

Sorting and Searching

A Gentle Introduction

Jeremy Evert

November 10, 2025

Contents

1	Introduction	1
1.1	Why Sorting and Searching Matter	1
1.2	A First Search: Linear Search	2
1.3	Sorting to Help Searching	3
1.4	Our First Sort: Bubble Sort	3
1.5	Quoting the Data: Bubble Sort Trace CSV	7
1.6	Where We Are Going Next	8
2	Bubble Sort in the Wild: Timing and Big-O	11
2.1	From Algorithm to Experiment	11
2.2	Timing Bubble Sort	12
2.3	Big-O Intuition for Bubble Sort	15
2.4	Fitting an n^2 Curve	15
2.5	The Plot: Data vs. Big-O	19
2.6	Looking Ahead	20
3	Merge Sort: Faster Sorting, Same Experiment	23
3.1	A Recursive Strategy for Sorting	23
3.2	Merge Sort With a Recursive Trace	24
3.3	Timing Merge Sort	28
3.4	Big-O Intuition for Merge Sort	32
3.5	Fitting an $n \log n$ Curve	33
3.6	The Plot: Merge Sort vs. $n \log n$	37
3.7	Bubble Sort vs. Merge Sort	38
3.8	Looking Ahead	39

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.

```
1 def linear_search(data, target):  
2     """  
3     Return the index of 'target' in the list 'data',  
4     or -1 if the target is not found.  
5     """  
6     for index, value in enumerate(data):  
7         if value == target:  
8             return index  
9     return -1  
10  
11  
12 if __name__ == "__main__":  
13     numbers = [9, 3, 7, 2, 10]  
14     target = 7  
15
```

```
16     position = linear_search(numbers, target)
17     if position != -1:
18         print(f"Found {target} at index {position}.")
19     else:
20         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`.

```
1  #!/usr/bin/env python3
2  """
3  bubble_sort_basic.py
4
5  Bubble sort with a simple trace:
6
7  - Prints the list after each full bubble pass so you can
8    see the numbers "bubbling" toward the end.
9  - Appends a CSV row for each pass, including a timestamp,
10    so you can track when the data was generated.
11  """
12
13  import csv
14  from pathlib import Path
15  from datetime import datetime
```



```
16
17
18 def bubble_sort_with_trace(data):
19     """
20     Perform bubble sort and record the list state after
21     each pass.
22
23     Returns:
24         sorted_list: the sorted copy of the input list
25         trace: a list of (pass_number, state_list)
26             snapshots
27
28     pass_number = 0 is the initial state before any
29     passes.
30     """
31     # Work on a copy so we don't mutate the original list
32     working_list = data[:]
33     list_length = len(working_list)
34
35     trace = []
36
37     # Record the initial (unsorted) state
38     trace.append((0, working_list[:]))
39
40     # Each loop over pass_index is one full "bubble pass"
41     for pass_index in range(list_length):
42         did_swap_on_this_pass = False
43
44         # Walk through the list up to the last unsorted
45         # position
46         for compare_index in range(0, list_length -
47             pass_index - 1):
48             left_value = working_list[compare_index]
49             right_value = working_list[compare_index + 1]
50
51             # If neighbors are out of order, swap them
52             if left_value > right_value:
53                 working_list[compare_index], working_list
54                 [compare_index + 1] = (
55                     right_value,
56                     left_value,
57                 )
58                 did_swap_on_this_pass = True
59
60         # Record the state of the list *after* this pass
```

```

55         pass_number = pass_index + 1
56         trace.append((pass_number, working_list[:]))
57
58         # If no swaps were made, the list is already
59         sorted
60         if not did_swap_on_this_pass:
61             break
62
63     return working_list, trace
64
65 def print_trace(trace):
66     """
67     Print a friendly view of how the list changes after
68     each pass.
69     """
70     print("Bubble sort trace (state after each full pass)
71           :")
72     for pass_number, state in trace:
73         if pass_number == 0:
74             label = "Start"
75         else:
76             label = f"Pass {pass_number}"
77         print(f"{label:>6}: {state}")
78
79 def append_trace_to_csv(trace, csv_path: Path):
80     """
81     Append the bubble sort trace to a CSV file.
82
83     Columns:
84         timestamp, pass_number, state_list
85
86     - timestamp: when this run was recorded
87     - pass_number: 0 for initial state, 1, 2, ... for
88       later passes
89     - state_list: space-separated string version of the
90       list
91     """
92     csv_path.parent.mkdir(parents=True, exist_ok=True)
93
94     file_already_exists = csv_path.exists()
95     run_timestamp = datetime.now().isoformat(timespec="
96         seconds")

```

