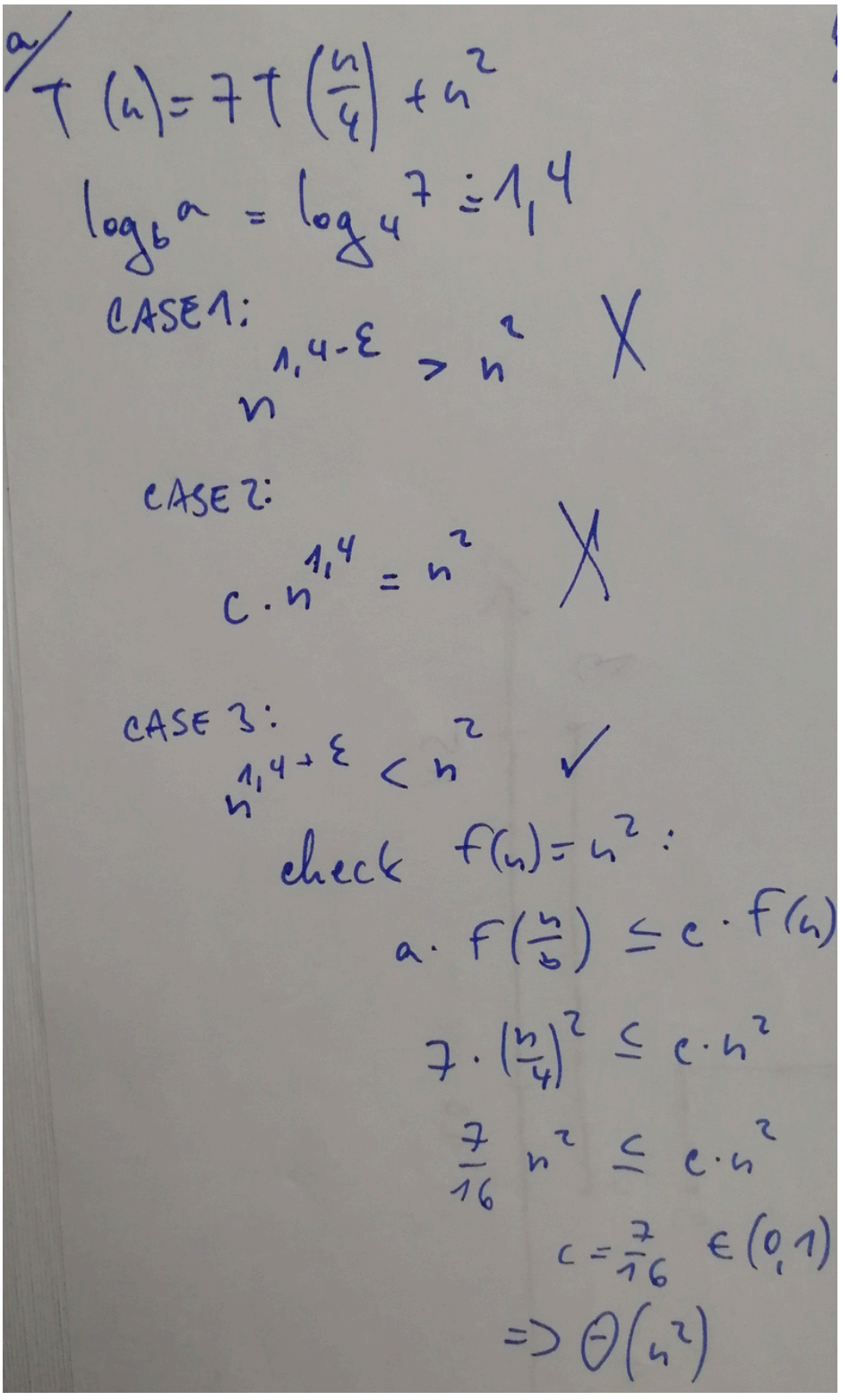
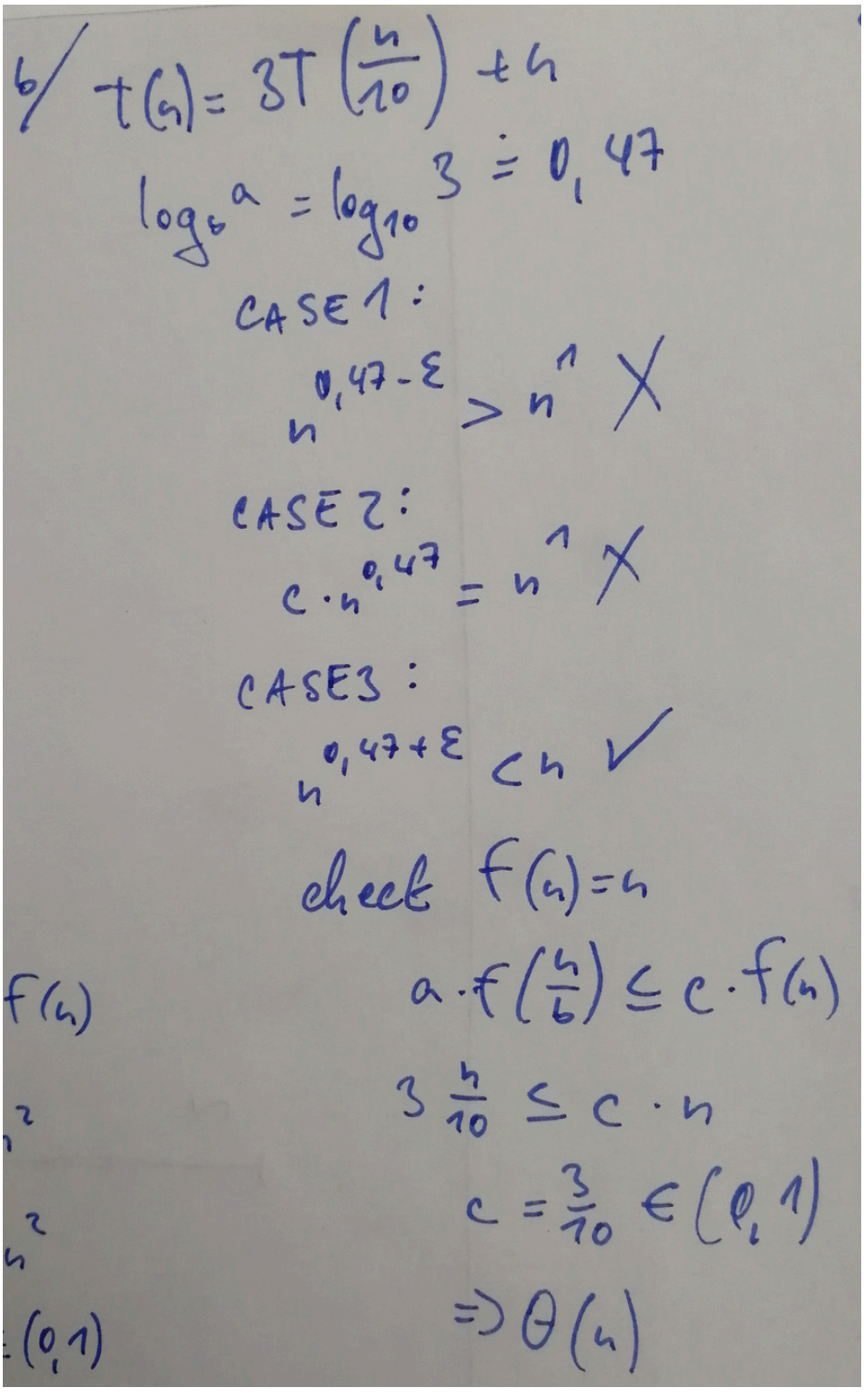
Jakub Vokoun, A23B0235P, 3 hours

