exerxise sheet 01

exe. 1

a)

$$n = 3^j, j \in N$$

$$T(n) \leq 3 * T(\frac{n}{3}) + c * n * log_3(n)$$
$$T(3^j) \leq 3 * T(3^{j-1}) + c * n * log_3(3^j)$$

$$T(\frac{n}{3}) \leq 3 * T(\frac{n}{9}) + c * \frac{n}{3} * log_3(\frac{n}{3})$$
$$T(3^{j-1}) \leq 3 * T(3^{j-2}) + c * 3^{j-1} * log_3(3^{j-1})$$

$$\begin{aligned} T(n) &\leq 3 * (3 * T(\frac{n}{9}) + c * \frac{n}{3} * log_3(\frac{n}{3})) + c * n * log_3(n) \\ &\leq 3^2 T(\frac{n}{9}) + c * n * log_3(\frac{n}{3}) + c * n * log_3(n) \end{aligned}$$

$$\begin{aligned} \rightarrow n = 3^j \\ &\leq 3^2 T(3^{j-2}) + c * 3^j * log_3(3^{j-1}) + c * 3^j * log_3(3^j) \\ &\leq 3^2 T(3^{j-2}) + c * 3^j * (j-1) + c * 3^j * (j) \end{aligned}$$

$$\rightarrow \text{nach k Schritten}$$

$$T(3^j) \leq 3^k T(3^{j-k}) + c * \sum_{i=0}^{k-1} 3^{j-i} * (j-i)$$

$$\rightarrow \text{für k = j}$$

$$T(3^j) \leq 3^j T(1) + c * \sum_{i=0}^{j-1} 3^{j-i} * (j-i)$$

$$\rightarrow \text{da T(1)} <= \text{c}$$

$$T(3^j) \leq 3^j * c + c * \sum_{i=0}^{j-1} 3^{j-i} * (j-i)$$

$$\sum_{i=0}^{j-1} 3^{j-i} * (j-i) = 3^j * j + 3^{j-1}(j-1) + 3^{j-2}(j-2) + \dots$$

$$\rightarrow 3^j * j \text{ dominiert}$$
$$\begin{aligned} \rightarrow T(3^j) &\leq *3^j * c + c * 3^j * j \\ &\leq *n * c + c * n * log_3 n \end{aligned}$$

$$\rightarrow T(n) = O(n * logn) \rightarrow \text{Laufzeit O(n*logn)}$$

b)

Induktionsanfang:

$$\begin{aligned} n = 1 \rightarrow T(1) &\leq c \\ \rightarrow T(1) &\leq c' * 1 log_3(1) = 0, c' \geq c, c > 0 \end{aligned}$$

Induktionsvorraussetzung:

$$
\begin{aligned}
j = k, n = 3^j \\
T(3^k) &\leq c' * 3^k * log_3(3^k) = c' * 3^k * k \\
&\leq c' * n * log_3(n)
\end{aligned}
$$

Induktionsschritt:

$$
\begin{aligned}
j = k+1, n = 3^{k+1} \\
T(3^{k+1}) &\leq 3T(\frac{3^{k+1}}{3}) + c * 3^{k+1} * log_3 3^{k+1} \\
&\leq 3T(3^k) + c * 3^{k+1} * (k+1) \\
(IV) &\leq 3(c' * 3^k * k) + c * 3^{k+1} * (k+1) \\
&\leq c' * 3^{k+1} * k + c * 3^{k+1} * (k+1) \\
&\leq 3^{k+1} * (c' * k + c * (k+1)) \\
&\leq 3^{k+1} * (c' * k + c * k + c) \\
&\leq 3^{k+1} * ((c' + c) * k + c)
\end{aligned}
$$

$$
\begin{aligned}
\rightarrow ((c' + c) * k + c) &\leq c' * (k+1) \\
((c' + c) * k + c) &\leq c' * k + c' \\
c' * k + c * k + c &\leq c' * k + c' \\
\leftrightarrow c * k + c \leq c' &\rightarrow c \leq c'
\end{aligned}
$$

$$
\rightarrow T(3^{k+1}) \leq 3^{k+1} * c' * (k+1)
$$

exe. 2

```python
def equivalenz_test(card1, card2):
    # The Equivalence Test: Cards are considered equivalent if they are the
same
    return card1 == card2


def fraud_detection(cards, n, n0, k):
    # Base case: If there is only one card, return it with count 1
    if n == 1:
        return (False,{cards[0]: 1})

    # Divide the cards in half
    mid = n // 2
    left_cards = cards[:mid]
    right_cards = cards[mid:]

    # Recursive call for left and right halves
    left_counts = fraud_detection(left_cards, len(left_cards),n0, k)
    right_counts = fraud_detection(right_cards, len(right_cards),n0, k)

    # Combine the counts considering the equivalence test
    combined_counts = left_counts[1].copy()

    for right_card, right_count in right_counts[1].items():
        # Look in the left half for cards that are equivalent to the right card
```

```
        found_equivalent = False
        for left_card in combined_counts:
            if equivalenz_test(left_card, right_card):
                # If equivalent, sum the counts
                combined_counts[left_card] += right_count
                found_equivalent = True
                break

        # If no equivalent card was found, add the right card
        if not found_equivalent:
            combined_counts[right_card] = right_count
    # Check if any card appears more than n/k times
    for card, count in combined_counts.items():
        if count > n0/ k:
            return (True, {card: count})

    # Return the combined counts
    return (False,combined_counts)
```