```

94     with csv_path.open("a", newline="") as f:
95         writer = csv.writer(f)
96
97         # Write header only if this is a new file
98         if not file_already_exists:
99             writer.writerow(["timestamp", "pass_number",
100                             "state_list"])
101
102         for pass_number, state in trace:
103             state_str = " ".join(str(x) for x in state)
104             writer.writerow([run_timestamp, pass_number,
105                             state_str])
106
107 if __name__ == "__main__":
108     # Example data; later chapters can experiment with
109     # other lists.
110     numbers = [9, 3, 7, 2, 10]
111
112     print("Original list:", numbers)
113     sorted_numbers, trace = bubble_sort_with_trace(
114         numbers)
115
116     print()
117     print_trace(trace)
118
119     print()
120     print("Sorted list: ", sorted_numbers)
121
122     # Write CSV file into ../data relative to this script
123     base_dir = Path(__file__).resolve().parent
124     csv_file = base_dir.parent / "data" / "
        bubble_sort_basic_trace.csv"
125
126     append_trace_to_csv(trace, csv_file)
127     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
7 2025-11-10T18:17:12,0,9 3 7 2 10
8 2025-11-10T18:17:12,1,3 7 2 9 10
9 2025-11-10T18:17:12,2,3 2 7 9 10
10 2025-11-10T18:17:12,3,2 3 7 9 10
11 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.

```

1  #!/usr/bin/env python3
2  """
3  bubble_sort_timing.py
4
5  Measure wall-clock time for bubble sort on lists of
6  increasing size
7  and append the results to a CSV file.
8
9  Each row in the CSV has:
10 timestamp, n_items, elapsed_seconds
11 """
12
13 import csv
14 import random
15 import time
16 from datetime import datetime
17 from pathlib import Path
18
19 def bubble_sort(arr):
20     """
21     In-place bubble sort with early exit if the list is
22     already sorted.
23     No tracing, just sorting as fast as this simple
24     algorithm allows.
25     """
26     n = len(arr)
27     for i in range(n):
28         swapped = False
29
30         # After each pass, the largest element among the
31         # unsorted part
32         # "bubbles" to the end of the list.
33         for j in range(0, n - i - 1):
34             if arr[j] > arr[j + 1]:
35                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
36                 swapped = True

```



```

35         # If we made no swaps, the list is already sorted
36         if not swapped:
37             break
38
39
40 def time_bubble_sort(n, max_value=10**6):
41     """
42     Generate a random list of length n, sort it with
43     bubble_sort,
44     and return the elapsed wall-clock time in seconds.
45     """
46     data = [random.randint(0, max_value) for _ in range(n)]
47
48     start = time.perf_counter()
49     bubble_sort(data)
50     end = time.perf_counter()
51
52     return end - start
53
54 def append_result(csv_path: Path, n: int, elapsed: float)
55 :
56     """
57     Append a single timing result to the CSV file.
58
59     Columns:
60         timestamp, n_items, elapsed_seconds
61     """
62     csv_path.parent.mkdir(parents=True, exist_ok=True)
63     file_exists = csv_path.exists()
64     timestamp = datetime.now().isoformat(timespec="
65         seconds")
66
67     with csv_path.open("a", newline="") as f:
68         writer = csv.writer(f)
69
70         # Write a header row only if the file is new.
71         if not file_exists:
72             writer.writerow(["timestamp", "n_items", "
73                 elapsed_seconds"])

```

```

74
75 def main():
76     # Figure out where we are and where the data
       directory lives.
77     base_dir = Path(__file__).resolve().parent
78     csv_file = base_dir.parent / "data" / "
       bubble_sort_timing.csv"
79
80     # List sizes to test. Feel free to tweak this for
       bigger/smaller runs.
81     sizes = [100, 200, 400, 800, 1600, 3200, 6400, 10000,
       20000, 30000, 40000]
82     trials_per_size = 3 # Run multiple trials per size
       for smoother data.
83
84     print("Bubble sort timing experiment")
85     print(f"Results will be appended to: {csv_file}")
86     print()
87
88     for n in sizes:
89         for trial in range(1, trials_per_size + 1):
90             elapsed = time_bubble_sort(n)
91             append_result(csv_file, n, elapsed)
92             print(f"n = {n:5d}, trial = {trial}, time = {
       elapsed:.6f} seconds")
93
94     print("\nDone. Data appended to CSV; ready for
       plotting in Chapter 2 and beyond.")
95
96
97 if __name__ == "__main__":
98     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:

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

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” $O(n^2)$ 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.

