

复杂度实测与编写测试

七道题。前三题动手测复杂度，中间三题练异常处理，最后一题是这门课第一次要求你写 `pytest` 测试，而且是给别人的代码写。

截止：小作业 4 · 12 月 8 日 周二 23:59

本次目标

- 亲手测出不同复杂度的实际差别
- 用记忆化把一个指数算法降到线性
- 正确使用 `try / except / raise`，分清异常和断言
- 写出第一套 `pytest` 测试

AI 政策：Level 1

允许使用，须附 [AI 使用声明](#)。

11-7 特别提醒：可以让 AI 帮你补测试用例，但边界情况必须你自己先想一遍。[L8](#) 说过，AI 最常漏的就是「空」和「一个」。

环境准备

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

A 段 · 复杂度与记忆化

11-1 复杂度实测

写一个脚本，对下面每个操作测量在 `n = 1000, 5000, 10000, 20000` 时的耗时：

1. `list.append(x)` 执行 n 次
2. `list.insert(0, x)` 执行 n 次
3. `x in list` 在长度 n 的列表里查 100 次
4. `x in set` 在大小 n 的集合里查 100 次

把结果整理成一张表，在提交说明里回答：

- n 翻倍时，每个操作的耗时大约翻几倍？
- 由此推断它们各自的复杂度是什么

这道题在教你怎么判断复杂度

不用看代码也能判断： n 翻倍，耗时翻 2 倍 $\rightarrow O(n)$ ；翻 4 倍 $\rightarrow O(n^2)$ ；几乎不变 $\rightarrow O(1)$ 。这个方法在你拿到一段看不懂的代码时特别有用。

11-2 字符串拼接

用同样的方法对比这三种写法在 `n = 5000, 10000, 20000` 时的耗时：

```
# 写法一：循环里直接拼接
s = ""
for w in words:
    s += w

# 写法二：每次都把中间结果留下来
parts = []
for w in words:
    parts.append(parts[-1] + w if parts else w)
s = parts[-1]

# 写法三：先收集再 join
s = "".join(words)
```

在提交说明里给出耗时表，并回答：

- 写法一和写法三差多少？为什么差距不是数量级？
- 写法二为什么慢得多？

结论不能写成「循环拼接一律是 $O(n^2)$ 」

[讲义](#) 里实测过：当字符串只有 `s` 一个引用时，CPython 可以原地扩展，写法一和写法三在同一量级；一旦中间结果被别的名字或容器引用，这个优化就不成立，写法二才退化成平方级。

写代码时仍然推荐 `join`，因为它不依赖这条实现细节。

11-3 给递归加记忆化

拿你在 [Lab 7](#) 写的 `count_ways(amount, coins)`（硬币组合计数），做三件事：

1. 加一个计数器，测出 `count_ways(100, [1, 2, 5, 10])` 需要多少次函数调用
2. 用字典给它加上记忆化
3. 再测一次调用次数，算出加速比

在提交说明里给出两个数字和比值。

记忆化的键要包含所有变化的参数

`count_ways` 有两个参数都在变。如果你只用 `amount` 当键, `count_ways(10, [1, 2, 5, 10])` 会返回 **129**, 而正确答案是 **11**。想清楚: 什么才是唯一确定一个子问题的东西?

B 段 · 异常与测试

11-4 安全的转换

实现三个函数, 转换失败时返回默认值而不是崩溃:

```
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: ...
```

要求: 只捕获具体的异常类型, 不许写裸的 `except:` 。

在提交说明里写出每个函数你捕获了哪些异常, 以及为什么是这些。

11-5 找出并修正错误的异常处理

下面这段代码有三个问题。找出来, 改对, 并在提交说明里逐条说明。

```
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 {}
```

三个问题分属不同层面

一个是异常处理本身的，一个是资源管理的，一个是「吞掉错误」造成的后果。

改写后的 `load_config` 要能区分「文件不存在」和「内容格式不对」两种情况，不要都返回空字典。

11-6 自定义异常

给 [Lab 9](#) 的 `Temperature` 类改造：把「低于绝对零度」从「打印一句话返回 False」改成抛出自定义异常。

```
class TemperatureError(Exception):  
    """温度不合法。"""  
    ...
```

在提交说明里回答：什么情况下抛异常比返回 False 更好？什么情况下相反？

11-7 给别人的代码写测试

下面是一个「已经写好」的模块。你的任务不是改它，是给它写一套 `pytest` 测试。

```
# textstats.py
def word_count(text: str) -> int:
    """统计单词数。"""
    return len(text.split())

def longest_word(text: str) -> str:
    """返回最长的单词。长度相同时返回先出现的。"""
    words = text.split()
    return max(words, key=len)

def average_word_length(text: str) -> float:
    """返回平均单词长度。"""
    words = text.split()
    return sum(len(w) for w in words) / len(words)
```

写一个 `test_textstats.py`，要求：

- 每个函数至少 4 个测试
- 必须覆盖：正常情况、空字符串、单个单词、多个空格
- 至少用一次 `pytest.raises`

然后运行 `python3 -m pytest -v`，把输出贴到提交说明里。

最后回答：这三个函数里，有几个在空字符串上会出问题？分别是什么问题？

这道题的重点是「测别人的代码」

你没写这个模块，所以你能从文档字符串描述的行为出发去设计测试，这正是 [L8](#) 说的「先写测试再看代码」。

你会发现文档说的和代码做的不完全一致。这就是测试的价值。

提交前自查

- ☐ 11-1、11-2 有完整的耗时表和复杂度推断

- ☐ 11-2 回答了写法一与写法三为何同量级、写法二为何慢
- ☐ 11-3 的记忆化键包含了所有变化的参数
- ☐ 11-4 没有裸的 `except:`
- ☐ 11-5 找出了三个问题，改写后能区分缺文件与格式错误
- ☐ 11-6 回答了「何时抛异常、何时返回值」
- ☐ 11-7 每个函数至少 4 个测试，用过 `pytest.raises`
- ☐ 11-7 回答了空字符串的问题
- ☐ AI 使用声明写了