18YZALG — Tutorial 1 Assignments

Tutorial 1 — Mini-Projects

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

Quick facts

Item	Description
Who	Student groups (recommended **2–3** students)
When	Next tutorial, i.e., February 24th
What	Short live demo
What we practice	Algorithm description, correctness tests, basic time/memory complexity, simple benchmarking
Grading	Extra credit to the to the continuous assessment component (tutorial part of the course)

What every group must include

- **Problem statement**: input format, output format, and edge cases (5–10 lines).
- **At least two approaches** to the same task (three is even better). Include a clear algorithm description (plain English or try with pseudocode).
- **Complexity**: informal Big-O time and memory discussion for each approach.
- **Benchmark**: measure runtime for multiple input sizes and interpret the trend (a table is enough; a plot is optional but welcomed).
- **Correctness checks**: a small test suite (simple `assert` tests are sufficient). Test your implementation on some trivial cases.
- **Conclusion**: when would you use which approach, and why?

Submission format

- You do not need to submit any files beforehand, the only deliverable here is the live demonstration.
- For the demonstration you can use a presentation format, Jupyter notebook, or whatever you are comfortable with. The only requirement is it has to contain all the required contents..

Live demonstration checklist (presentation)

Aim for a concise demo, 10-15 minutes is more than enough. You may split speaking roles inside the group, but it is optional.

Example presentation could look like:

- **1 minute**: Problem statement + what counts as a correct output.

- **2–3 minutes**: Approaches (Method 1 baseline, Method 2 improvement, Method 3 improvement). Show high-level logic, not every line.
- **1–2 minutes**: Complexity: explain Big-O in one or two sentences per method.
- **1–2 minutes**: Benchmark: show a runtime table and explain the scaling.
- **1–2 minutes**: Correctness: run tests or show representative tests.
- **1–2 minutes**: Final takeaway (which method wins when, and trade-offs).

Assessment

Maximum of 20 points for this given presentation, later translated to the overall tutorial part of the course assessment.

Criterion	Points	What we look for
Correctness & tests	0–5	Works on edge cases; tests are clear and actually run
Algorithm description & complexity	0–5	Approaches are explained; Big-O time/memory reasoning is plausible
Benchmark & interpretation	0–5	Multiple input sizes; fair timing; conclusions match results
Demo clarity	0–5	Clear structure and presentation

Mini-Projects

Pick exactly one. Each project is designed so you can compare multiple approaches, reason about complexity, and validate with tests and benchmarks.

1. Duplicate detection at scale

Goal. Given a list of IDs, decide whether any value appears at least twice.

Required content

- Define the input clearly (integers/strings are enough) and specify edge cases (empty list, all unique, all equal).
- Implement at least two methods and ensure they return identical results on the same tests.
- Benchmark across at least 5 input sizes and identify where the baseline becomes impractical.

Suggested approaches to compare

- Method 1 (baseline): pairwise comparison of all pairs (i < j).
- Method 2: sort a copy and scan adjacent elements.
- Method 3: scan with a set of seen elements.

Demo focus

- Show a tiny example list and walk through what each method does.
- Run your tests live (or show the command and output).
- Show the benchmark table and explain the scaling trend.

Stretch goals (optional)

- Return the first duplicate encountered in a stream (online variant).
- Return all duplicates and compare the cost to the boolean variant.

2. Membership queries: list vs sorted+binary vs set

Goal. You have a database of allowed IDs and a list of query IDs. For each query, decide membership.

Required content

- Specify input sizes using two parameters: n = database size, q = number of queries.
- Compare at least two strategies, one of which includes preprocessing.
- Benchmark for different q/n ratios (e.g., few queries vs many queries).

Suggested approaches to compare

- Baseline: for each query, linear search through a list (O(nq)).
- Preprocess: sort once + binary search each query (use `bisect`).
- Preprocess: build a set once + membership checks (average O(1) per query).

Demo focus

- Explain why preprocessing can pay off when q is large.
- Include at least one benchmark where sorting beats linear search.

Stretch goals (optional)

- Find the break-even point: for a fixed n, vary q and find when preprocessing wins.
- Add a short memory discussion (list vs set).

3. Fibonacci: recursion, iteration, memoization

Goal. Compute F(n) and study the impact of algorithm design choices.

Required content

- Define the Fibonacci sequence you use (F(0), F(1) base cases).
- Include at least one intentionally slow baseline method to demonstrate scaling issues.
- Benchmark for increasing n and report the largest n you can compute within a fixed time budget.

Suggested approaches to compare

- Baseline: naive recursion (exponential time).
- Improved: iterative loop (linear time).
- Improved: recursion with memoization (linear time).

Demo focus

- Show that naive recursion becomes unusable quickly (but still correct).

- Explain how memoization changes the number of repeated subproblems.

Stretch goals (optional)

- Count function calls in the naive recursion and show calls vs n.
- Compare memoization with bottom-up DP explicitly.

4. Prime numbers up to N: trial division vs sieve

Goal. Generate all primes up to N and compare algorithmic approaches.

Required content

- Define what you output (list of primes, count of primes, or boolean primality array).
- Include correctness checks for small N (e.g., N=30) where you know the exact primes.
- Benchmark for multiple N values and describe scaling.

Suggested approaches to compare

- Baseline: trial division by all integers up to n-1 (or up to n).
- Improved: trial division up to sqrt(n), skipping evens.
- Improved: sieve of Eratosthenes.

Demo focus

- Explain why sqrt(n) is enough for trial division.
- Explain why the sieve trades memory for speed.

Stretch goals (optional)

- Use `bytearray` for a compact sieve representation.
- Compare returning a list of primes vs a boolean mask.

5. Top-K frequent items (practical data task)

Goal. Given a text or a list of tokens, find the top-K most frequent items.

Required content

- Specify how you tokenize input (simple split is fine) and how you handle case/punctuation (state your choice).
- Implement at least two counting approaches and return the same top-K result.
- Benchmark on scaled input sizes (e.g., repeat the text).

Suggested approaches to compare

- Baseline: repeated counting (e.g., using `list.count` in a loop) to show inefficiency.
- Improved: dictionary counting (hash map).
- Optional: `collections.Counter` and compare constant factors.

- Optional: compare sorting all counts vs using a heap for top-K.

Demo focus

- Show your top-K output for a small sample text.
- Show a benchmark table and explain why the baseline scales poorly.

Stretch goals (optional)

- Add ties handling (stable ordering) and explain your rule.
- Compare memory usage for different implementations (qualitatively).

Closing note

Keep it simple. The goal is not maximum performance at any cost — it is learning how algorithm design, data structures, and complexity reasoning translate into real runtime behavior.