1.

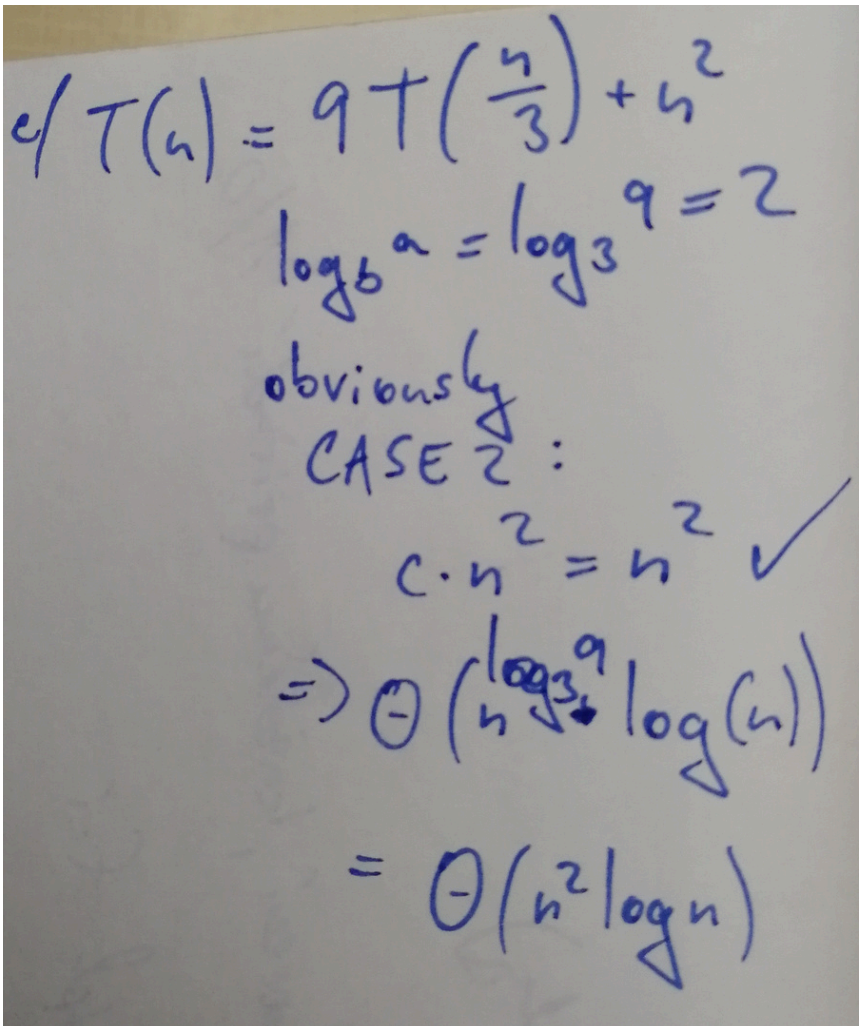
a.


