file, groups runs by input size, and computes the average time for each n. Then it fits a curve of the form

$$T(n) \approx an^2 + b$$

to the data, and also builds an "ideal" $\mathcal{O}(n^2)$ curve kn^2 that passes through the smallest data point.

Listing 2.2 shows `scripts/bubble_sort_analyze.py`.

Listing 2.2: Analyzing and plotting bubble sort timing data.

```python
#!/usr/bin/env python3
"""
bubble_sort_analyze.py

Read bubble_sort_timing.csv, compute average time for each n,
fit a quadratic curve, and compare it to an ideal O(n^2) curve.

Outputs:
    figures/bubble_sort_timing_n2.png
"""

import csv
from collections import defaultdict
from pathlib import Path

import numpy as np
import matplotlib.pyplot as plt


def load_timing_data(csv_path: Path):
    """
    Load timing data from CSV and group times by n_items.

    Returns:
        n_values (sorted list of ints)
        avg_times (list of floats, same order as n_values)
    """
    times_by_n = defaultdict(list)

    with csv_path.open("r", newline="") as f:
        reader = csv.DictReader(f)
        for row in reader:
            try:
                n = int(row["n_items"])
```

```
                t = float(row["elapsed_seconds"])
            except (KeyError, ValueError):
                # Skip malformed rows
                continue
            times_by_n[n].append(t)

    n_values = sorted(times_by_n.keys())
    avg_times = [sum(times_by_n[n]) / len(times_by_n[n])
       for n in n_values]

    return n_values, avg_times


def fit_quadratic(n_values, avg_times):
    """
    Fit a curve of the form T(n) ~= a*n^2 + b using least
        squares.

    Returns:
        a, b, fitted_values
    """
    n = np.array(n_values, dtype=float)
    t = np.array(avg_times, dtype=float)

    # We model t ~= a * n^2 + b
    x = n**2
    a, b = np.polyfit(x, t, 1)

    fitted = a * x + b
    return a, b, fitted


def ideal_n2_curve(n_values, avg_times):
    """
    Construct an "ideal" O(n^2) curve k * n^2, scaled so
       that it
    matches the average time at the smallest n.
    """
    n = np.array(n_values, dtype=float)
    t = np.array(avg_times, dtype=float)

    # Anchor k so that k * n0^2 = t0 at the smallest n.
    n0 = n[0]
    t0 = t[0]
    k = t0 / (n0**2)
```

```python

    ideal = k * n**2
    return ideal


def plot_results(n_values, avg_times, fit_times, ideal_times, fig_path: Path):
    """
    Plot measured data, fitted quadratic, and ideal O(n^2) curve.
    """
    n = np.array(n_values, dtype=float)
    t = np.array(avg_times, dtype=float)

    fig_path.parent.mkdir(parents=True, exist_ok=True)

    plt.figure()
    # Measured data
    plt.plot(n, t, "o", label="Measured avg time")

    # Fitted quadratic
    plt.plot(n, fit_times, "-", label="Fitted a*n^2 + b")

    # Ideal O(n^2)
    plt.plot(n, ideal_times, "--", label="Ideal k*n^2")

    plt.xlabel("Number of items (n)")
    plt.ylabel("Time (seconds)")
    plt.title("Bubble Sort: Timing vs. Input Size")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()

    plt.savefig(fig_path, dpi=300)
    plt.close()

    print(f"Saved figure to: {fig_path}")


def main():
    base_dir = Path(__file__).resolve().parent
    data_file = base_dir.parent / "data" / "bubble_sort_timing.csv"
    fig_file = base_dir.parent / "figures" / "bubble_sort_timing_n2.png"
```

```python

    if not data_file.exists():
        raise FileNotFoundError(f"Could not find timing data at {data_file}")

    print(f"Loading timing data from: {data_file}")
    n_values, avg_times = load_timing_data(data_file)

    if not n_values:
        raise RuntimeError("No valid timing data found in CSV.")

    print("Fitting quadratic model T(n) ~= a*n^2 + b ...")
    a, b, fit_times = fit_quadratic(n_values, avg_times)
    ideal_times = ideal_n2_curve(n_values, avg_times)

    print(f"Fit parameters: a = {a:.6e}, b = {b:.6e}")
    print("Generating plot...")
    plot_results(n_values, avg_times, fit_times, ideal_times, fig_file)

    print("Done. This figure is ready to drop into Chapter 2.")


if __name__ == "__main__":
    main()
```

