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
```

```
16
17
18 def bubble_sort_with_trace(data):
19     """
20     Perform bubble sort and record the list state after
           each pass.
21
22     Returns:
23         sorted_list: the sorted copy of the input list
24         trace: a list of (pass_number, state_list)
               snapshots
25
26     pass_number = 0 is the initial state before any
           passes.
27     """
28     arr = data[:]  # copy so we don't mutate the original
           list
29     n = len(arr)
30     trace = []
31
32     # Record the initial (unsorted) state
33     trace.append((0, arr[:]))
34
35     for i in range(n):
36         swapped = False
37
38         # One "bubble pass"
39         for j in range(0, n - i - 1):
40             if arr[j] > arr[j + 1]:
41                 # Swap out-of-order neighbors
42                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
43                 swapped = True
44
45         # Record the state of the list *after* this pass
46         trace.append((i + 1, arr[:]))
47
48         # If no swaps were made, the list is already
               sorted
49         if not swapped:
50             break
51
52     return arr, trace
53
54
55 def print_trace(trace):
```

```
56      """
57      Print a friendly view of how the list changes after
            each pass.
58      """
59      print("Bubble sort trace (state after each full pass)
            :")
60      for pass_number, state in trace:
61          if pass_number == 0:
62              label = "Start"
63          else:
64              label = f"Pass {pass_number}"
65          print(f"{label:>6}: {state}")
66
67
68  def append_trace_to_csv(trace, csv_path: Path):
69      """
70      Append the bubble sort trace to a CSV file.
71
72      Columns:
73          timestamp, pass_number, state_list
74
75      - timestamp: when this run was recorded
76      - pass_number: 0 for initial state, 1, 2, ... for
            later passes
77      - state_list: space-separated string version of the
            list
78      """
79      csv_path.parent.mkdir(parents=True, exist_ok=True)
80
81      file_exists = csv_path.exists()
82      run_timestamp = datetime.now().isoformat(timespec="
            seconds")
83
84      with csv_path.open("a", newline="") as f:
85          writer = csv.writer(f)
86
87          # Write header only if this is a new file
88          if not file_exists:
89              writer.writerow(["timestamp", "pass_number",
                    "state_list"])
90
91          for pass_number, state in trace:
92              state_str = " ".join(str(x) for x in state)
93              writer.writerow([run_timestamp, pass_number,
                    state_str])
```

```
94
95
96  if __name__ == "__main__":
97      # Example data; later chapters can experiment with
            other lists.
98      numbers = [9, 3, 7, 2, 10]
99
100     print("Original list:", numbers)
101     sorted_numbers, trace = bubble_sort_with_trace(
            numbers)
102
103     print()
104     print_trace(trace)
105
106     print()
107     print("Sorted list:  ", sorted_numbers)
108
109     # Write CSV file into ../data relative to this script
110     base_dir = Path(__file__).resolve().parent
111     csv_file = base_dir.parent / "data" / "
            bubble_sort_basic_trace.csv"
112
113     append_trace_to_csv(trace, csv_file)
114     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`.

```
1  pass_number,state_list
2  0,9 3 7 2 10
3  1,3 7 2 9 10
4  2,3 2 7 9 10
5  3,2 3 7 9 10
6  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.