a/
$T(n) = 7T\left(\frac{n}{4}\right) + n^2$
$\log_b a = \log_4 7 \doteq 1,4$
CASE 1:
$n^{1,4-\varepsilon} > n^2$ X
CASE 2:
$c \cdot n^{1,4} = n^2$ X
CASE 3:
$n^{1,4+\varepsilon} < n^2$ ✓
check $f(n) = n^2$:
a. $f\left(\frac{n}{b}\right) \leq c \cdot f(n)$
$7 \cdot \left(\frac{n}{4}\right)^2 \leq c \cdot n^2$
$\frac{7}{16} n^2 \leq c \cdot n^2$
$c = \frac{7}{16} \in (0,1)$
$\Rightarrow \Theta(n^2)$

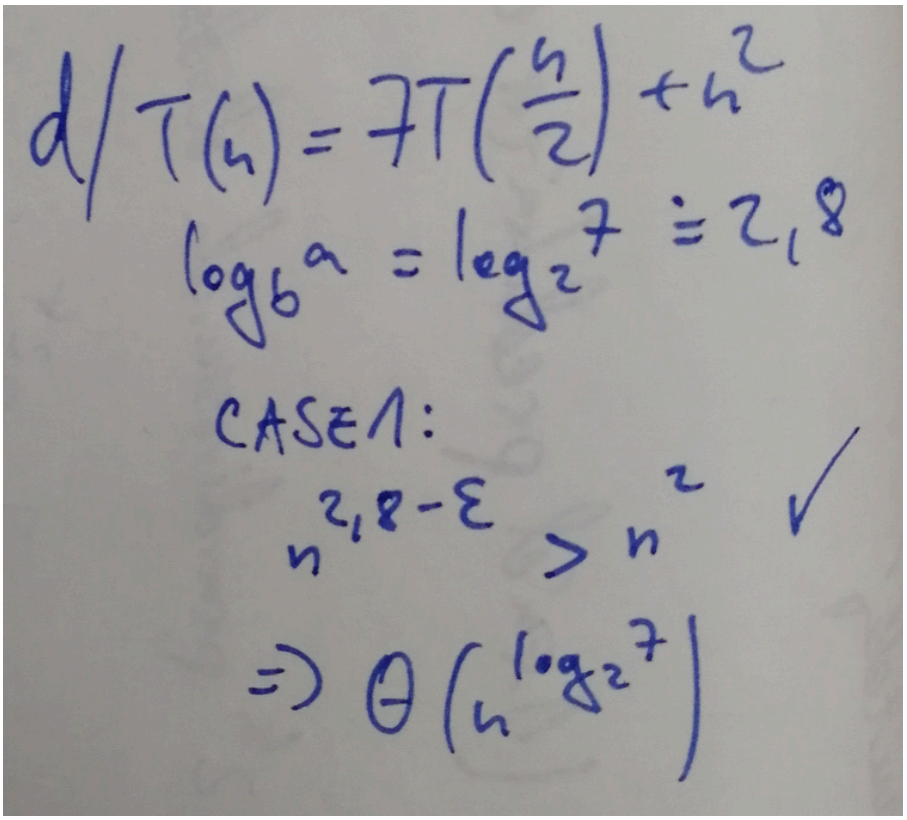
b.


b/ $T(n) = 3T\left(\frac{n}{10}\right) + n$
$\log_b a = \log_{10} 3 \doteq 0,47$
CASE 1:
$n^{0,47-\varepsilon} > n$ X
CASE 2:
$c \cdot n^{0,47} = n$ X
CASE 3:
$n^{0,47+\varepsilon} < n$ ✓
check $f(n) = n$
$a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$
$3\frac{n}{10} \leq c \cdot n$
$c = \frac{3}{10} \in (0,1)$
$\Rightarrow \Theta(n)$
$f(n)$
n^2
n^2
$\in (0,1)$

c.


