张文超 2022233388

Problem 1

a) False.

$f(n) = n \quad g(n) = n^2 \Rightarrow f(n) = O(g(n))$

However, $\log_2 f(n) = \lg n$

$\log_2 g(n) = 2\log_2 n$.

So, $\log_2 f(n) = O(\log_2 g(n))$ is not true.

b) False.

$2(n) = O(n)$

but $2^{2n} = 4^n \neq O(2^n)$

c) True

$f(n) = O(g(n)) \Rightarrow |f(n)| \leq C(g(n))$ for all $n \geq n_0$.

Squaring both sides, we get

$|f(n)|^2 \leq C^2 |g(n)|^2$ for all $n \geq n_0$.

Thus, $f(n)^2 = O(g(n)^2)$

Problem 2

$f_7 < f_8 < f_2 < f_4 < f_1 < f_3 < f_5 < f_6$

Problem 3

The best and ~~worse~~ worst case time complexity of selection sort is $O(N^2)$, this is because no matter whether the array is sorted or not, selection sort needs to ~~trac~~ traverse the entire array to find the smallest element and swap it.

Problem 4

The idea of this algorithm is to divide n cards to 2 halfs, and then recursely search for possible majority of elements in each half.

① If there is a majority element in both ~~sid~~ halves, and they are the same, then this is the majority element of the entire sequence.

② If there is a majority element in both halves, but they are different, then we need to recalculate their frequencies in the whole ~~seq~~ sequence. to see which one exceeds $\lfloor n/2 \rfloor$.

③ If only a half ~~of two kids~~ has a majority element, then verify it in the whole sequence.

④ If there is no majority element in either half, then there is no majority element in the whole sequence.

This algorithm use $O(n \log n)$ machine calls: each recursion requires $O(n)$ machine calls to compute the frequency, and the depth of the recursion is $O(\log \log n)$.

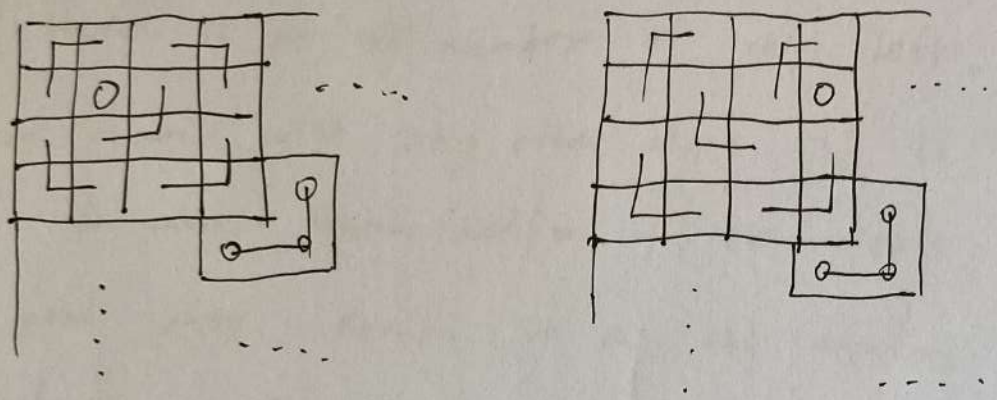
```
def find_majority_element(cards):
    if len(cards) == 1:
        return "Yes"
    # divide
    mid = len(cards) // 2
    left = cards[:mid]
    right = cards[mid:]
    # recurse
    left_result = find_majority_element(left)
    right_result = find_majority_element(right)
    # ①
    if left_result != "No" and
        right_result != "No" and
        machine(left[0], right[0]):
        return "Yes"
    # ②
    if left_result != "No" and
        right_result != "No":
        left_count = count_frequency(cards, left[0])
        right_count = count_frequency(cards, right[0])
        if left_count > mid or
            right_count > mid:
            return "Yes"
        else:
            return "No"
    # ③
    if left_result != "No":
        left_count = count_frequency(cards, left[0])
        if left_count > mid:
            return "Yes"
        else:
            return "No"
    if right_result != "No":
        right_count = count_frequency(cards, right[0])
        if right_count > mid:
            return "Yes"
        else
            return "No"
    # ④
    return "No"

def count_frequency(cards, card):
    frequency = 0
    for c in cards:
        if machine(c, card):
            frequency += 1
    return frequency
```

Problem 5

For a nxn floor, n is a power of 2.

Divide horizontally and vertically until each peace is a 4x4 rug. Where the billboard is placed, lay corresponding carpet numbers as required.



Then three adjacent carpet areas of correspondingind size place the imagainary billboard seat in the center of the current whole, which can be covered by a small carpet.

Problem 6.

The algorithm is based on that there are more good chips than bad chips in the batch.

Divide the chips into pairs and test each pair using the machine.

If both chips reports good, then they are either both good or both bad. Keep one chip from each pair and discard another.

If one chip reports good and the other reports bad, then they are diffent. Discard both chips.

Repeat this process until only one chip ~~or~~ remains or there is an odd number of chips left.

If there is only one chip left, then it must be a good chip. Return it as the answer.

If there is an odd number of chips left, then test one of them with any other chip. If it reports good for more than half of the tests, then it is a good chip, Return it as the answer. Otherwise, discard it and ~~con~~ continue with the remaining even number of chips.

The algorithm has a time complexity of $O(n)$.