2XC3 — Graded Lab 1

Sorting runtimes & optimizations

Week 1 (Experiments 1–3)

Goal: run clean experiments, make clear plots, and write a report.

You’ll implement + measure — the report is a big part of the grade.

Weeks 1 & 2 structure

Week 1 (this week): “bad” sorts

- Experiment 1: compare Bubble vs Selection vs Insertion
- Experiment 2: implement improved Bubble + improved Selection; compare
- Experiment 3: test “near-sorted” lists using swaps parameter
- Deliverable focus: clean plots + short conclusions per experiment

Week 2 (next week): “good” sorts

- Experiments 4–8: Merge / Quick / Heap comparisons + variations
- Near-sorted behaviour for Quick sort
- Dual-pivot Quick sort + bottom-up Merge sort
- Small-list case: when insertion sort can beat “good” sorts
- Note: strategy-pattern portion postponed (per lab handout)

What you submit

Submission:

- report.pdf (or report.docx) — professional formatting matters
- code.zip — all source code, including experiment scripts

Important:

- Your report can lose up to 20% for messy figures, poor formatting, or missing required elements.
- For each experiment, include: (1) experiment setup, (2) graph(s), (3) a short conclusion.

Report structure

Front Page

- Title page
- Table of Contents
- Table of Figures
- Executive summary (bullet form is fine)

Body + appendix

- A clearly marked section for each experiment
- For each: experiment setup + graph(s) + conclusion
- Appendix: explain how to navigate your code (where each implementation/experiment lives)

Week 1 checklist (Experiments 1–3)

By the end of Week 1, you should (ideally) have:

Experiment outputs

- Week 1: (E1) lengths vs time(Plot rt for 3 algos), (E2) 2 plots bubble vs selection (orig vs improved), (E3) swaps vs time for each algo. Short write-ups: what you changed, what you measured, and what you observed

Experiment 1

- Compare Bubble, Selection, Insertion (given in bad_sorts.py)
- Choose list sizes + number of runs
- One graph: list length vs runtime (3 curves)

Experiments 2–3

- E2: implement bubblesort2 + improved selection (min+max)
- E2: two graphs (og vs improved for each algorithm)
- E3: near-sorted behaviour using swaps parameter (fixed length)

Experiment 1 — baseline comparison (Bubble/Selection/Insertion)

Goal: Perform runtime analysis to see which sorting algorithm is quickest.

What to do

- Use the provided implementations in bad_sorts.py
- Design a “suitable” timing experiment (you choose list sizes and number of runs)
- Compare the three algorithms on the same inputs

What to include in the report

- Explicit experiment setup (sizes, runs, etc.)
- Graph: list length vs time (3 curves)
- Brief discussion + conclusion

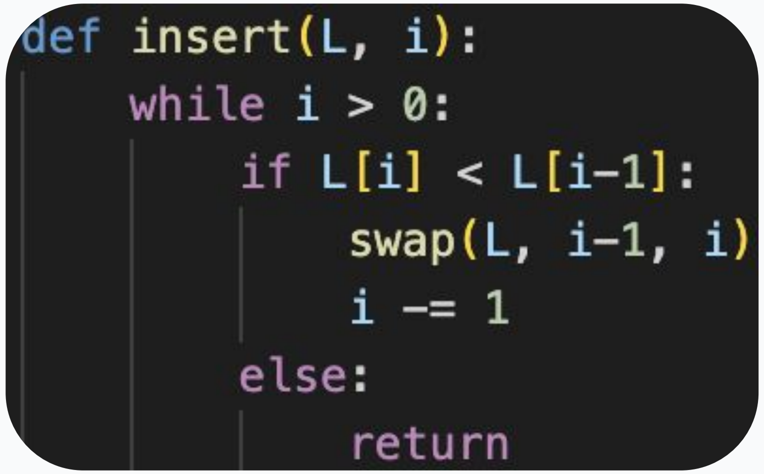
Tip: pick sizes large enough to see separation, but not so large that $O(n^2)$ runs forever.