c/ $T(n) = 9T\left(\frac{n}{3}\right) + n^2$
$\log_b a = \log_3 9 = 2$
obviously
CASE 2:
$c \cdot n^2 = n^2$ ✓
$\Rightarrow \Theta(n^{\log_3 9} \log(n))$
$= \Theta(n^2 \log n)$

d.


d/ $T(n) = 7T\left(\frac{n}{2}\right) + n^2$
$\log_b a = \log_2 7 \doteq 2,8$
CASE 1:
$n^{2,8-\varepsilon} > n^2$ ✓
$\Rightarrow \Theta(n^{\log_2 7})$

e.

e/ $T(n) = T(\sqrt{n}) + 2$

$T(n) = T(\sqrt[4]{n}) + 2 + 2$

$T(n) = T(\sqrt[8]{n}) + 2 + 2 + 2$

$\vdots$

$T(n) = T(\sqrt[2^k]{n}) + 2k$

$= T\left(\underbrace{n^{(\frac{1}{2})^k}}\right) + 2k$

$n^{(\frac{1}{2})^k} = 1$ //log

$\log n^{(\frac{1}{2})^k} = \log 1$

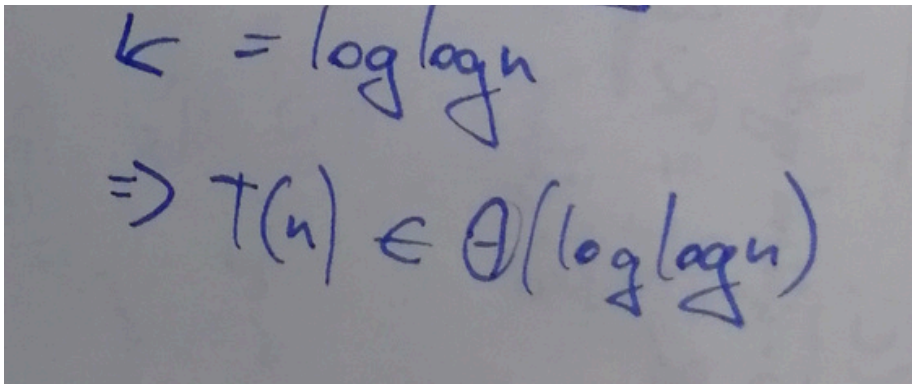
$(\frac{1}{2})^k \log n = \underbrace{\log 1}_{=0}$ $n > 0$

~~$(\frac{1}{2})^k = \frac{\log 1 = 0}{\log n}$~~

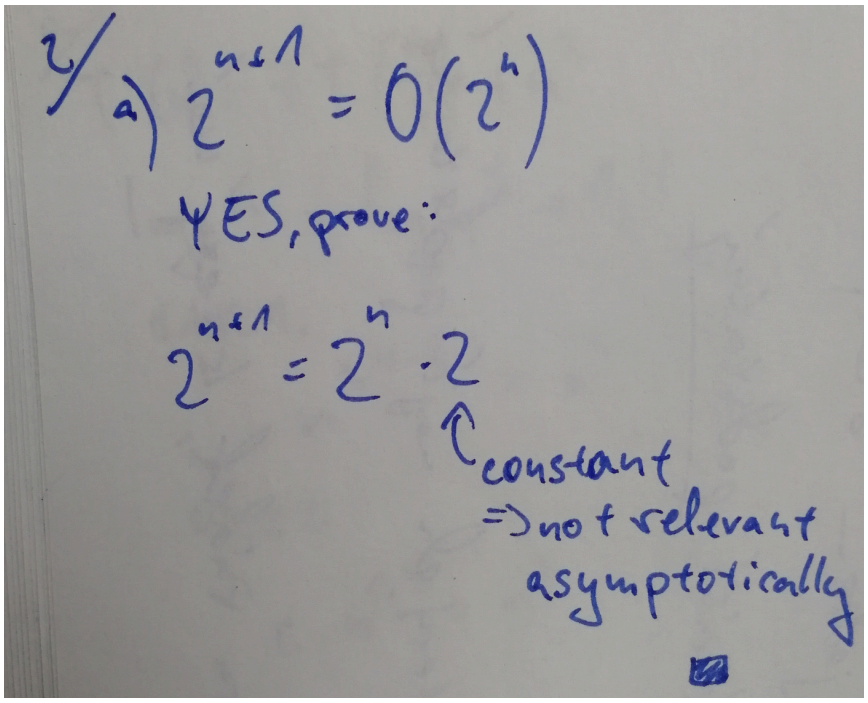
$(\frac{1}{2})^k \cdot \log n = 0$ $n > 0$