```

1  #!/usr/bin/env python3
2  """
3  bubble_sort_analyze.py
4
5  Read bubble_sort_timing.csv, compute average time for
6  each n,
7  fit a quadratic curve, and compare it to an ideal  $O(n^2)$ 
8  curve.
9
10 Outputs:
11 figures/bubble_sort_timing_n2.png
12 """
13
14 import csv
15 from collections import defaultdict
16 from pathlib import Path
17
18 import numpy as np
19 import matplotlib.pyplot as plt
20
21 def load_timing_data(csv_path: Path):
22     """
23     Load timing data from CSV and group times by n_items.
24     Returns:
25         n_values (sorted list of ints)
26         avg_times (list of floats, same order as n_values
27                     )
28     """
29     times_by_n = defaultdict(list)
30
31     with csv_path.open("r", newline="") as f:
32         reader = csv.DictReader(f)
33         for row in reader:
34             try:
35                 n = int(row["n_items"])

```

```

35         t = float(row["elapsed_seconds"])
36     except (KeyError, ValueError):
37         # Skip malformed rows
38         continue
39     times_by_n[n].append(t)
40
41     n_values = sorted(times_by_n.keys())
42     avg_times = [sum(times_by_n[n]) / len(times_by_n[n])
43                  for n in n_values]
44
45     return n_values, avg_times
46
47 def fit_quadratic(n_values, avg_times):
48     """
49     Fit a curve of the form  $T(n) \sim a \cdot n^2 + b$  using least
50     squares.
51
52     Returns:
53         a, b, fitted_values
54     """
55     n = np.array(n_values, dtype=float)
56     t = np.array(avg_times, dtype=float)
57
58     # We model  $t \sim a \cdot n^2 + b$ 
59     x = n**2
60     a, b = np.polyfit(x, t, 1)
61
62     fitted = a * x + b
63     return a, b, fitted
64
65 def ideal_n2_curve(n_values, avg_times):
66     """
67     Construct an "ideal"  $O(n^2)$  curve  $k \cdot n^2$ , scaled so
68     that it
69     matches the average time at the smallest n.
70     """
71     n = np.array(n_values, dtype=float)
72     t = np.array(avg_times, dtype=float)
73
74     # Anchor k so that  $k \cdot n_0^2 = t_0$  at the smallest n.
75     n0 = n[0]
76     t0 = t[0]
77     k = t0 / (n0**2)

```

```

77     ideal = k * n**2
78     return ideal
79
80
81
82 def plot_results(n_values, avg_times, fit_times,
83                 ideal_times, fig_path: Path):
84     """
85     Plot measured data, fitted quadratic, and ideal  $O(n^2)$  curve.
86     """
87     n = np.array(n_values, dtype=float)
88     t = np.array(avg_times, dtype=float)
89
90     fig_path.parent.mkdir(parents=True, exist_ok=True)
91
92     plt.figure()
93     # Measured data
94     plt.plot(n, t, "o", label="Measured avg time")
95
96     # Fitted quadratic
97     plt.plot(n, fit_times, "--", label="Fitted  $a*n^2 + b$ ")
98
99     # Ideal  $O(n^2)$ 
100     plt.plot(n, ideal_times, "--", label="Ideal  $k*n^2$ ")
101
102     plt.xlabel("Number of items (n)")
103     plt.ylabel("Time (seconds)")
104     plt.title("Bubble Sort: Timing vs. Input Size")
105     plt.legend()
106     plt.grid(True)
107     plt.tight_layout()
108
109     plt.savefig(fig_path, dpi=300)
110     plt.close()
111
112     print(f"Saved figure to: {fig_path}")
113
114 def main():
115     base_dir = Path(__file__).resolve().parent
116     data_file = base_dir.parent / "data" / "
117         bubble_sort_timing.csv"
118     fig_file = base_dir.parent / "figures" / "
119         bubble_sort_timing_n2.png"

```

```

118
119     if not data_file.exists():
120         raise FileNotFoundError(f"Could not find timing
121                                data at {data_file}")
122
123     print(f"Loading timing data from: {data_file}")
124     n_values, avg_times = load_timing_data(data_file)
125
126     if not n_values:
127         raise RuntimeError("No valid timing data found in
128                            CSV.")
129
130     print("Fitting quadratic model  $T(n) \sim a \cdot n^2 + b \dots$ "
131          )
132     a, b, fit_times = fit_quadratic(n_values, avg_times)
133     ideal_times = ideal_n2_curve(n_values, avg_times)
134
135     print(f"Fit parameters: a = {a:.6e}, b = {b:.6e}")
136     print("Generating plot...")
137     plot_results(n_values, avg_times, fit_times,
138                 ideal_times, fig_file)
139
140     print("Done. This figure is ready to drop into
141          Chapter 2.")
142
143 if __name__ == "__main__":
144     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:

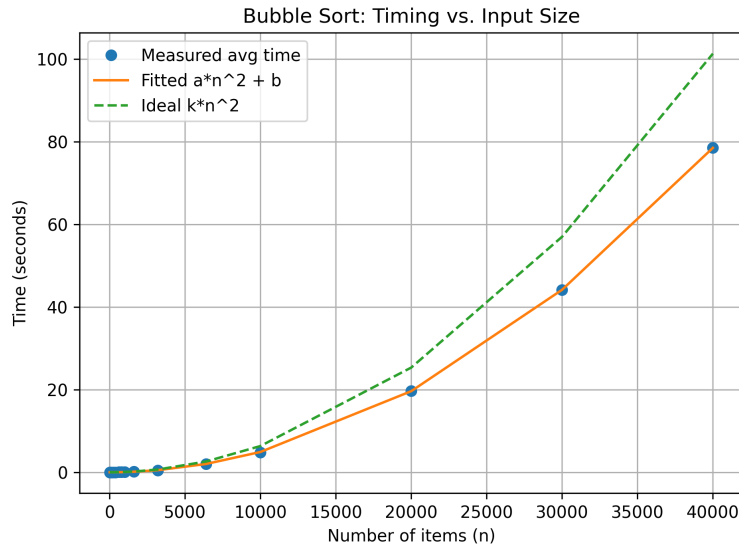


Figure 2.1: Measured bubble sort times vs. input size, along with a fitted quadratic curve and an ideal $\mathcal{O}(n^2)$ curve.

- 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.

Chapter 3

Merge Sort: Faster Sorting, Same Experiment

In the first two chapters we met linear search and bubble sort, and we saw how bubble sort's running time grows roughly like n^2 when we time it on larger and larger inputs.¹ Now it is time to bring in a faster sorting algorithm: *merge sort*. Unlike bubble sort, merge sort is both *recursive* and *asymptotically faster*: its running time grows on the order of $n \log n$ instead of n^2 .

Our plan in this chapter mirrors what we did for bubble sort:

1. understand the algorithm at a high level,
2. study a small, traceable implementation on a toy list, and
3. time the algorithm on large random lists, then fit a curve to the data and compare it to the $\mathcal{O}(n \log n)$ story.

3.1 A Recursive Strategy for Sorting

Bubble sort works locally: it looks at neighboring pairs and swaps them until the list is in order. Merge sort takes a very different approach:

1. If the list has length 0 or 1, it is already sorted.
2. Otherwise, split the list into two halves: a left half and a right half.

¹See Chapters 1 and 2 for the full code and plots connecting bubble sort to its $\mathcal{O}(n^2)$ behavior.

3. Recursively sort the left half.
4. Recursively sort the right half.
5. Merge the two sorted halves into a single sorted list.

The key idea is that it is very easy to merge two *already sorted* lists: you repeatedly pick the smaller of the two front elements and append it to a new list. The recursion does the heavy lifting of breaking the big problem (sorting n items) into smaller subproblems.

A high-level picture:

$$[9, 3, 7, 2, 10] \longrightarrow [9, 3] \text{ and } [7, 2, 10]$$

Then:

$$[9, 3] \longrightarrow [9] \text{ and } [3] \longrightarrow [3, 9],$$

$$[7, 2, 10] \longrightarrow [7] \text{ and } [2, 10] \longrightarrow [2] \text{ and } [10] \longrightarrow [2, 10],$$

and finally we merge $[3, 9]$ and $[2, 7, 10]$ into a fully sorted list $[2, 3, 7, 9, 10]$.

Just as with bubble sort, we want to connect this description to real code and real data.

3.2 Merge Sort With a Recursive Trace

To make the recursion visible, we wrote a small script `scripts/merge_sort_basic.py`. It sorts the list $[9, 3, 7, 2, 10]$ using merge sort and records a trace of the recursive calls and merges.

Listing 3.1 shows the code.

Listing 3.1: Merge sort with a recursive trace, stored in `scripts/merge_sort_basic.py`.

```

1  #!/usr/bin/env python3
2  """
3  merge_sort_basic.py
4
5  A recursive merge sort implementation with a simple trace
6  :
7  - Uses recursion to split the list into halves and then
   merge them.

```

```

8 - Records the subarray at each recursive call and after
   each merge.
9 - Prints a step-by-step trace so you can see the
   recursion unfold.
10 - Appends a CSV row for each step, including a timestamp,
11     so you can replay the recursion later or quote it in
        the book.
12 """
13
14 import csv
15 from datetime import datetime
16 from pathlib import Path
17
18
19 def merge_sort_with_trace(data):
20     """
21     Perform merge sort and record a trace of the
22         recursion.
23
24     Returns:
25         sorted_list: the sorted copy of the input list
26         trace: a list of (step_number, depth, phase,
27             subarray)
28
29     Fields:
30         - step_number: 1, 2, 3, ...
31         - depth: recursion depth (0 for the outermost
32             call)
33         - phase: "call" when a subarray is first seen,
34             "merged" when that subarray has been
35             fully merged
36         - subarray: a snapshot copy of the current
37             subarray at this step
38     """
39     trace = []
40     step = 0
41
42     def helper(arr, depth):
43         nonlocal step
44
45         # Record that we are "calling" merge sort on this
46             subarray
47         step += 1
48         trace.append((step, depth, "call", arr[:]))

