Tutorial 3: Quality of an Algorithm and Python Collections

The Art of Computer Programming

30 September 2026

Objective. This tutorial puts into practice the third lecture: computing the complexity of an algorithm on paper, checking it by measuring running times, and choosing the right Python collection for a task. Part A is done with pen and paper (no computer); Part B implements the algorithms of Part A in Python and in C and times them; Part C is about Python lists, sets, dictionaries, and comprehensions. Do Parts A and B in the first half of the session and Part C in the second half; Part D is for those who finish early. Exercises are roughly ordered by importance within each part.

Reminders

- Create a folder `tutorial03` next to `tutorial02`, and write each program in its own file named after the exercise. Compile C programs with `$ gcc -Wall -Wextra -std=c23 triangle.c -o triangle`.
- Timing a piece of code (tutorial 2): in Python, `import time` and `time.perf_counter()` before and after, the difference is in seconds; in C, `#include <time.h>`, `clock()` before and after, the difference divided by `CLOCKS_PER_SEC` is in seconds.
- *The doubling test.* To check a complexity experimentally, time the program for input sizes n, $2n$, $4n$, $8n$ and look at the ratio between successive times: about 2 for $\Theta(n)$, about 4 for $\Theta(n^2)$, a little more than 2 for $\Theta(n \log n)$, about $\sqrt{2} \approx 1.4$ for $\Theta(\sqrt{n})$, and close to 1 for $\Theta(1)$ or $\Theta(\log n)$. Times below a millisecond are unreliable: choose n so that the smallest run takes at least a few hundredths of a second, and never compare times measured on different machines.

Part A: Complexity on paper

Q1. Asymptotic notation.

(a) Order the following functions of n from slowest to fastest growing: $n \log n$, 2^n, n^2, $\log n$, 1, n, $\sqrt{n}$, n^3.

(b) For each statement, say whether it is true or false and justify in one line: $3n^2 + 10n \in O(n^2)$; $n \log n \in O(n^2)$; $n^2 \in O(n \log n)$; $2^{n+1} \in O(2^n)$; $2^{2n} \in O(2^n)$; $\log_2 n \in \Theta(\log_{10} n)$.

(c) Go back to the definition of $O()$ given in the lecture: find explicit values of N and α such that $3n^2 + 10n \leq \alpha n^2$ for all $n > N$. Is there a single correct answer?

Q2. A product. Consider the following algorithm.

Input: an integer $n \geq 1$
1: $p \leftarrow 1$
2: **for** $i \leftarrow 1$ **to** n **do**
3: $\quad p \leftarrow p \times (1 + 1/n)$
4: **end for**
5: **return** p

(a) Compute the result for $n = 1$, $n = 2$, $n = 4$. Which mathematical quantity does it compute? What does it tend to when n grows?

(b) Count the elementary operations as a function of n (one assignment and one comparison per loop iteration, one division, one addition, one multiplication, one assignment per loop body). Give the complexity in $\Theta()$ notation.

(c) Propose an algorithm computing exactly the same value in $O(\log n)$ multiplications (tutorial 2, fast exponentiation).

Q3. A triangle. Same questions for the following algorithm: exact number of times line 4 is executed, what the algorithm returns, complexity in $\Theta()$ notation, and an algorithm computing the same value in $O(1)$.

Input: an integer $n \geq 1$
1: $s \leftarrow 0$
2: **for** $i \leftarrow 1$ **to** n **do**
3: **for** $j \leftarrow i$ **to** n **do**
4: $s \leftarrow s + 1$
5: **end for**
6: **end for**
7: **return** s

What changes if line 3 becomes **for** $j \leftarrow 1$ **to** 10? If it becomes **for** $j \leftarrow 1$ **to** $i \times i$? Give the complexity in each case.

Q4. Doubling. Same questions for the following algorithm: exact number of times line 5 is executed (as a function of n, using $\lfloor \log_2 n \rfloor$), complexity in $\Theta()$ notation, and a faster algorithm computing the same value (in $O(\log n)$, or in $O(1)$ if you know `n.bit_length()` in Python).

Input: an integer $n \geq 1$
1: $count \leftarrow 0$
2: **for** $i \leftarrow 1$ **to** n **do**
3: $j \leftarrow 1$
4: **while** $j \leq n$ **do**
5: $count \leftarrow count + 1$
6: $j \leftarrow 2 \times j$
7: **end while**
8: **end for**
9: **return** $count$

What is the complexity if line 6 becomes $j \leftarrow j + 2$? If line 4 becomes **while** $j \leq i$?

Q5. A recursive function. Consider the function defined by $f(0) = 1$ and $f(n) = 1 + 1/f(n-1)$ for $n \geq 1$, computed by the recursive program that follows directly from this definition.