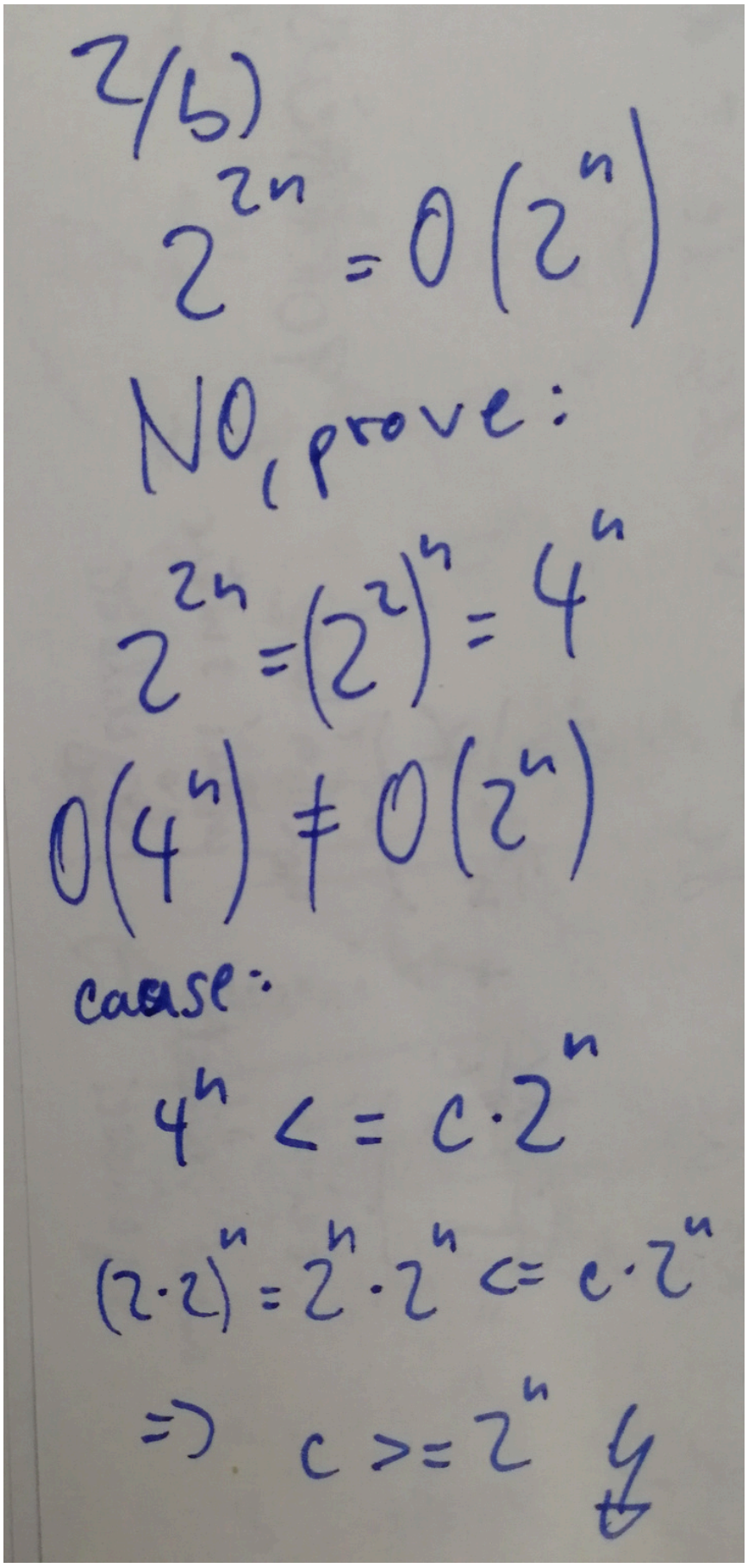
$\leftarrow \log_2 n$ $n \geq 1$

$2^k >> \log_2 n$ //log

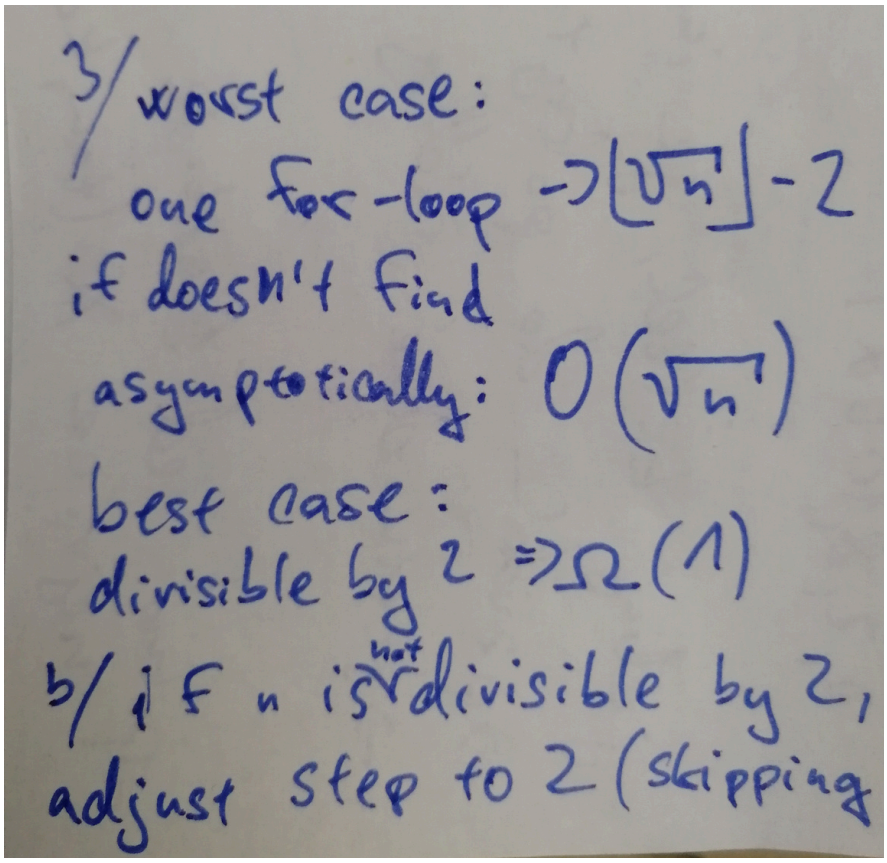

k = log log n
⇒ T(n) ∈ Θ(log log n)

2.


2/
a) 2^(n+1) = O(2^n)
YES, prove:
2^(n+1) = 2^n · 2
constant
⇒ not relevant
asymptotically


2/b)
$2^{2n} = O(2^n)$
NO, prove:
$2^{2n} = (2^2)^n = 4^n$
$O(4^n) \neq O(2^n)$
cause:
$4^n <= c \cdot 2^n$
$(2 \cdot 2)^n = 2^n \cdot 2^n <= c \cdot 2^n$
$=> c >= 2^n$ 4

3.



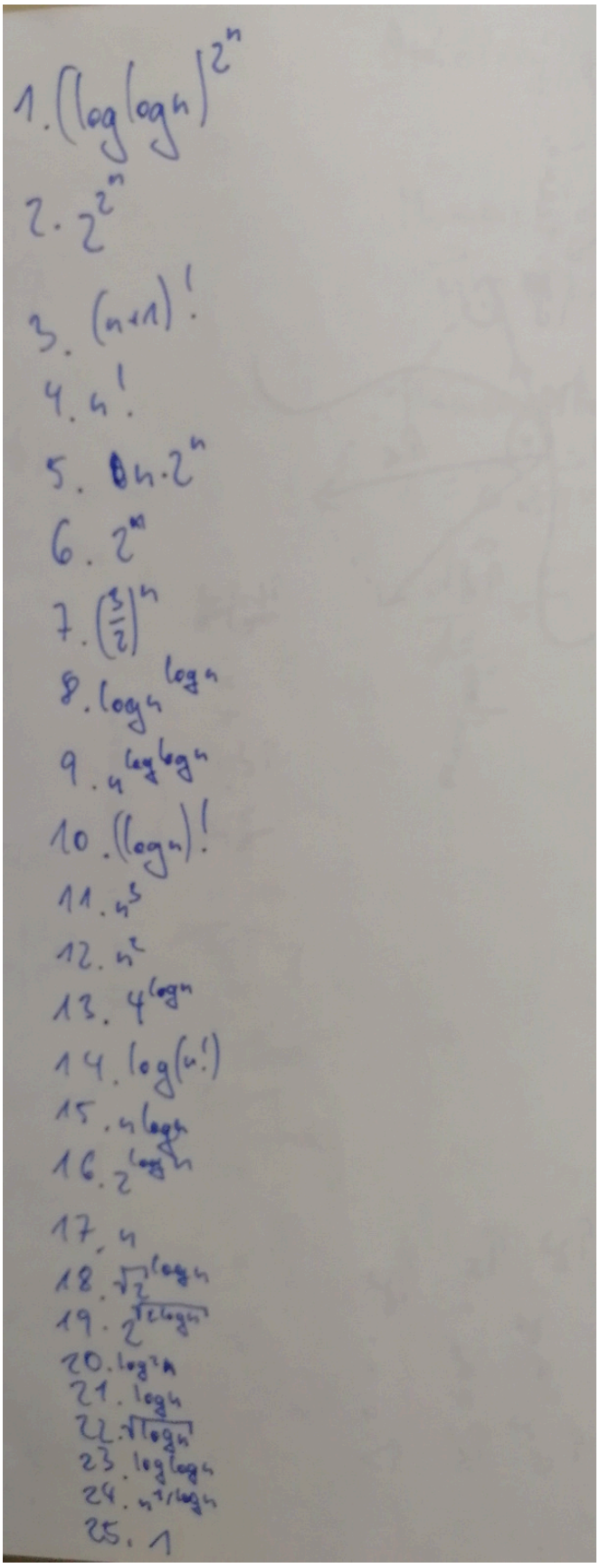
if n is not divisible by 2, adjust step to 2 (therefore skipping every even number)

could be extended to more number (e.g. multiples of 3,5,7) - but that wouldn't be that simple to implement, i would stick to step size *2, unless it's critical

4.