```

```

44         if len(arr) <= 1:
45             # Already sorted; nothing to merge
46             return arr[:]
47
48         mid = len(arr) // 2
49         left = helper(arr[:mid], depth + 1)
50         right = helper(arr[mid:], depth + 1)
51
52         # Merge two sorted halves
53         merged = []
54         i = j = 0
55
56         while i < len(left) and j < len(right):
57             if left[i] <= right[j]:
58                 merged.append(left[i])
59                 i += 1
60             else:
61                 merged.append(right[j])
62                 j += 1
63
64         # Append any leftovers
65         merged.extend(left[i:])
66         merged.extend(right[j:])
67
68         # Record the merged result at this depth
69         step += 1
70         trace.append((step, depth, "merged", merged[:]))
71
72         return merged
73
74     sorted_list = helper(data, depth=0)
75     return sorted_list, trace
76
77
78 def print_trace(trace):
79     """
80     Print a friendly view of the merge sort recursion.
81     """
82     print("Merge sort recursive trace:")
83     for step_number, depth, phase, subarray in trace:
84         print(
85             f"step={step_number:2d}, depth={depth:2d}, "
86             f"phase={phase:7s}, subarray={subarray}"
87         )
88

```

```

89
90 def append_trace_to_csv(trace, csv_path: Path):
91     """
92     Append the merge sort trace to a CSV file.
93
94     Columns:
95         timestamp, step_number, depth, phase, subarray
96
97     - timestamp: when this run was recorded
98     - step_number: 1, 2, 3, ...
99     - depth: recursion depth at this step
100    - phase: "call" or "merged"
101    - subarray: space-separated string version of the
      subarray
102    """
103    csv_path.parent.mkdir(parents=True, exist_ok=True)
104
105    file_exists = csv_path.exists()
106    run_timestamp = datetime.now().isoformat(timespec="
      seconds")
107
108    with csv_path.open("a", newline="") as f:
109        writer = csv.writer(f)
110
111        # Write header only if this is a new file
112        if not file_exists:
113            writer.writerow(
114                ["timestamp", "step_number", "depth", "
      phase", "subarray"]
115            )
116
117        for step_number, depth, phase, subarray in trace:
118            sub_str = " ".join(str(x) for x in subarray)
119            writer.writerow(
120                [run_timestamp, step_number, depth, phase
      , sub_str]
121            )
122
123
124 if __name__ == "__main__":
125     # Small data set so the trace is easy to follow.
126     numbers = [9, 3, 7, 2, 10]
127
128     print("Original list:", numbers)
129     sorted_numbers, trace = merge_sort_with_trace(numbers

```

```

130         )
131     print()
132     print_trace(trace)
133
134     print()
135     print("Sorted list: ", sorted_numbers)
136
137     # Write CSV file into ../data relative to this script
138     base_dir = Path(__file__).resolve().parent
139     csv_file = base_dir.parent / "data" / "
        merge_sort_basic_trace.csv"
140
141     append_trace_to_csv(trace, csv_file)
142     print(f"\nMerge sort trace appended to {csv_file}")

```

This script does a few important things for us:

- The function `merge_sort_with_trace` returns both the sorted list and a trace of the recursion.
- Each trace entry records:
 - a step number,
 - the recursion depth,
 - whether we are at a “call” or a “merged” state, and
 - the current subarray.
- The trace is printed in a human-readable form so we can follow how the list is split and then merged back together.
- The same trace is appended to a CSV file `data/merge_sort_basic_trace.csv`, so that we can load it into a spreadsheet or refer to it later in the book.

Conceptually, this is the merge sort analog of the bubble sort trace script from Chapter 1: instead of watching values “bubble” to the end, we watch subarrays split and then merge as the recursion unwinds.

3.3 Timing Merge Sort

Just as we did with bubble sort in Chapter 2, we now turn merge sort into a timing experiment. The idea is the same:

- choose a list size n (for example, 100, 200, 400, ...),
- generate a random list of n integers,
- sort it with merge sort, and
- measure the elapsed time.

We repeat this process for a range of input sizes and several trials per size, then store the results in a CSV file.

Listing 3.2 shows the timing script `scripts/merge_sort_timing.py`.

