2XC3 — Graded Lab 1

Sorting runtimes & optimizations

Week 2 (Experiments 4-8)

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

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

Week 2 structure

What you will do (Experiments 4–8):

- Experiment 4: Compare Quick vs Heap vs Merge sorts
- Experiment 5: Test near-sorted inputs: when Quicksort stops being slow.
- Experiment 6: Implement a dual-pivot Quick sort and compare to standard Quick sort.
- Experiment 7: Implement a bottom-up (iterative) Merge sort and compare to recursive Merge sort.
- Experiment 8: Small-list case: when Insertion sort can beat “good” sorts.

What to submit:

- Experiment setup (sizes, runs, fixed parameters).
- Graph(s) required for that experiment.
- A short conclusion (specific and tied to the plot).

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.

Experiment 4 — baseline comparison (Merge/Heap/Quick)

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

What to do

- Use the provided implementations in good_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: Quick sort should stand out in many typical cases — measure to confirm.

Experiment 5 — near-sorted inputs and Quick sort

Goal: Test how sensitive each algorithm(specifically quicksort) is to “almost sorted” inputs.

Setup

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

Report requirements

- Fix list length (constant).
- Vary swaps using create_near_sorted_list(length, max_value, swaps).
- Graph: swaps vs time (Merge / Quick / Heap curves).
- Brief discussion + conclusion.

Quicksort optimization

1 Pivot vs 2 Pivots

Key idea

- Choose 2 pivots, p and q (e.g. first two elements):
  where $p \leq q$
- Partition into 3 parts:
  $< p \mid p \ldots q$ (inclusive) $\mid > q$
- Maintain pointers and move elements to the correct parts as you scan
- Place pivots into final positions after partitioning

Hints

- Elements that are duplicates of the pivots should go in the middle partition.
- Base Case:
  - Sub-lists of size 0 and 1 are already sorted
- After Partition:
  - left < p, middle [p,q], right > q

Experiment 6 — dual-pivot Quick sort

Goal: Test if extra partitioning overhead pays off in runtime

Setup

- Implement dual_quicksort(L)
- Instead of having 1 pivot, have 2 pivots
- 2 subproblems (Left/Right) -> 3 subproblems (Left/Middle/Right)

Report requirements

- Explicit setup (lengths, runs).
- One graph: list length vs time (standard vs dual-pivot).
- Brief discussion + conclusion.

Experiment 7 — bottom-up (iterative) Merge sort

Handle non-power-of-2 lengths in your implementation (lists won't always divide evenly).

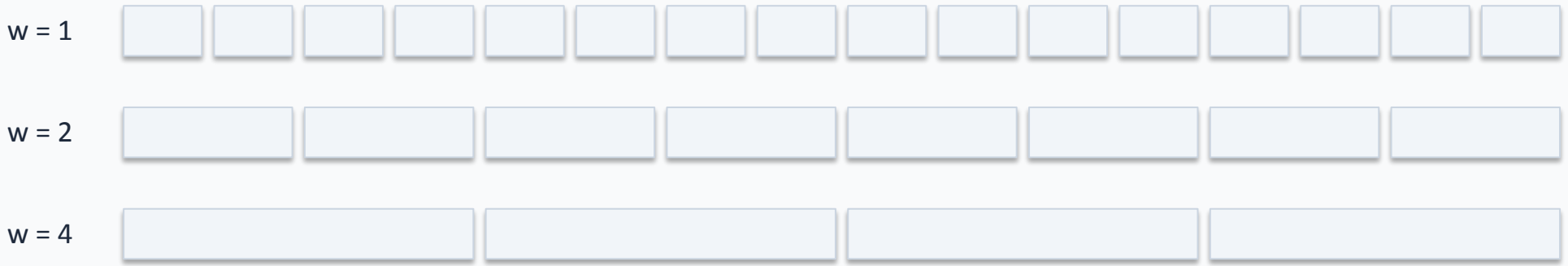
Key Idea

- Instead of top-down recursion (split → merge), start with small "sorted" windows and iteratively merge them.
- Window size increases each pass until the whole list is merged.
- Implement: bottom_up_mergesort(L).

Report requirements

- Explicit setup (lengths, runs).
- One graph: list length vs time (recursive vs bottom-up).
- Brief discussion + conclusion.

Window size doubles each pass



Experiment 8 — small lists: when Insertion sort wins

Goal: Connect to real implementations: Many “fast” sorts switch to insertion sort for small subproblems.

Setup

- Compare Insertion sort vs Merge sort vs Quick sort on small list lengths.
- Identify the crossover: where insertion stops being competitive as n grows.
- Pick a meaningful range of list lengths, so you can observe something worth reporting!

Report requirements

- Explicit setup (small lengths, runs).
- One graph: list length vs time (3 curves).
- Brief discussion + conclusion.
- A few sentences on why this is practical (hybrid strategies).

Analysis & Grading (Part 2)

Keep conclusions specific: refer to shapes of curves, crossovers, and the parameter ranges you tested.

In each conclusion, answer (briefly)

- Which algorithm/variation is faster in your tested range?
- Do results match Big-O expectations? If not, what constant factors could explain it?
- For near-sorted inputs: which algorithm benefits most from “more sortedness”?
- Are results stable across runs (noise)?

Part 2 grade breakdown

- Experiment 4 — 10%
- Experiment 5 — 10%
- Experiment 6 — 15%
- Experiment 7 — 15%
- Experiment 8 — 10%
- Report formatting penalty: up to −20%

Graded Lab 1 checklist (quick recap)

By the end of Week 2, you should have

E1: 1 plot (lengths vs time) comparing Selection / Insertion / Bubble
E2: 2 plots bubble and selection (orig vs improved)
E3: 1 plot (swaps vs time) for each algorithm.
E4: 1 plot (length vs time) comparing Merge / Quick / Heap.
E5: 1 plot (swaps vs time) showing near-sorted sensitivity for the “good” sorts (fixed length).
E6: dual-pivot quicksort implemented + 1 plot (length vs time) vs standard quicksort.
E7: bottom-up mergesort implemented + 1 plot (length vs time) vs recursive mergesort.
E8: small-n comparison including insertion sort + 1 plot (length vs time) + short practical note

Submission:

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