Insertion sort optimization

swap-based insert vs shift-and-insert

Traditional insert(swap-based)

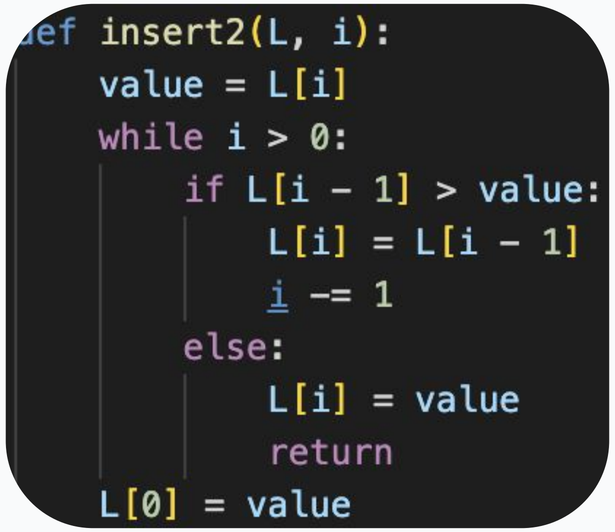
- Moves the new item by repeatedly swapping adjacent elements
- A value can switch multiple positions via many swaps(multiple writes per step)
- One graph: list length vs runtime (3 curves)



```
def insert(L, i):
    while i > 0:
        if L[i] < L[i-1]:
            swap(L, i-1, i)
            i -= 1
        else:
            return
```

Optimized insert(shift-and-insert)

- Stores the item being inserted in a temp variable
- Shift larger elements right until the right position to insert the value is found
- Avoids multiple swap()s and reduces assignments



```
ef insert2(L, i):
    value = L[i]
    while i > 0:
        if L[i - 1] > value:
            L[i] = L[i - 1]
            i -= 1
        else:
            L[i] = value
            return
    L[0] = value
```

Experiment 2 — implement variations & compare

Goal: See if “small” algorithm tweaks create measurable speedups.

BubbleSort2

- Instead of repeatedly swapping values, try to “insert” values into place and shift elements
- This is analogous to the improved insertion-sort concept from lecture
- Name your implementation: bubblesort2()

SelectionSort2

- In each pass, track both the min and max values
- Place the min at the left boundary and the max at the right boundary
- Update loop boundaries accordingly
- Name your implementation: selectionsort2()

Experiment 3 — near-sorted lists (swaps vs runtime)

Goal: Test how sensitive each algorithm is to “almost sorted” inputs.

Setup

- Use create_near_sorted_list(length, max_value, swaps) from bad_sorts.py
- Fix list length (recommended: 5000 or another reasonable constant)
- Vary swaps to smoothly move from “sorted” → “random-like”
- Compare Bubble / Selection / Insertion

Report requirements

- Explicit experiment setup (fixed length, swaps range, runs)
- Graph: swaps vs time (3 rt analysis)
- Choose a swaps range that shows meaningful results
- Brief discussion + conclusion

What a strong “analysis” looks like

In each experiment conclusion, aim to answer:

- Which algorithm/variation is faster in your tested range? Where do curves cross (if they do)?
- Do results match what you expect from Big-O? If not, why might constants/overhead matter?
- For near-sorted inputs: which algorithm benefits most from “more sortedness”? Explain
- Are your results stable across multiple runs? Mention noise and how you reduced it.

Keep it short: 3–6 sentences is usually enough if they are specific.

Grading breakdown

Part 1 (Week 1)

- Experiment 1 — 10%
- Experiment 2 — 20%
- Experiment 3 — 10%

Part 2 (Week 2)

- Experiment 4 — 10%
- Experiment 5 — 10%
- Experiment 6 — 15%
- Experiment 7 — 15%
- Experiment 8 — 10%

Reminder: reports can lose up to 20% if they look unprofessional (formatting + messy graphs).