

Measuring Complexity and Writing Tests

Seven problems. Three measure complexity by hand, three exercise exception handling, and the last asks you to write your first pytest suite — for somebody else's code.

Due: Homework 4 · Tue 8 Dec, 23:59

What this session is for

- Measure the practical difference between complexity classes yourself
- Bring an exponential algorithm down to linear with memoization
- Use try / except / raise correctly, and keep exceptions and assertions apart
- **Write your first pytest suite**

AI policy: Level 1

Permitted, with an [AI usage declaration](#).

A note on 11-7: you may have an AI suggest extra test cases, but **you must think through the edge cases first yourself**. [L8](#) noted that "empty" and "one" are exactly what an AI most often misses.

Setup

```
python3 -m pip install pytest
```

Part A — Complexity and memoisation

11-1 Measuring complexity

Write a script timing each of these at `n = 1000, 5000, 10000, 20000`:

1. `list.append(x)`, n times
2. `list.insert(0, x)`, n times
3. `x in list`, 100 lookups in a list of length n
4. `x in set`, 100 lookups in a set of size n

Produce a table of results and answer in your notes:

- **When n doubles, roughly how much does each timing grow?**
- What complexity does that imply for each?

This problem teaches you how to identify complexity empirically

You can classify code without reading it: **double n and the time doubles $\rightarrow O(n)$; it quadruples $\rightarrow O(n^2)$; it barely moves $\rightarrow O(1)$** . This is genuinely useful when handed code you do not understand.

11-2 String concatenation

Compare these three at `n = 5000, 10000, 20000` the same way:

```
# method one: concatenate in the loop
s = ""
for w in words:
    s += w

# method two: keep every intermediate result alive
parts = []
for w in words:
    parts.append(parts[-1] + w if parts else w)
s = parts[-1]

# method three: collect and join
s = "".join(words)
```

Give the timing table in your notes and answer:

- How far apart are methods one and three, and why is the gap not orders of magnitude?
- Why is method two so much slower?

The conclusion is not “concatenating in a loop is always $O(n^2)$ ”

The [lecture notes](#) measure this: when the string has exactly one reference, CPython extends it in place, so methods one and three land in the same range. **The moment an intermediate result is referenced by another name or held in a container, that optimization stops applying**, and only then does method two degrade to quadratic.

Write `join` anyway: it does not depend on that implementation detail.

11-3 Memoize a recursion

Take your `count_ways(amount, coins)` from [Lab 7](#) and do three things:

1. Add a counter and measure how many calls `count_ways(100, [1, 2, 5, 10])` needs
2. Add memoization with a dictionary
3. Measure again and compute the speedup

Give both numbers and the ratio in your notes.

The cache key must include every varying parameter

`count_ways` has two arguments that both change. Key on `amount` alone and `count_ways(10, [1, 2, 5, 10])` returns **129** when the correct answer is **11**. **Work out what actually identifies a subproblem uniquely.**

Part B — Exceptions and testing

11-4 Safe conversions

Implement three functions returning a default rather than crashing:

```
def safe_int(text: str, default: int = 0) -> int: ...
def safe_float(text: str, default: float = 0.0) -> float: ...
def safe_div(a: float, b: float, default: float = 0.0) -> float: ...
```

Requirement: **catch only specific exception types**; a bare `except:` is not acceptable.

In your notes, say which exceptions each function catches and why those.

11-5 Find and fix bad exception handling

The code below has **three** problems. Find them, fix them, and explain each in your notes.

```
def load_config(filename):
    try:
        f = open(filename)
        data = f.read()
        config = {}
        for line in data.split("\n"):
            key, value = line.split("=")
            config[key] = int(value)
        return config
    except:
        return {}
```

The three problems are at different levels

One is about the exception handling itself, one is about resource management, and one is about the consequences of swallowing errors.

Your rewritten `load_config` must tell “the file is not there” apart from “the contents are malformed”. Returning an empty dictionary for both is not enough.

11-6 A custom exception

Modify the `Temperature` class from [Lab 9](#): instead of printing a message and returning `False` for sub-absolute-zero values, **raise a custom exception**.

```
class TemperatureError(Exception):
    """Invalid temperature."""
    ...
```

Answer in your notes: **when is raising better than returning `False`, and when is the reverse true?**

11-7 Test somebody else's code

Here is a module that is "already written". Your job is not to change it — it is to write a **pytest** suite for it.

```
# textstats.py
def word_count(text: str) -> int:
    """Count the words."""
    return len(text.split())

def longest_word(text: str) -> str:
    """Return the longest word. On a tie, the first one."""
    words = text.split()
    return max(words, key=len)

def average_word_length(text: str) -> float:
    """Return the mean word length."""
    words = text.split()
    return sum(len(w) for w in words) / len(words)
```

Write `test_textstats.py` such that:

- each function has **at least four tests**
- coverage includes: the normal case, an empty string, a single word, repeated spaces
- `pytest.raises` is used at least once

Then run `python3 -m pytest -v` and paste the output into your notes.

Finally: how many of these three functions misbehave on an empty string, and how?

The point is testing code you did not write

You did not write this module, so your only guide is **the behavior the docstrings describe** — which is precisely [L8's](#) "write the tests before reading the code".

You will find the documentation and the implementation do not entirely agree. **That is what testing is for.**

Before you submit

- ☐ 11-1 and 11-2 have complete timing tables and complexity conclusions
- ☐ 11-2 answers why methods one and three are the same order and why method two is slower
- ☐ 11-3's cache key includes every varying parameter
- ☐ 11-4 contains no bare `except:`
- ☐ 11-5 identifies all three problems, and the rewrite separates a missing file from malformed contents
- ☐ 11-6 answers when to raise and when to return
- ☐ 11-7 has at least four tests per function and uses `pytest.raises`
- ☐ 11-7 answers the empty-string question
- ☐ AI usage declaration written