Listing 3.2: Timing merge sort on growing input sizes.

```
1  #!/usr/bin/env python3
2  """
3  merge_sort_timing.py
4
5  Measure wall-clock time for merge sort on lists of
6  increasing size
7  and append the results to a CSV file.
8
9  Each row in the CSV has:
10 timestamp, n_items, elapsed_seconds
11 """
12
13 import csv
14 import random
15 import time
16 from datetime import datetime
17 from pathlib import Path
18
19 def merge_sort(arr):
20     """
21 Recursive merge sort that returns a new sorted list.
22
23 This version does no tracing; it is written to be as
24 straightforward and efficient as possible for timing.
25 """
26 n = len(arr)
27 if n <= 1:
28     return arr
29
30 mid = n // 2
31 left = merge_sort(arr[:mid])
```

```

32     right = merge_sort(arr[mid:])
33
34     merged = []
35     i = j = 0
36
37     # Merge the two sorted halves
38     while i < len(left) and j < len(right):
39         if left[i] <= right[j]:
40             merged.append(left[i])
41             i += 1
42         else:
43             merged.append(right[j])
44             j += 1
45
46     # Append leftovers from either half
47     merged.extend(left[i:])
48     merged.extend(right[j:])
49
50     return merged
51
52
53 def time_merge_sort(n, max_value=10**6):
54     """
55     Generate a random list of length n, sort it with
56     merge_sort,
57     and return the elapsed wall-clock time in seconds.
58     """
59     data = [random.randint(0, max_value) for _ in range(n)]
60
61     start = time.perf_counter()
62     _ = merge_sort(data)
63     end = time.perf_counter()
64
65     return end - start
66
67 def append_result(csv_path: Path, n: int, elapsed: float)
68     :
69     """
70     Append a single timing result to the CSV file.
71
72     Columns:
73         timestamp, n_items, elapsed_seconds
74     """

```

```

74     csv_path.parent.mkdir(parents=True, exist_ok=True)
75     file_exists = csv_path.exists()
76     timestamp = datetime.now().isoformat(timespec="
        seconds")
77
78     with csv_path.open("a", newline="") as f:
79         writer = csv.writer(f)
80
81         # Write a header row only if the file is new.
82         if not file_exists:
83             writer.writerow(["timestamp", "n_items", "
                elapsed_seconds"])
84
85         writer.writerow([timestamp, n, f"{elapsed:.6f}"])
86
87
88 def main():
89     # Figure out where we are and where the data
        directory lives.
90     base_dir = Path(__file__).resolve().parent
91     csv_file = base_dir.parent / "data" / "
        merge_sort_timing.csv"
92
93     # Match your bubble_sort_timing sizes so we can
        compare later.
94     sizes = [100, 200, 400, 800, 1600, 3200, 6400, 10000,
        20000, 30000, 40000]
95     trials_per_size = 3 # Run multiple trials per size
        for smoother data.
96
97     print("Merge sort timing experiment")
98     print(f"Results will be appended to: {csv_file}")
99     print()
100
101     for n in sizes:
102         for trial in range(1, trials_per_size + 1):
103             elapsed = time_merge_sort(n)
104             append_result(csv_file, n, elapsed)
105             print(f"n = {n:5d}, trial = {trial}, time = {
                elapsed:.6f} seconds")
106
107     print("\nDone. Data appended to CSV; ready for
        plotting alongside bubble sort.")
108
109

```

```

110 if __name__ == "__main__":
111     main()

```

A few parallels with the bubble sort timing script:

- `merge_sort` is a clean, recursive implementation of merge sort with no extra printing or tracing.
- `time_merge_sort(n)` generates a random list of length n , sorts it, and returns the elapsed wall-clock time.
- `append_result` appends a row to `data/merge_sort_timing.csv` with the timestamp, the input size, and the elapsed time in seconds.
- The `main()` function loops over the same list sizes used for bubble sort (100, 200, 400, ..., 40 000) and runs several trials per size.

At this point, we have two timing engines:

- `bubble_sort_timing.py` producing `data/bubble_sort_timing.csv`, and
- `merge_sort_timing.py` producing `data/merge_sort_timing.csv`.

The next step is to analyze the merge sort timing data and connect it to the theoretical $\mathcal{O}(n \log n)$ behavior.

3.4 Big-O Intuition for Merge Sort

Recall that bubble sort's running time grows like n^2 because it performs on the order of n^2 neighbor comparisons. Merge sort behaves very differently.

At each level of recursion:

- we split the list into two halves, and
- we merge those halves back together.

If you imagine the recursion as a tree:

- the root represents the original problem of size n ,
- the next level has two subproblems of size roughly $n/2$,
- the next has four subproblems of size roughly $n/4$, and so on.

The height of this recursion tree is about $\log_2 n$, because each level cuts the size of the subproblems in half.

At each level, the total amount of work done by all merges combined is proportional to n :

- at the top level we merge one list of size n ,
- at the next level we merge two lists of size about $n/2$ each (still about n total elements),
- at the next level we merge four lists of size about $n/4$ each (again about n total), and so on.

So we have about $\log_2 n$ levels, each costing about cn work for some constant c . The total work is therefore on the order of

$$T(n) \approx cn \log_2 n,$$

which is what we summarize by saying:

$$T(n) \in \mathcal{O}(n \log n).$$

The timing script gives us a way to check this story against reality.

3.5 Fitting an $n \log n$ Curve

To make the connection quantitative, we wrote an analysis script `scripts/merge_sort_analyze.py`. It reads `data/merge_sort_timing.csv`, groups the data by input size, and computes the average time for each n . Then it fits a curve of the form

$$T(n) \approx a n \log_2 n + b$$

to the data and also builds an “ideal” $\mathcal{O}(n \log n)$ curve $k n \log_2 n$ scaled to match the smallest data point.

