Project:
Algorithmic complexity and graphs

Group: Johannes Timmreck, Alexander Prosch, Jonas Conrad

Table of Contents

3. Complexities

Function 1

```
def function_1(n: int) -> None:
    temp_list = list()
    for i in range(n**2):
        temp = 0
        for j in range(i):
            temp += j
        temp_list.append(temp)
    sum(temp_list)
```

This algorithm uses a loop within a loop. For that reason, it might be tempting to say this algorithm has a runtime complexity of `O(n**2)`. However, the outer loop gets factored by 2, resulting in the same factorization of all operations within the loop itself.
The function starts off by initializing the object `temp_list` as a type list. As this operation is independent of the variable `n` and the list is created empty, we can consider it as `O(1)`. In the next line, we enter the outer loop, looping over the range of `n` factored by 2, resulting in `O(n**2)` which is the base factorization of all following operations. Although the outer loop is factored by 2, the initializing of `temp` stays `O(1)` but it will be called `n**2` times at the end.
Following this is the initialization of the object `temp` in the inner for-loop, using the value of `i` as its base. The inner for-loop runs at most `n**2` times, however, due to the factorization of `n**2` by the outer loop, this for-loop runs with a complexity of `O(n**4)` over all its lines.
`temp += j` is independent of the variable `n` thus being `O(1)`, but it will be called `n**4` times at the end. This marks the end of the inner for-loop. The outer for-loop ends with `temp` being appended to `temp_list` which results in `O(1)`. The closing function of `function_1` is a `sum` of the content of `temp_list`. This operation is `O(n**2)` as `temp_list` will be of size `n**2` by the end of the for-loop.
The upper bound of the complexity of `function_1` is `O(n**4)`. A verification for this complexity can be seen in figure 1.

Written out complexity:

```
1 + n**2 * (1 + (n**2 * 1) + 1) + n**2
```

Shortened complexity:

```
1 + 3n**2 + n**4
```

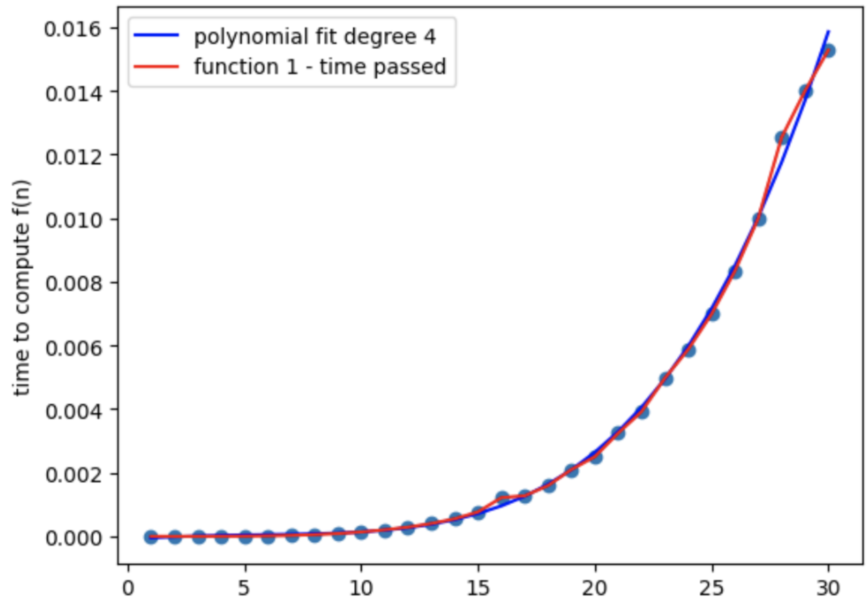

Figure 1. Polynomial fitting of n**4 to the measured time of function_1()

Function 2

```
def function_2(n: int) -> None:
    print(n)
    for i in range(n):
        temp_list = [j+i for j in range(n)]
        shuffle(temp_list)
        max(temp_list)
```

Even if the code seems a bit confusing at first, the nested loop should be a giveaway - there is a for-loop inside another for-loop. This algorithm has `O(n**2)` runtime complexity, which means it has quadratic complexity. It starts off with an `O(1)` print function and enters the outer for-loop right after, setting the baseline of `O(n)` for the following lines. The following line is the most tricky one of the function, as it houses an inline for-loop, looping over the range of the variable `n`. The outer loop closes off by first shuffling the created list and retrieving the largest value. Both of these functions need to go over the list once thus having a complexity of `O(n)` each.
The upper bound of the complexity of `function_2` is `O(n**2)`. A verification for this complexity can be seen in figure 2.

Written out complexity:
`1 + n * (1 + n * (1) + n + n)`
Shortened complexity:
`1 + 3n**2`

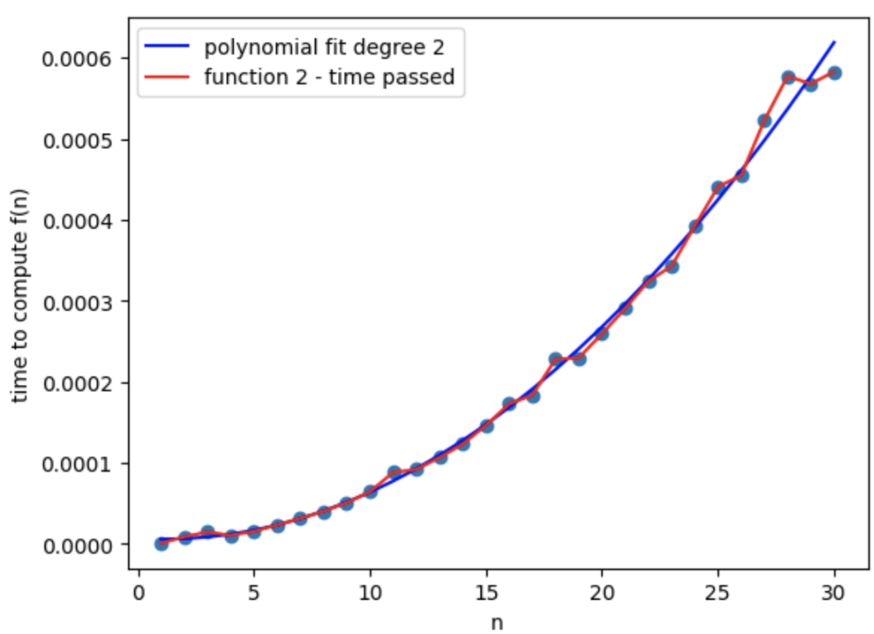

Figure 2. Polyinomail fitting of n**2 to the measured time of function_2()