Assignment 1 [30 points max.]

General reminder: solutions without any justification / explanation on how you got the answer are considered insufficient. Providing the code written in some programming language (e.g., Python) is not expected and does not typically count as sufficient justification / explanation (especially, when you should demonstrate the behaviour of some algorithm). Do not use AI tools.

Compulsory [max. 5 points]

1. Find asymptotic complexity for the (real) complexity expressed by the recurrent formula: **[5 points]**
   a. $T(n)=7T(n/4) + n^2$
   b. $T(n)=3T(n/10) + n$
   c. $T(n)=9T(n/3) + n^2$
   d. $T(n)=7T(n/2) + n^2$
   e. $T(n)=T(\sqrt{n}) + 2$

Elective [up to 15 points]

2. Is a) $2^{n+1} = O(2^n)$, and b) $2^{2n} = O(2^n)$? Prove your answer. **[2 points]**

3. Consider the following algorithm for primality test:

```
Input: An integer n >= 2
Output: true, if n is prime and false otherwise

    1.  s = $\lfloor\sqrt{n}\rfloor$
    2.  for i = 2 to s do
    3.      if n is divisible by i then return false
    4.  end for
    5.  return true
```

   a. Find (asymptotic) complexity of this algorithm in the worst-case, $O(f(n))$, and best-case, $\Omega(g(n))$, scenario. Prove your answer. **[2 points]**
   b. Propose a way to reduce the number of divisions. **[3 points]**

4. Consider a program performing *n* operations, one per one nanosecond. Construct a table showing how long it takes to perform *n* = 8, 32, 64, 256, 4096, 16384, 65536, 1048576 operations, if the program works with time-complexity: $\log n$, n, $n \log n$, n^2, a n^3, whereas the base of logarithms is 2. Use time units most appropriate for humans to understand the impact of different time-complexities. **[3 points]**

   Hint: use Excel or similar

5. Order the following 25 functions by their rate of growth such that $g_1=\Omega(g_2)$, $g_2=\Omega(g_3)$,… $g_{24}=\Omega(g_{25})$: **[5 points]**

$\left(\frac{3}{2}\right)^n$	$\left(\sqrt{2}\right)^{\log n}$	n^2	$(\log n)!$	n^3
$\log^2 n$	$\log(n!)$	$2^{(2^n)}$	$n^{1/\log n}$	$\log\log n$
$n\cdot 2^n$	$n^{\log\log n}$	$\log n$	2^n	$2^{\log n}$
$(\log n)^{\log n}$	$4^{\log n}$	$(n+1)!$	$\sqrt{\log n}$	$n!$
$2^{\sqrt{2\cdot\log n}}$	n	$n\cdot\log n$	1	$(\log\log n)^{(2^n)}$

Hints:

The base of logarithms is 2.

$$2^{\sqrt{2\cdot\log n}} = n^{\sqrt{2/\log n}}$$

$$\left(\sqrt{2}\right)^{\log n} = \sqrt{n}$$

$$n! \geq c\cdot\left(\frac{n}{2}\right)^{\left(\frac{n}{2}\right)}\ ; c > 0$$

Elective – BONUS [up to 10 points]

Rules: You may choose either block A, which is designed for working alone, or block B, which is designed for working in teams of 2-4 students. Mixing tasks from these blocks is not allowed, the student needs to specify clearly, which block they chose.

Block A

BONUS 1 **[5 points]**

Let A be an array of different integers. For each element A[i], find an index j such that j < i, and A[j] < A[i], and there is no index k such that k > j, k < i, and A[k] < A[i]. In other words, the element A[j] is the leftmost nearest smaller element of A[i].

a. Write indices for A = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15}.
b. Design an algorithm that can do it with complexity $\Theta(n)$.

Hint: you need to avoid testing all elements all the time

BONUS 2 **[5 points]**

Let A and B arrays of integers ranging from 0 to 9. Design an **efficient** algorithm that can decide if factorial products of these two arrays are the same. A factorial product of an array of numbers is a multiplication of factorials of these numbers, e.g., the factorial product of A = {0, 1, 2, 3} is $0!\cdot 1!\cdot 2!\cdot 3!$.

Hint: do not calculate the factorial product, think about definitions of the factorial

Block B (TEAM)

BONUS 1 **[10 points]**

Consider the following Python code:

```
def hybrid_algo(n, memo={}):
    if n in memo:
        return memo[n]
    if n == 0:
        return 0
    if n == 1:
        return 1

    k = bin(n).count('1')  # Count of 1s in binary representation of n

    if k % 2 == 0:
        result = hybrid_algo(n // 2, memo) + n
    else:
        result = hybrid_algo(n - 1, memo) - n

    memo[n] = result
    return result
```

a. Derive the recurrence relation that best describes the algorithm's time complexity (and discuss it with the other members of the team).
b. Analyze the worst-case complexity in terms of Big-O, considering the effect of bin(n).count('1') on recursion depth.
c. Provide an example input n where the execution time significantly deviates from typical cases and explain why.
d. Determine whether the memoization (see memo) significantly reduces complexity in the average case.