CSE321

Introduction to Algorithm Design and Analysis

Homework #3

Oğuzhan Ağkuş

161044003

1- I designed a decrease and conquer algoritm to alternate boxes in black-white pattern. Its pseudocode is like that:

```
algorithm_1 (row):
    n = len(row)
    for i in range(2, n/2, 2):        # start index, center of row, increment by 2
        row[i], row[n-i+1] = row[n-i+1], row[i]   → swap
```

★ row is an array which contains n times B and n times W

→ Explanation with an example:

(n=5) row = B B B B B W W W W W → indexes 1 2 3 4 5 6 7 8 9 10 — 2 swap

(n=6) row = B B B B B B W W W W W W → indexes 1 2 3 4 5 6 7 8 9 10 11 12 — 3 swap

- Start from 2. index because first element should be black.
- Swap it with (n-i+1). element. Its actually 2. element from end.
- Then increment the index by 2.
- Repeat it.

★ swap_count = $\lfloor \frac{n}{2} \rfloor$

★ After every step, unordered list size is decreasing. Because we have already ordered 4, i element where i is swap count. It is decrease by constant technique.

- Complexity Analysis

First n boxes in a row are always black and left are white.

So n whatever it is, to alternate 2n boxes we must do $\frac{n}{2}$ swap operation.

Its complexity is $O(\lfloor \frac{n}{2} \rfloor)$. Ignore the co-efficient $\frac{1}{2}$, $O(n)$.

Average and worst cases cannot be bigger than $O(n)$.

→ So average case = $O(n)$
worst case = $O(n)$

We have a special situation which is n=1. If n=1, row will be "BW" which is already ordered. No need to swap. Its $O(1)$

→ Best case = $O(1)$

Note: You didnot want to implement this algorithm from us but I wanted to do it. I added it to .py file and wrote a test function for it.

2 - I desinged two decrose and conquer algorithm for fake coin problem. Because it is not clear that if we have information about fake coin's weight. It can be lighter or heavier.

Firstly I assume that fake coin is lighter. According that the pseudo code is like that:

```
algorithm_2 (coins):
    n= len(coins)
    if (n%2 != 0): ⟶ If coin count is odd, choose a coin
        coins.pop()

    left = coins[:n//2]  ) → Divide into two equal count
    right = coins [n//2:]

    if (left==right): ⟶ Popped coin is fake.
        print("popped coin is fake")

    else if (left < right) → Left is lighter, fake coin should be there.
        algorithm_2 (left)
    else if (right<left) → Right is lighter, fake coin should be there.
        algorithm_2(right)
```

Recursive calls

In each step, we will sure that half of n coins is not fake.

We decrease our search scope. For example, we have 16 coins. Divide by two, 8-8. One side is lighter. Ignore the other 8 coins. Now we have 8 coin. Repeat same steps. 4-4, → 2-2 → 1-1 → light one is fake.

*We decrease our input by variable $\frac{n}{2}$. So it is an decrease and conquer algorithm.

- Complexity Analysis

Best case → When coin count is odd, we seperate a coin. If weightbridge is balanced, the coin we seperated is fake. We can find it at one try. So it is O(1)

n=4 → 2-2 (1 1) } 2 steps

n=8 → 4-4 / 2-2 (1 1) } 3 steps

n=16 → 8-8 / 4-4 / 2 2 / (1 1) } 4 steps

n	step
4	2
8	3
16	4

} $\log_2(n)$ = step count

Average and worst case complexity = O(log n)

Note: I explain this method but I have implemented two version to solve different situations.
- Fake coin is lighter
- Fake coin can be lighter or heavier.

Second situation's complexity is not same as this one. Its O(n).

I wrote test cases for this implementations.

3- I implemented insertion and quick sort functions. Also I wrote a random array generator which return an array with size n. I tested my algorithms by using these random arrays.

My functions returns the swap counts.

Example array → [54, 15, 67, 45]

Insertion Sort

54, 15, 67, 45

15, 54, 67, 45

15, 54, 67, 45

15, 54, 45, 67

15, 45, 54, 67 ✓ Sorted

★ Decrease and conquer algorithm

Swap count = 3

Quick Sort

54, 15, 67, 45 (pivot)

15, 54, 67, 45

15, 45, 67, 54 (Sort)

15, 45, 67, 54

15, 45, 54, 67 ✓ Sorted

★ Divide and conquer algorithm

Swap count = 3

When I examined my test results when n is increasing insertion sort function return a swap count much more bigger than quick sort. So quick sort does less operations and it is faster than insertion sort. If n is smaller they return close result as like above example.

	Insertion sort	Quick sort
Best case	$O(n)$	$O(n \log n)$
Average case	$O(n^2)$	$O(n \log n)$
Worst case	$O(n^2)$	$O(n^2)$

- Best cases occurs when arrays are already sorted in increasing order.
- Worst cases occurs when arrays are sorted in decreasing order.

4- If we want to find median of an array, we need to sort this array. Because median's definition is express over a sorted array.

You wanted to a decrease and conquer algorithm to solve this problem.
Then I noticed that insertion sort is decrease and conquer algorithm from previous question.

After sorting the array, we need to get middlest element of array. Its complexity $O(1)$

Insertion sort's complexity's at best, average, and worst case is $O(n)$, $O(n^2)$, $O(n^2)$ accordingly.

So complexity of this algorithm →
best → $O(n) + O(1) = O(n^2)$
average → $O(n^2) + O(1) = O(n^2)$
worst → $O(n^2) + O(1) = O(n^2)$

I implemented this algorithm and wrote test cases for it.

5- I've implement an algorithm. It contains some helper functions, like calculating multiplication of an array's element, and finding lower bound for sum of elements. Their complexity is $O(n)$. ✶

Main part is a recursive function. Like $f(n) = 2f(n-1) + k$, where k is constant.
It's complexity is $O(2^n)$. Because it finds all possible sub-arrays. If we have n element of list it can have 2^n sub-array.

✶ step is done each time. So worst case is $\underline{O(n \cdot 2^n)}$