18YZALG — Tutorials 5–6 Assignments

Tutorials 5–6 — Mini-Projects

Choose **one** mini-project per group and present your solution in the tutorial session.

Quick facts

Item	Description
Who	Student groups (recommended 2–3 students)
When	Tutorial presentation session
What	Short live demo (approx. 10–15 minutes per group)
What we practice	Sorting strategy choice, benchmark design, edge cases, correctness, and trade-off analysis
Grading	Credit to the continuous assessment (tutorial part of the course)

What every group must include

- **Problem statement:** input, output, edge cases, and a short explanation of what makes the task interesting.
- **At least three approaches:** one simple baseline, one strong practical method, and one more robust or hybrid alternative. Bubble Sort and Selection Sort may appear only as tiny-input or pedagogical baselines.
- **Complexity:** informal Big-O time and memory discussion for each approach.
- **Benchmark plan:** at least **6 input families** and **5+ input sizes**. Include random data, nearly sorted data, reverse-sorted data, many duplicates, low-entropy keys (for example only a few distinct values), and one adversarial case chosen to hurt a method.
- **Metrics:** runtime is required; add at least **two** more measurements when relevant, for example comparisons, swaps or moves, extra memory, recursion depth, heap size, number of temporary runs, or delay before first output.
- **Correctness checks:** a small test suite with representative edge cases. Your compared methods should agree on the same inputs.
- **Failure analysis:** show one case where a method loses badly or behaves unexpectedly, and explain why.
- **Conclusion:** when would you use which approach, and why?

Submission format

- No files needed to submit beforehand.
- Only deliverable is the **live demonstration**.

- Use slides, a notebook, terminal demo - anything is fine, as long as you cover the required content.

Live demonstration checklist (structure)

Aim for a concise demo. You can split speaking roles inside the group.

Suggested structure that fits *sorting* mini-projects well:

- story and formal problem statement.
- benchmark design: input families, metrics, and fairness rules.
- approaches (baseline, practical method, robust or hybrid method).
- complexity and the reason you expect the methods to differ.
- correctness and edge cases.
- benchmark results and interpretation.
- one failure case, then a final recommendation.

Assessment (20 points)

Criterion	Points	What we look for
Correctness & tests	0–5	Works on edge cases; tests are clear and actually run
Algorithm choice & explanation	0–5	Approaches are sensible; trade-offs are explained honestly
Benchmark design & interpretation	0–5	Input families are varied; timing is fair; conclusions match results
Demo clarity	0–5	Clear structure, readable visuals, good pacing

Mini-Projects

Pick **exactly one**. Send me an email with what your group has picked - it is first come, first served.

1. Adaptive sorting engine: choose the algorithm, do not hard-code it

Goal. Imagine you are writing the sorting component of a small analytics system. On one day it sorts short arrays that are already almost ordered. On another day it sorts long records with many repeated keys. On a third day it must preserve the order of equal-key records because the original order carries meaning. A real engineer would not ask only “which sorting algorithm is best?” but also “what kind of input do I have, and which method fits it?”. Your task is to build a sorter that first inspects the input and then chooses a sorting strategy automatically.

Required content

- Explain what input features your program measures before sorting: for example size, estimated disorder, duplicate ratio, run lengths, stability requirement, or memory budget.
- Fix your decision rules **before** the final benchmark. Do not change the rule by hand from test to test.
- Compare your chooser with at least two fixed baselines such as “always QuickSort” or “always MergeSort”.
- Include at least one case where your chooser makes a bad decision and explain why that happened.
- Make it clear how much time the inspection step itself costs.

Suggested approaches to compare

- Method 1 (baseline): always use one fixed algorithm.
- Method 2 (baseline): always use a different fixed algorithm with a contrasting behavior.
- Method 3: a rule-based adaptive chooser.
- Optional Method 4: a hybrid design, for example QuickSort with an insertion-sort cutoff and a HeapSort fallback.

Demo focus

- Show a few tiny inputs and explain what your chooser “sees” in them.
- Identify one or two break-even points where the chosen algorithm changes.
- Be honest about the failure case: the interesting part is understanding why the rule was fooled.

Stretch goals (optional)

- Tune thresholds automatically from a training set instead of setting them by hand.
- Separate the goal “fastest on average” from the goal “most robust across all cases”.

2. QuickSort under attack: make QuickSort robust on bad inputs

Goal. QuickSort is famous because it is often extremely fast in practice, but poor pivot choices and unfortunate input structure can damage it badly. In this project you will play both attacker and engineer. First, design inputs that hurt a simple QuickSort. Then, modify the algorithm so that it behaves better under pressure. The point is not only to say that one version is faster, but to show *why* pivot quality and partition strategy matter.

Required content

- Implement at least **three pivot rules**, for example first element, random pivot, and median-of-three.
- Compare at least **two partition strategies**, for example standard 2-way partition and 3-way partition for duplicate-heavy data.
- Benchmark on adversarial inputs such as sorted, reverse-sorted, all equal, few distinct values, organ-pipe, sawtooth, and random data.

- Measure runtime and at least two more relevant metrics, such as comparisons, swaps, and maximum recursion depth.
- Explain which inputs are genuinely bad for each variant and why.