When you run this script, it reads `data/bubble_sort_timing.csv` and produces a figure file named `figures/bubble_sort_timing_n2.png`. That file is a snapshot of the current state of your experiment: whatever timing data you have collected so far is what gets plotted.

2.5 The Plot: Data vs. Big-O

Figure 2.1 shows the result of running `bubble_sort_timing.py` for a range of input sizes and then plotting the average times using `bubble_sort_analyze.py`.

The dots represent the measured average time for each input size n. The solid line is the fitted curve $an^2 + b$, and the dashed line is the ideal curve kn^2 scaled to match the smallest data point.

What we see:

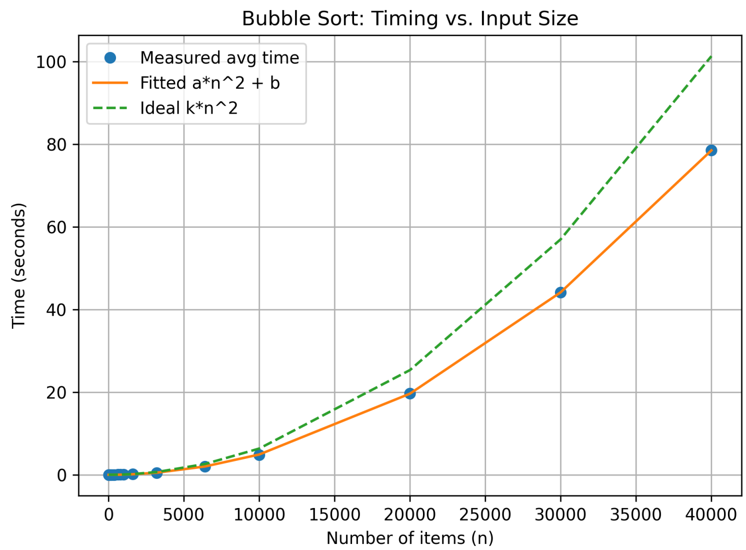

Figure 2.1: Measured bubble sort times vs. input size, along with a fitted quadratic curve and an ideal $\mathcal{O}(n^2)$ curve.

- The data points hug an n^2-shaped curve very closely once n is moderately large.
- Doubling n tends to multiply the running time by a factor close to four, especially for larger lists where timing noise is smaller.
- The exact constants a, b, and k depend on your machine, your Python version, and how busy your computer is, but the *shape* of the curve is consistently quadratic.

This is the heart of Big-O analysis: we ignore the messy, system-dependent details and focus on how the running time scales as n grows. Bubble sort is simple enough that we can both *prove* the $\mathcal{O}(n^2)$ behavior on paper and *see* it in real timing data.

2.6 Looking Ahead

In this chapter we:

- turned bubble sort into an experiment by timing it on random lists of increasing size,

- stored those results in a CSV file and analyzed them with a short Python script, and
- saw that the timing data follows an $\mathcal{O}(n^2)$ curve very closely.

In the next chapters we will:

- compare bubble sort with faster sorting algorithms such as merge sort and quicksort,
- visualize how their timing curves differ, and
- deepen our understanding of Big-O notation by looking at other growth rates like $\mathcal{O}(n \log n)$ and $\mathcal{O}(n)$.

By the time we are done, you will be able to look at a timing plot and say, with some confidence, "that algorithm is behaving like n^2" or "that one looks closer to $n \log n$." And you will know how to build the experiments to justify your claim.