(a) Compute $f(0), \ldots, f(5)$ as fractions. Recognize the numerators and denominators (tutorial 2). What does $f(n)$ tend to?
(b) How many calls does the computation of $f(n)$ make? Give the time complexity, then the *space* complexity, remembering where the pending calls live (lecture 2).
(c) Write an iterative version with the same time complexity and $O(1)$ space.

Q6. Best, worst, average. The primality test of tutorial 2 tries every divisor $d = 2, 3, \ldots$ with $d \times d \leq n$ and stops at the first one that divides n.

(a) What is the worst-case complexity as a function of n? For which inputs is it reached? What is the best case?
(b) Is it correct to say that linear search in an array of n elements (first example of the lecture) takes $\Omega(n)$ time? $O(1)$ time? Explain what each statement means and for which case (best, worst, average) it holds.

Q7. Mystery functions. For each of the following Python functions, work out (approximately) the value returned as a function of n, and its complexity in $\Theta()$ notation. Do it on paper; you will check your answer in Q10. Beware: the two functions look alike, but do not have the same complexity.

Python
```python
def mystery1(n):
    c = 0
    i = n
    while i > 0:
        for j in range(i):
            c += 1
        i = i // 2
    return c
```

Python
```python
def mystery2(n):
    c = 0
    for i in range(1, n + 1):
        j = i
        while j > 0:
            c += 1
            j = j // 2
    return c
```

Part B: Measuring

Q8. The doubling test in Python. Write the algorithms of Q3 and Q4 as Python functions `triangle(n)` and `doubling(n)`. For each, print the value returned and the time taken for $n = 500$, 1 000, 2 000, 4 000

(`triangle`) and $n = 100\,000$, $200\,000$, $400\,000$, $800\,000$ (`doubling`), and the ratio between each time and the previous one. Compare with the doubling test of the reminders. Then time your $O(1)$ versions on the same inputs.

Q9. The same in C.

(a) Write the algorithm of Q2 in C, with `p` of type `double`, and print the result for $n = 1000$. If you get 1, explain why using lecture 1 and fix it.
(b) Write `triangle` and `doubling` in C (with `long long` results) and time them for the sizes of Q8, then for sizes 16 times larger. Compare the C times with the Python times for the same n: what is the constant factor between the two languages?
(c) Compile again with `-O2` added to the `gcc` command line (this asks the compiler to optimize the code) and run the doubling test again. Which function still behaves as predicted, and which one does not? Look at the times of `triangle` for much larger n (10^7, 10^8) and guess what the compiler did.

Q10. Mystery solved. Type in the two functions of Q7 and apply the doubling test, starting from $n = 10^6$ for `mystery1` and from $n = 125\,000$ for `mystery2` (why not the same sizes?). Compare the ratios with your predictions. Also print c/n for `mystery1` and $c/(n \log_2 n)$ for `mystery2` (`import math`, `math.log2(n)`) for each n: what do they tend to?

Q11. Linear versus binary search. Build a list of n random integers between 0 and $10n$ (`import random`, then `random.randint(0, 10 * n)` in a loop) and sort it with `l.sort()`, which takes $O(n \log n)$ (lecture on sorting). Write a function `linear_search(l, x)` that returns a position of `x` in `l`, or `-1`, with a `for` loop, and type in the function `binary_search(l, x)` of the lecture.

(a) Check on a few values that both functions agree.
(b) Time 100 searches of random values (some absent, some present) with each function, for $n = 10^4$, 10^5, 10^6. By which factor does each time grow when n is multiplied by 10? Which complexity does this suggest?
(c) Add a counter of loop iterations to `binary_search` and print its maximum over the 100 searches for $n = 10^6$. Compare with $\log_2 n$.
(d) Also time `x in l`, which does a linear search in the interpreter's C code. How much faster than your Python loop is it? Does it change the complexity?

Part C: Python collections

Q12. Predict before running. Write down what the following program (left column, then right column) prints, and which line, if any, raises an error; then run it to check. Explain using the notions of object reference (lecture 2), copy, and mutability (lecture 3).

Python

```
def extend1(l):
    l.append(1)


def extend2(l):
    l = l + [1]


a = [1, 2, 3]
b = a
c = a[:]
b.append(4)
```

Python

```
c.append(5)
print(a, b, c)
print(a is b, a is c, a == b)
extend1(a)
extend2(a)
print(a)
t = (1, [2, 3])
t[1].append(4)
print(t)
t[0] = 0
```

Q13. Cheap and expensive list operations.

(a) Build a list of the integers from 0 to $n - 1$ in two ways: with `l.append(i)` for increasing i, and with `l.insert(0, i)` for decreasing i. Time both for $n = 10\,000$, $20\,000$, $40\,000$, $80\,000$. Explain the ratios using the table of complexities of the lecture.
(b) Write a function `rotate(l, k)` returning the list rotated by k positions to the left: `[1, 2, 3, 4, 5]` rotated by 2 gives `[3, 4, 5, 1, 2]`. Do it once with slicing and concatenation, once with a loop

repeating k times `l.pop(0)` followed by `l.append(...)`. Give the complexity of each, and check by timing with $n = 20\,000$ and $k = n/2$.