Listing 3.3 shows the analysis script.

Listing 3.3: Analyzing and plotting merge sort timing data.

```

1 #!/usr/bin/env python3
2 """
3 merge_sort_analyze.py
4 
5 Read merge_sort_timing.csv, compute average time for each
   n,

```

```

6 | fit an  $n \cdot \log(n)$  curve, and compare it to an ideal  $O(n \log n)$  curve.
7 |
8 | Outputs:
9 |     figures/merge_sort_timing_nlogn.png
10 | """
11 |
12 | import csv
13 | from collections import defaultdict
14 | from pathlib import Path
15 |
16 | import numpy as np
17 | import matplotlib.pyplot as plt
18 |
19 |
20 | def load_timing_data(csv_path: Path):
21 |     """
22 |     Load timing data from CSV and group times by n_items.
23 |
24 |     Returns:
25 |         n_values (sorted list of ints)
26 |         avg_times (list of floats, same order as n_values)
27 |     """
28 |     times_by_n = defaultdict(list)
29 |
30 |     with csv_path.open("r", newline="") as f:
31 |         reader = csv.DictReader(f)
32 |         for row in reader:
33 |             try:
34 |                 n = int(row["n_items"])
35 |                 t = float(row["elapsed_seconds"])
36 |             except (KeyError, ValueError):
37 |                 # Skip malformed rows
38 |                 continue
39 |             times_by_n[n].append(t)
40 |
41 |     n_values = sorted(times_by_n.keys())
42 |     avg_times = [sum(times_by_n[n]) / len(times_by_n[n])
43 |                  for n in n_values]
44 |
45 |     return n_values, avg_times
46 |
47 | def fit_nlogn(n_values, avg_times):

```

```

48     """
49     Fit a curve of the form  $T(n) \approx a * n * \log_2(n) + b$ 
        using least squares.
50
51     Returns:
52         a, b, fitted_values
53     """
54     n = np.array(n_values, dtype=float)
55     t = np.array(avg_times, dtype=float)
56
57     # We model  $t \approx a * (n \log n) + b$ 
58     x = n * np.log2(n)
59     a, b = np.polyfit(x, t, 1)
60
61     fitted = a * x + b
62     return a, b, fitted
63
64
65 def ideal_nlogn_curve(n_values, avg_times):
66     """
67     Construct an "ideal"  $O(n \log n)$  curve  $k * n * \log_2(n)$ ,
68     scaled so that it matches the average time at the
        smallest n.
69     """
70     n = np.array(n_values, dtype=float)
71     t = np.array(avg_times, dtype=float)
72
73     x = n * np.log2(n)
74
75     # Anchor k so that  $k * x_0 = t_0$  at the smallest n.
76     x0 = x[0]
77     t0 = t[0]
78     k = t0 / x0
79
80     ideal = k * x
81     return ideal
82
83
84 def plot_results(n_values, avg_times, fit_times,
        ideal_times, fig_path: Path):
85     """
86     Plot measured data, fitted  $a * n * \log(n) + b$ , and ideal
         $k * n * \log(n)$  curve.
87     """
88     n = np.array(n_values, dtype=float)

```

```

89     t = np.array(avg_times, dtype=float)
90
91     fig_path.parent.mkdir(parents=True, exist_ok=True)
92
93     plt.figure()
94
95     # Measured data
96     plt.plot(n, t, "o", label="Measured avg time")
97
98     # Fitted n log n curve
99     plt.plot(n, fit_times, "-", label="Fitted a*n*log2(n)
100             + b")
101
102     # Ideal O(n log n)
103     plt.plot(n, ideal_times, "--", label="Ideal k*n*log2(
104             n)")
105
106     plt.xlabel("Number of items (n)")
107     plt.ylabel("Time (seconds)")
108     plt.title("Merge Sort: Timing vs. Input Size")
109     plt.legend()
110     plt.grid(True)
111     plt.tight_layout()
112
113     plt.savefig(fig_path, dpi=300)
114     plt.close()
115
116     print(f"Saved figure to: {fig_path}")
117
118 def main():
119     base_dir = Path(__file__).resolve().parent
120     data_file = base_dir.parent / "data" / "
121             merge_sort_timing.csv"
122     fig_file = base_dir.parent / "figures" / "
123             merge_sort_timing_nlogn.png"
124
125     if not data_file.exists():
126         raise FileNotFoundError(f"Could not find timing
127             data at {data_file}")
128
129     print(f>Loading timing data from: {data_file}")
130     n_values, avg_times = load_timing_data(data_file)
131
132     if not n_values:

```



```

129         raise RuntimeError("No valid timing data found in
130                               CSV.")
131
132     print("Fitting model T(n) ~= a*n*log2(n) + b ...")
133     a, b, fit_times = fit_nlogn(n_values, avg_times)
134     ideal_times = ideal_nlogn_curve(n_values, avg_times)
135
136     print(f"Fit parameters: a = {a:.6e}, b = {b:.6e}")
137     print("Generating plot...")
138     plot_results(n_values, avg_times, fit_times,
139                 ideal_times, fig_file)
140
141     print("Done. This figure is ready to drop into the
142           merge sort chapter.")
143
144 if __name__ == "__main__":
145     main()

```

When you run this script, it produces a figure file named `figures/merge_sort_timing_nlogn.png`. Just like in the bubble sort chapter, the figure reflects the timing data you have collected so far: if you rerun `merge_sort_timing.py` to gather more data, then rerun `merge_sort_analyze.py`, the plot will update.