Correctness:

1. **Divide and Conquer**: The algorithm recursively divides the set of bank cards into smaller subsets. This is a common approach for problems like this because it allows us to reduce the complexity of comparing each card to all others.
2. **Equivalence Testing**: For each division, the algorithm recursively processes the left and right halves of the card set. It uses the `equivalence_test` function to determine if two cards are equivalent. By combining results from the left and right subsets, the algorithm keeps track of how many cards are equivalent to each other.
3. **Correctness**: The correctness follows from the fact that the algorithm tests equivalence between all cards across both halves and tracks counts for equivalent cards. In the merging step, we ensure that counts of equivalent cards from both halves are added up. If any card's count exceeds $\frac{n}{k}$, it immediately returns `True` with the card in question.
4. **Base Case**: The base case ensures that when a single card is left, it is counted as 1, which is then used to build up the counts in higher recursive levels.
5. **Final Check**: After combining counts from the left and right halves, the algorithm checks if any card appears more than $\frac{n}{k}$ times and returns the result if true.

Time Complexity Analysis:

- **Divide and Conquer**: The algorithm splits the list of cards into two halves in each recursive call, which gives it a $\log n$ factor in the time complexity, similar to merge sort.
- **Equivalence Testing**: In each merge step, for each card in the right half, the algorithm compares it with all the cards in the left half to check for equivalence. This takes $O(n)$ time in the worst case for each level of recursion. Hence, at each level, the merging step takes $O(n)$ comparisons.
- **Overall Complexity**: Since there are $O(\log n)$ levels of recursion (due to halving the list at each step), and at each level, $O(n)$ comparisons are made for equivalence testing, the time complexity of the basic divide-and-conquer approach would be $O(n \log n)$.

However, since the problem allows k equivalence checks to find the set of cards that are equivalent to more than $\frac{n}{k}$, the total complexity will be $O(k \cdot n \cdot \log n)$. The k factor comes in because for each card, we may have to perform equivalence checks across k sets.

Algorithm Theory: Exercise Sheet 1
Kai Malaka (km324), Manuel Di Biase (md509)

Exercise 3

a).

```
FUNCTION maximum_area_rectangle( array )
    maximum_area = 0
    FOR i = 0 to array length - 1
        length = 0
        height = array[i]
        FOR j = i to array length - 1
            length = length + 1
            height = min{height, array[j]}
            maximum_area = max{maximum_area, length * height}
        END FOR
    END FOR
    RETURN maximum_area
END FUNCTION
```

b).

```
FUNCTION maximum_area_rectangle_intersecting_bar( array , bar )
    left = bar - 1
    right = bar + 1
    height = array[bar]
    maximum_area = height
    FOR length = 2 to array length
        IF right >= array length or (left >= 0 and array[left] > array[right]):
            height = min{height, array[left]}
            left = left - 1
        ELSE
            height = min{height, array[right]}
            right = right + 1
        END IF
        maximum_area = max{maximum_area, length * height}
    END FOR
    RETURN maximum_area
END FUNCTION
```

The running time of this function consists of three parts:

(1) Initialization:
The assignment of values to the variables left, right, height, and maximum_area takes a constant time. - c_1

(2) Amount of iterations in the main loop:
The loop always runs from 2 to the array length. For an array of size n, this corresponds to n-1 iterations.

(3) Code in the loop:
The condition takes a constant time to evaluate, because it is only using comparison operators. The code in both paths only uses a min function and one addition/subtraction which also takes a constant time to execute. After the condition is a max function, which also takes a constant time to evaluate. So the whole code block inside the loop takes a constant time to evaluate. - c_2

The running time of the function is $c_1 + (n-1) \cdot c_2$ which is a linear function, so it is in $\mathcal{O}(n)$

c).

```
FUNCTION maximum_area_rectangle(array: list[int]) -> int:
    IF length of array == 1:
        RETURN array[0]
    END IF
    RETURN max{
        maximum_area_rectangle( lower half of array ),
        maximum_area_rectangle_intersecting_bar(array, central array element),
        maximum_area_rectangle( upper half of array ) }
END FUNCTION
```

maximum_area_rectangle_intersecting_bar is the function defined above. If the array doesn't have a clear center element either one of the two central elements can be used.

Running timen $T(n)$:
Basis:
let $n = 1$.
The evaluation of the condition and the return statement happen in some constant time c_1.
$T(1) = c_1$

let $n \geq 2$.
The evaluation of the condition and the max function executes in some constant time c_2.
The evaluation of the maximum_area_rectangle_intersecting_bar is, as shown above in $\mathcal{O}(n)$
Lastly the maximum_area_rectangle function gets called twice with arrays of size $n/2$
$T(n) = 2 \cdot T(\frac{n}{2}) + c_2 + n \cdot c_3$

For this recurrence relation the master theorem states that $T(n) = \mathcal{O}(n \; log \; n)$