# n / complexity	log_(2)(n)	n	nlogn	n^2	n^3
8	3 ns	8 ns	24 ns	64 ns	512 ns
32	5 ns	32 ns	160 ns	1.024 µs	32.768 µs
64	6 ns	64 ns	384 ns	4.096 µs	262.144 µs
256	8 ns	256 ns	2.048 µs	65.536 µs	16.777216 ms
4096	12 ns	4.096 µs	49.152 µs	16.777216 ms	1.1453246122666667 min
16384	14 ns	16.384 µs	229.376 µs	268.435456 ms	1.2216795864177779 hour
65536	16 ns	65.536 µs	1.048576 ms	4.294967296 s	3.2578122304474073 day
1048576	20 ns	1.048576 ms	20.97152 ms	18.325193796266667 min	3.665933762613346 decade

5.



Block A

Bonus 1

a: (indices of array on the first line, array elements on the second line and found indices on the last line) -> (-1) means not found

e.g. the second column means:

- **the index of that column (element in array) is 1**
- **the element in array is 8**
- **the nearest smaller left-side neighbour is on the index 0**

```
[00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13, 14, 15]
[00, 08, 04, 12, 02, 10, 06, 14, 01, 09, 05, 13, 03, 11, 07, 15]
[-1, 00, 00, 02, 00, 04, 04, 06, 00, 08, 08, 10, 08, 12, 12, 14]
```

b:

example of implementation in python

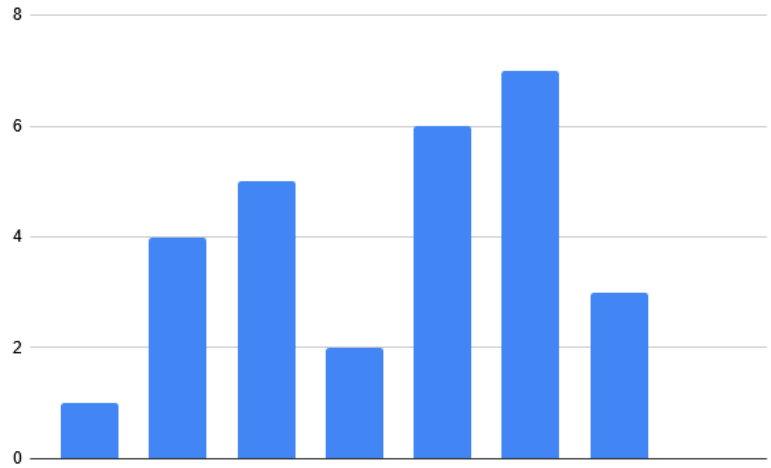
```python
def find_nearest_smaller_items_on_the_left(arr: list[int]) -> list[int]:
    res: list[int] = [-1 for _ in range(len(arr))]

    # go through the list once
    for i in range(1, len(arr)):
        # assume, the smallest item on the left is the left neighbour
        j: int = i - 1
        # if not, go to its (already calculated) smallest neighbour, until the condition is met
        while arr[j] >= arr[i] and j > 0:
            j = res[j]

        # if valid, save j, but if not, save -1 (not found)
        # j shouldn't be -1, because that means not found, but python interprets it as last item of array
        # the basic condition of being smaller should be met
        if j != -1 and arr[j] < arr[i]:
            res[i] = j
        else:
            res[i] = -1

    return res
```

The algorithm explained:

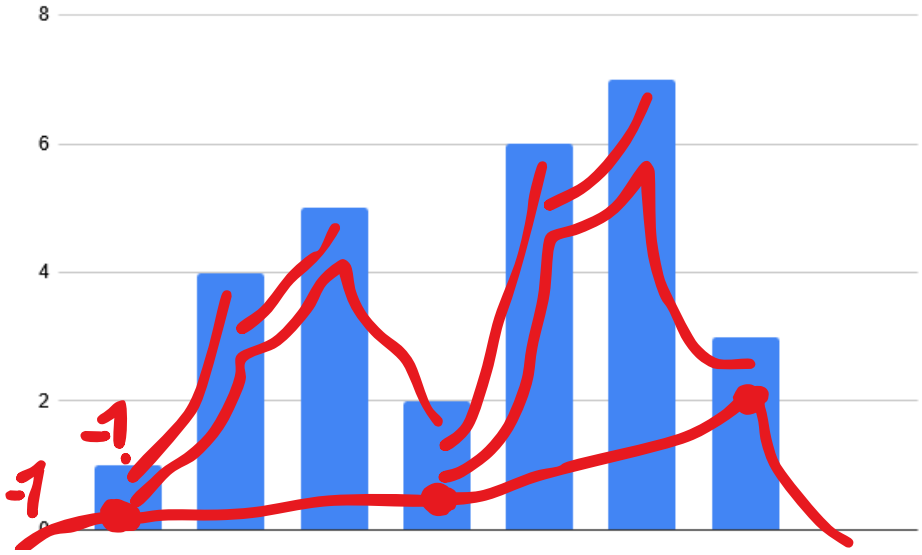
1. **Create result array, of the length of the array - there would be stored the indices found - and initialize to -1 (not found - wasn't specified in assignment).**
2. **Go through the array once, for every element could either happen:**
   a. **Is the first element:**
      Skip, it doesn't have any left neighbour.
   b. **Its direct left neighbour is smaller:**
      Save neighbour index to result array.
   c. **Its direct left neighbour isn't smaller:**
      Until you find smaller element, go to element smaller than the one you are currently on. In pseudocode:
      while (ARR[evaluated_element] > ARR[smaller_adept]):
      smaller_adept = RES[smaller_adept]
      (where ARR is the given array, and RES is the result array)
3. **Return result array.**

The complexity is of O(n), because in the loop could only happen:

a. **First element is clearly O(1).**
b. **Direct smaller neighbour is also clearly O(1).**
c. **The worst case is, if the array have ascending elements, and then one smaller e.g.: [1,2,3,9,0]. When evaluating 0, the algorithm would have to go through the whole array one by one. However, this cannot happen often, because the array can only be continuously ascending for so long. This means, that you can only have one go-through of array, the length of n. More common are partially ascending arrays, as example: [1,4,5,2,6,7,3,0] we can visualize as:**



where the algorithm must go through it like:



Here, the last element has visually the longest path, but it only goes through 3 elements, until the algorithm decides it hasn't left smaller neighbour.

In conclusion, the case when elements in array are sorted in ascending order is not a special case, because if we consider the average length of path in that array, the long paths of smaller elements after ascending parts of an array are averaged with many paths of length 1 in the ascending part. This means, that ultimately, this is of complexity n.

Bonus 2

example of implementation in Python:

```python
def factorial_products_equal(arr1: list[int], arr2: list[int]):
    prime_factors: list[list[int]] = [
        [0, 0, 0, 0],  # 0
        [0, 0, 0, 0],  # 1
        [1, 0, 0, 0],  # 2
        [1, 1, 0, 0],  # 3
        [3, 1, 0, 0],  # 4
        [3, 1, 1, 0],  # 5
        [4, 2, 1, 0],  # 6
        [4, 2, 1, 1],  # 7
        [7, 2, 1, 1],  # 8
        [7, 4, 1, 1],  # 9
    ]  # for each number, the count of prime factors 2,3,5,7

    arr1_prime_factors: list[int] = [0 for _ in range(4)]  # 2,3,5,7
    arr2_prime_factors: list[int] = [0 for _ in range(4)]  # 2,3,5,7

    # for each number in arr 1
    for i in range(len(arr1)):
        # add the prime factor counts
        for j in range(len(arr1_prime_factors)):
            arr1_prime_factors[j] += prime_factors[arr1[i]][j]

    # foreach in arr2 add the prime factor counts
    for i in range(len(arr2)):
        for j in range(len(arr2_prime_factors)):
            arr2_prime_factors[j] += prime_factors[arr2[i]][j]

    # check if both arrays have the same number of prime factors
    for i in range(len(arr1_prime_factors)):
        if arr1_prime_factors[i] != arr2_prime_factors[i]:
            return False

    return True
```

The algorithm:

We use the fact, that the integers range from 0 to 9, and that the factorial can be decomposed into prime factor multiplication. Meaning 4! = 4 * 3! = 2 * 2 * 3 * 2! = 2^3 * 3

The generic pseudocode:

Function Are_Equal(arr1, arr2):

arr1_primes, arr2_primes

for each element in array1:

x = prime factors of that element

arr1_primes.add(x)

for each element in array2:

```
        x = prime factors of that element
        arr2_primes.add(x)
    for each prime_factor in [2,3,5,7]:
        if arr1_primes[prime_factor] != arr2_primes[prime_factor]:
            return False
    return True
```

We go through both arrays, getting the total number of prime factors. Comparing this information instantly reveals if the arrays have equal factorial product, without the need of calculating the factorials.