3.6 The Plot: Merge Sort vs. $n \log n$

Figure 3.1 shows the result of timing merge sort across a range of input sizes and then plotting the average times using `merge_sort_analyze.py`.

As with bubble sort, the dots in the figure represent the measured average time for each input size n . The solid line is the fitted curve $an \log_2 n + b$, and the dashed line is the ideal curve $kn \log_2 n$.

What we see:

- The measured times line up very closely with an $n \log n$ shaped curve once n is moderately large.
- Doubling n no longer multiplies the running time by a factor of four (as with n^2); instead, the growth is much gentler.
- As before, the exact constants a , b , and k depend on your machine and environment, but the *shape* of the curve is consistently $n \log n$.

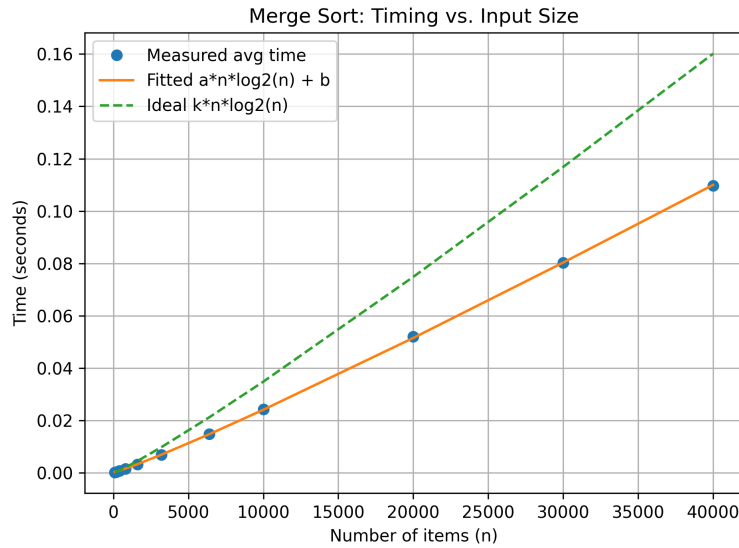


Figure 3.1: Measured merge sort times vs. input size, along with a fitted $an \log_2 n + b$ curve and an ideal $\mathcal{O}(n \log n)$ curve.

3.7 Bubble Sort vs. Merge Sort

We now have timing plots for two sorting algorithms:

- bubble sort, with an $\mathcal{O}(n^2)$ curve (Figure 2.1), and
- merge sort, with an $\mathcal{O}(n \log n)$ curve (Figure 3.1).

On small inputs, the two algorithms may appear to have similar running times. In fact, for very tiny lists, bubble sort can sometimes be competitive because it is simple and has low constant overhead.

However, as n grows:

- the bubble sort curve bends sharply upward, reflecting its n^2 growth, while
- the merge sort curve rises much more slowly, tracking $n \log n$.

If you imagine pushing n to ten times, a hundred times, or a thousand times its current size, the difference becomes dramatic. An algorithm that takes time proportional to n^2 will eventually become painfully slow, while

an $n \log n$ algorithm like merge sort remains practical for much larger data sets.

This is the payoff of Big-O analysis:

- We can reason on paper about how the running time should scale.
- We can then design experiments, collect data, and fit curves to see whether reality matches our theory.
- When the math and the measurements agree, we get strong evidence that we understand the algorithm's behavior.

3.8 Looking Ahead

At this point, we have:

- traced bubble sort and merge sort on small lists to understand how they work,
- timed bubble sort and seen an $\mathcal{O}(n^2)$ curve emerge from real data,
- timed merge sort and seen an $\mathcal{O}(n \log n)$ curve emerge,
- and compared the two curves to build intuition about why asymptotically faster algorithms matter.

In the chapters that follow, we will:

- explore other sorting algorithms (such as quicksort and insertion sort) and place them on the same timing plots,
- connect these sorting ideas back to searching, including binary search on sorted data, and
- practice reading and writing Big-O notation until it feels like a natural language for describing algorithmic behavior.

The more you run these experiments yourself—modifying list sizes, adding more trials, or trying different machines—the stronger your intuition will be. The goal is not just to memorize which algorithms are “fast,” but to understand how and why their running times grow the way they do.