Q14. Membership test. For $n = 10^4$, 10^5, 10^6, build `l = list(range(n))`, `s = set(l)`, and `r = range(n)`, then time 100 tests `x in l`, `x in s`, `x in r` for random `x` between 0 and $2n$. Explain the three behaviors with the lecture. Why choose random values up to $2n$ rather than up to n?

Q15. Counting words. Download a long text, for instance *Alice's Adventures in Wonderland* from `https://www.gutenberg.org/cache/epub/11/pg11.txt`, and save it as `alice.txt` in your tutorial folder. Reading a file will be explained in a later lecture; for now, the following two lines give you the list of its words, in lower case:

Python

```
text = open("alice.txt", encoding="utf-8").read().lower()
words = text.split()
```

(You may want to strip punctuation from each word with `w.strip(".,;:!?_()")`, adding to the string any other character you see sticking to words, such as curly quotes.)

(a) Print the number n of words, then the number d of *distinct* words obtained with a set. What is the complexity of building the set?
(b) Count how many times each word appears, in a dictionary mapping each word to its number of occurrences. What is the complexity of building it? Find the most frequent word and its count with a loop over `counts.items()`.
(c) How many words appear exactly once? Print those of length at least 12.

Q16. Two-sum. Given a list `l` of n integers and a target T, we want to know whether there are two positions $i \neq j$ with `l[i] + l[j] == T`.

(a) Write the obvious function with two nested loops. What is its complexity?
(b) Write a second function that scans the list once, keeping in a set the values already seen: for each `x`, check whether `T - x` is in the set before adding `x` to it. Why does this correctly handle $i \neq j$? What is its complexity?
(c) Time both functions on a list of n random integers between 0 and $10n$ with $T = -1$, for $n = 1\,000$, $2\,000$, $4\,000$, $8\,000$. Why choose $T = -1$?

Q17. Comprehensions. Write each of the following as a single comprehension, and give its complexity.

(a) The list of the cubes of the multiples of 3 below 100.
(b) A dictionary mapping each word of `["one", "two", "three"]` to its length.
(c) The list of pairs (i, j) with $0 \leq i < j < 20$ and $i + j$ prime, using `is_prime` from tutorial 2.

Then rewrite (c) as explicit nested loops.

Part D: If you have time

Q18. Anagrams. Two words are anagrams if they have the same letters in a different order (`listen` and `silent`). Write a function that takes a list of words and returns a dictionary mapping each sorted sequence of letters, `"".join(sorted(w))`, to the list of words of the input having these letters (`sorted(w)` returns the list of the characters of `w` in order, in $O(m \log m)$ for a word of length m; `join` glues them back into a string): for `["listen", "silent", "rat", "tar", "art"]`, the result maps `"eilnst"` to `["listen", "silent"]` and `"art"` to `["rat", "tar", "art"]`. Give the complexity as a function of the number n of words and their maximum length m. Apply it to the distinct words of Q14 and print the groups of at least three words.

Q19. Social network. A social network is represented as a dictionary mapping each person to the set of their friends, e.g., `{"Alice": {"Bob", "Carol"}, "Bob": {"Alice"}, "Carol": {"Alice"}}`. Write the following functions and give the complexity of each as a function of the number of friends k of the people involved.

(a) `add_friendship(net, a, b)`: friendship is symmetric, and `a` or `b` may not be in the network yet.
(b) `are_friends(net, a, b)`.
(c) `friends_of_friends(net, a)`: the set of people at distance exactly 2 from `a`, i.e., friends of friends who are neither `a` nor a friend of `a`.

Test on a random network of 1 000 people (numbered from 0) and 5 000 random friendships, and find the person with the most friends.

Q20. Sieve of Eratosthenes. To find all primes up to n, create a list `prime = [True] * (n + 1)`, set positions 0 and 1 to `False`, then for each i from 2 with $i \times i \leq n$, if `prime[i]` is still `True`, set `prime[j]` to `False` for `j in range(i * i, n + 1, i)`. The positions still `True` are the primes. Implement it, count the primes up to 10^6 (there are 78 498), and compare the time with `count_primes` from tutorial 2. The sieve runs in $O(n \log \log n)$; what is the complexity of the tutorial 2 approach? What is the space complexity of the sieve?

Check

Whether or not you reach the end of the sheet, by the end of the session you should be comfortable with: reading a loop nest and giving its complexity in $\Theta()$ notation; checking a complexity experimentally with the doubling test; knowing which operations on Python lists are $O(1)$ and which are $O(n)$; using a set rather than a list when all you need is a membership test; using a dictionary to count or to group; writing a simple comprehension. If one of them is still unclear, ask us, or finish the corresponding exercises at home.