# Assignment: The Algorithmic Stress-Tester

Empirical Reality vs. Theoretical Complexity

**Course:** Data Structures & Algorithms **Type:** Individual project (programming + written analysis) **Suggested weight:** 15–20% of final grade **Estimated effort:** 8–12 hours

---

## 1. The Big Idea

Big O notation tells you how an algorithm *should* scale. Your CPU tells you how it *actually* runs. These two stories usually agree — but not always, and the gaps are where the real understanding lives.

In this project you will implement several solutions to **one** problem, each with a different time complexity, then build a benchmarking harness that measures how they behave as the input size $N$ grows. You will watch your exponential or cubic code choke on inputs your linear code shrugs off. Then you will explain, with evidence, *why* the measured curves do — and do not — match the theory.

The goal is not "the fast algorithm wins." The goal is to become someone who can predict, measure, and reason about performance rather than guess at it.

---

## 2. Learning Objectives

By completing this assignment you will be able to:

- Implement the same problem at multiple complexity classes and verify they all produce identical results.
- Design a fair benchmarking harness (warm-up, repeated trials, median timing, controlled inputs).
- Plot empirical runtime against $N$ and overlay theoretical growth curves.
- Identify and explain divergences between measured and predicted performance (constant factors, cache behavior, crossover points, language overhead, arithmetic cost).
- Communicate quantitative findings clearly in a short technical report.

---

## 3. Choose Your Problem

Pick **one** option. Both are valid; choose based on what interests you.

### Option A — Maximum Subarray Sum

Given an array of integers, find the largest sum of any contiguous subarray. Implement **all four**:

| Approach | Complexity |
|---|---|
| Brute force (all start/end pairs, re-sum each) | $O(N^3)$ |
| Optimized brute force (running sum) | $O(N^2)$ |
| Divide and conquer | O(N log N) |
| Kadane's Algorithm | O(N) |

### Option B — Fibonacci Sequence

Compute the *n*-th Fibonacci number. Implement **all three**:

| Approach | Complexity |
|---|---|
| Naive recursion | $O(2^N)$ |
| Iterative / memoized | O(N) |
| Matrix exponentiation (fast doubling) | O(log N) |

> **Option B caveat to investigate:** the O(log N) claim assumes arithmetic is O(1). For large *n*, Fibonacci numbers have thousands of digits and big-integer arithmetic is *not* constant time. Your measured curve may not look like log N — and explaining that is part of the assignment.

---

## 4. Requirements

### 4.1 Correct implementations

- Each variant must be a separate, named function.
- Include a small test that confirms **all variants return the same answer** on the same input before you benchmark anything. A fast wrong answer is worth nothing.

### 4.2 Input generation

- Generate random datasets across a wide range of *N*.
- **Option A:** N = 10 up to N = 1,000,000 (use a geometric scale, e.g. 10, 100, 1 000, 10 000, 100 000, 1 000 000).
- **Option B:** scale *n* appropriately per algorithm — the naive version may only survive to n ≈ 35–40, while the others reach far higher.
- Use a **fixed random seed** so your runs are reproducible, and reuse the *same* generated input across all variants at each *N*.

### 4.3 Benchmark harness

Your harness must:

- Time each variant at each *N*.

- **Skip or cap** runs that exceed a sensible time budget (e.g. 10 seconds) so a slow algorithm doesn't hang your whole experiment — record it as "did not finish."
- Run each measurement **multiple times** and report the **median** (or minimum), not a single noisy sample.
- Include a **warm-up run** that you discard, to avoid measuring cold-cache / interpreter-warmup effects.
- Output results in a clean, machine-readable form (CSV or similar) so you can plot them.

### 4.4 Visualization

- Produce at least one plot of **execution time vs. N** with all variants on the same axes.
- Overlay or compare against the **theoretical curves** (N, N log N, $N^2$, etc.), scaled to align with your data.
- A **log-log plot** is strongly recommended — on log-log axes a polynomial appears as a straight line whose slope is its exponent, which makes the complexity class visually obvious.

---

## 5. The Analysis (the part that matters most)

Your report must answer the following with reference to **your own data**, not generic textbook claims:

1. **Do the curves match the theory?** For each variant, does the measured growth match its Big O class? Use the log-log slope as evidence.
2. **Where does the slow code break?** At what $N$ did your $O(N^3)$ / $O(2^N)$ version hang or exceed the time budget? Roughly what runtime would it have needed to finish?
3. **Crossover points.** Is the asymptotically "better" algorithm always faster, even at small $N$? Find a value of $N$ (if any) where a "worse" algorithm wins, and explain why constant factors cause this.
4. **Explain the discrepancies.** Pick at least **two** factors below that show up in your data and explain how they affected your measurements:
   - Hidden constant factors and low-order terms
   - CPU cache behavior (cache-friendly linear scans vs. scattered access)
   - Memory allocation / recursion-stack overhead
   - Interpreter, JIT warm-up, or garbage-collection pauses
   - Integer overflow or big-integer arithmetic cost (especially Option B)
   - Measurement noise and how you controlled for it
5. **What surprised you?** One honest paragraph on something you expected to be true that the measurements contradicted.

---

## 6. Deliverables

1. **Source code** — all variants, the input generator, and the benchmark harness, runnable with clear instructions.

2. **Raw results** — your CSV (or equivalent) of timings.
3. **Plot(s)** — empirical vs. theoretical, ideally log-log.
4. **Report** — 2–4 pages (or equivalent) covering the analysis in Section 5. Embed your plots.

Submit as a single repository or archive with a short `README` explaining how to reproduce your results.

---

## 7. Technical Notes & Constraints

- **Language:** any language is allowed, but state which one and note its relevant quirks (interpreted vs. compiled, GC behavior, default integer width).
- **Overflow:** for Option A with large $N$, subarray sums can overflow 32-bit integers — use a wide enough type. For Option B, you will need arbitrary-precision integers for large $n$.
- **Recursion limits:** naive recursive solutions may hit stack/recursion-depth limits before they hit a time limit. If so, record that as the failure mode and explain it.
- **Fair timing:** measure only the algorithm, not the input generation or printing. Don't let one variant benefit from a cache the others didn't get.
- **No external algorithm libraries** for the core implementations — write them yourself. Plotting and timing libraries are fine.

---

## 8. Grading Rubric (100 pts)

| Criterion | Points |
|---|---|
| All variants implemented correctly and verified to agree | 25 |
| Benchmark harness: fair methodology (warm-up, repeats, median, time cap, shared inputs) | 20 |
| Data collection across full $N$ range + clear plots (log-log) | 15 |
| **Empirical vs. theoretical analysis & discrepancy explanations** | 30 |
| Code quality, reproducibility, and README | 10 |

---

## 9. Stretch Goals (optional)

- Add a **second language** and compare the same algorithm across both — the asymptotic class is identical, but the constants are not.
- Empirically **estimate the complexity** from your data by fitting the log-log slope, and compare it to the theoretical exponent.
- For Option B, measure the **big-integer arithmetic cost directly** and show how it bends the "O(log N)" matrix method away from a flat line.
- Add a third axis: **memory usage** vs. $N$, and discuss space–time trade-offs (e.g. memoization).

*Tip: start small. Get one correct implementation and one clean timing measurement working end-to-end before adding the rest. A working harness with two variants beats four half-finished variants you never managed to benchmark.*