Exercises for Algorithms and Data Structures Lecture 3

Question 1. Explain why measuring the running time empirically is not a good method of comparing the complexity of different algorithms. Explain how the methodology of counting operations of algorithms overcomes these limitations.

Question 2.
Listing 1 contains Python code for a function that returns the maximum element of a given list of integers.

a) What is the cost $T(n)$ of this algorithm for an input list of length/size n? In your analysis, use constants for every operation.

b) Based on your answer above, show whether the following statements are true using the useful limit theorem:

1. $T(n) \in O(n^2)$, $T(n) \in \Theta(n^2)$, $T(n) \in \Omega(n^2)$
2. $T(n) \in O(\log(n))$, $T(n) \in \Theta(\log(n))$, $T(n) \in \Omega(\log(n))$
3. $T(n) \in O(n\log(n))$, $T(n) \in \Theta(n\log(n))$, $T(n) \in \Omega(n\log(n))$
4. $T(n) \in O(n)$, $T(n) \in \Theta(n)$, $T(n) \in \Omega(n)$

c) Why do we not distinguish between the best and worst case scenarios (T_{best} and T_{worst}) for this function?

d) Perform the same analysis as in a), but now focus only on the basic operation. What is the basic operation? And what is the cost of the algorithm in terms of the basic operation, i.e., $B(n)$? Is $B(n) \in \Theta(n)$?

```
# Function that returns the maximum integer of a non-empty list
def find_max_element(ls: typing.List[int]) -> int:
    max_element = ls[0]
    for i in range(1, len(ls)):
        if ls[i] > max_element:
            max_element = ls[i]
    return max_element
```

Listing 1: Max element Search

Question 3.
Listing 2 contains Python code for determining whether a given integer value occurs in a numpy matrix. Answer the following questions about this function:

a) What is the best-case scenario for this code (in which the number of operations it performs is minimal)?

b) Similarly, what is the worst-case scenario for this code (maximum number of operations)?

c) What is the basic operation of the code?

d) Write down the number of basic operations that will be executed for these two cases, that is, B_{best} and B_{worst}? Think about what variables you use. (hint: use multiple!)

e) What is the tightest big-O bound (O) on the time complexity for two cases (best/worst)?

f) What is the tightest omega bound (Ω) on the time complexity for the two cases (best/worst)?

g) What is the Θ-complexity for the two cases (best/worst)?

```
# Function that returns whether a value is present in a matrix
def matrix_search(matrix: np.ndarray, value: int) -> bool:
    for y in range(matrix.shape[0]):
        for x in range(matrix.shape[1]):
            if matrix[y,x] == value:
                return True
    return False
```

Listing 2: Matrix Search

Question 4. (Levitin: exercise 2.1.2.a) Consider the following algorithm for finding the difference between two $n \times n$ matrices A and B:

```
for i in range(n):
    for j in range(n):
        diff[i,j] = A[i,j] - B[i,j]
```

What is its basic operation? How many times is it performed as a function of the matrix order n? As a function of the total number of elements in the input matrices?

Question 5.
(Levitin: exercise 2.2.5) List the following functions according to their order of growth from the lowest to the highest:

$$(n^2+3)!, \quad 2\log_2(n+50)^5, \quad 3^{3n}, \quad 0.05n^{10}+3n^3+1 \quad (\ln n)^3, \quad \sqrt{n}, \quad 3^{2n}$$

Here, $\ln n$ is the natural logarithm of n, also $\log_e n$.

Question 6. (Levitin: exercise 2.3.5)
Consider the following algorithm:

```
def foo(A: np.ndarray) -> float:
    val = 100
    sum_greater = 0
    sum_less = 0
    for i in range(len(A)):
        if A[i] > val:
            sum_greater = A[i]
        if A[i] < val:
            sum_less = A[i]
    }
    return sum_greater - sum_less
```

a) What does this algorithm compute?

b) What is its basic operation?

c) How many times is the basic operation executed?

d) What is the efficiency class of this algorithm?

e) Suggest an improvement or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

Question 7.
Question 1 from Levitin 2.2. Questions 4, 5, and 6 from Levitin 2.3.