Suggested approaches to compare

- Method 1: plain QuickSort with a weak pivot rule.
- Method 2: QuickSort with a better pivot rule.
- Method 3: 3-way QuickSort for duplicate-heavy inputs.
- Optional Method 4: an IntroSort-like version, meaning QuickSort with a HeapSort fallback after too much recursion.

Demo focus

- Show how duplicate-heavy data changes the behavior of partitioning.
- Use recursion depth as an early warning sign, not only total runtime.
- If possible, show a small visual or textual trace of how one bad pivot policy keeps producing unbalanced partitions.

Stretch goals (optional)

- Visualize partition sizes across the recursion tree.
- Compare deterministic and randomized variants in terms of robustness, not only speed.

3. External ORDER BY: sort more data than fits in memory

Goal. Database systems can sort data sets much larger than the available RAM. They do not do this by magic. A common idea is to sort small chunks in memory, write those sorted chunks to disk, and then merge them. In this project you will reproduce that idea on a smaller scale. Think of a log exporter, a gradebook export, or a large CSV report that must be sorted by several keys even though your program is not allowed to hold the whole file in memory at once.

Required content

- State a clear memory cap, for example a maximum number of records allowed in RAM at one time.
- Define a multi-key ordering rule clearly, for example timestamp ascending, priority descending, record ID ascending, and null values last.
- Generate sorted runs on disk, then merge them into the final output.
- Verify your result against a trusted in-memory sort on smaller data.
- Benchmark the effect of chunk size, number of temporary runs, and merge fan-out.
- Explain where the time goes: sorting time, merging time, and approximate I/O cost.

Suggested approaches to compare

- Method 1 (baseline): full in-memory sort on smaller data sets.
- Method 2: external sort with repeated 2-way merges.
- Method 3: external sort with a k-way merge driven by a heap.

Demo focus

- Explain the life cycle of the data: read chunk, sort chunk, write run, merge runs.
- Show one small example where the same records are sorted by multiple keys and ties are resolved correctly.
- Highlight the engineering trade-off: fewer merge passes usually help, but larger heap structures also have a cost.

Stretch goals (optional)

- Estimate total disk I/O performed by each variant.
- Allow the user to choose the sort specification at runtime.

4. Sorting a k-sorted stream: delayed packets and nearly ordered arrivals

Goal. Imagine timestamps from sensors, packets arriving over a network, or messages from several servers. They should be in order, but some items arrive slightly late. In many real systems each element is only a little bit away from its final sorted position. If every element is at most k positions away from where it belongs, then collecting the whole stream and running a full sort may be wasteful. Your task is to study this “almost sorted” situation and build a method that outputs correct results as early as possible.

Required content

- Define precisely what “k-sorted” means in your project.
- Implement the heap-based online algorithm that keeps only a buffer of size k+1.
- Compare it with at least two alternatives, for example full batch sorting and a local insertion-based repair method.
- Benchmark for different values of both n and k.
- Discuss not only runtime, but also memory use and delay before the first correct output appears.
- Include end-of-stream behavior and duplicate keys among your tests.

Suggested approaches to compare

- Method 1 (baseline): collect the whole stream and sort at the end.
- Method 2: min-heap online algorithm.
- Method 3: insertion-based local repair inside a moving window.

Demo focus

- Walk through a tiny stream by hand so that the audience can see why the heap method works.
- Show clearly when the online method is genuinely better than “just sort everything later”.
- Explain how the value of k changes both the memory requirement and the practical speed.

Stretch goals (optional)

- Estimate k online instead of being given it exactly.
- Show what goes wrong when the estimate is too small.

5. Interval analytics and resource allocation: sort once, answer many questions

Goal. A department manages meeting rooms, lab machines, or lecture spaces. Every reservation is an interval with a start time and an end time. Once those intervals are sorted carefully, the same data can answer several different questions: when is the system busy, how many resources are needed at once, and when does demand exceed capacity? In this project you will treat sorting as a tool that unlocks several nontrivial outputs from the same input.

Required content

- State your interval convention clearly, for example half-open intervals [start, end).
- From the same input, compute all of the following: merged busy intervals, total busy time, maximum overlap, the first time overlap exceeds a capacity C, and one valid assignment using the minimum number of rooms or machines.
- Handle touching intervals, nested intervals, identical endpoints, and empty input correctly.
- Compare a naive baseline with at least two sort-based methods.
- Explain separately why a heap helps in the resource-assignment part.
- Include at least one example where tie-breaking at equal timestamps changes the answer.

Suggested approaches to compare

- Method 1 (baseline): naive pairwise overlap checks.
- Method 2: sort by start time and merge intervals.
- Method 3: sort endpoints and sweep through time; use a heap to assign rooms efficiently.

Demo focus

- Show how one sorted order can be reused to answer several questions, not just one.
- Be careful with endpoint ties and explain the rule you chose.
- Use a small example to illustrate why the heap returns an efficient room assignment.

Stretch goals (optional)

- Add priorities or weights to intervals.
- Reconstruct and visualize one optimal room assignment, not just the number of rooms.

Closing note

Keep it practical. The goal is not to memorize that one sorting algorithm is always best. The goal is to learn how input structure, stability, memory limits, and benchmark design change the correct engineering choice.