Ahmed Semih Ozmekik

Question 1

1. Rearranging black and white boxes. We can solve this problem with **decrease-and-conquer** algorithm, i.e. with **decrease-by-a-constant** algorithm. Here is the pseudo code of our algorithm:

Algorithm 1 Arrangement of boxes algorithm.

$list \leftarrow [black_0, black_1, ..., black_n, white_0, white_1, ..., white_n]$
$first_i \leftarrow 0$
$last_i \leftarrow 2n - 1$
procedure ARRANGEBOX($list$, $first_i$, $last_i$)
    if $first_i >= last_i$ **then return**
    end if
    swap $list[first_i]$ **with** $list[last_i]$.
    $first_i \leftarrow first_i + 2$. ▷ Decrease 2 from top
    $last_i \leftarrow last_i - 2$. ▷ Decrease 2 from bottom
    ARRANGEBOX($list, first_i, last_i$) ▷ Call Recursion decreased-by-4
end procedure

2. Implementation of the algorithm.

Listing 1: Arrangement of boxes algorithm implementation.

```
def arrange_box(lst, first_idx, last_idx):
    if first_idx >= last_idx:
        return
    swap(lst, first_idx, last_idx)  # swap first and last item of given list
    arrange_box(lst, first_idx + 2, last_idx - 2)
```

3. Analysis of complexities.

Best Case: Our box array is identified as it can only have n number of black boxes in first and then remaining n number of boxes are white. We have a strict rule for the input. Since this is the case, we don't have any flexibility for the input to make the difference between any kind of cases. Meaning, there is no such thing as best case, or etc. Therefore, our analysis will cover all of the cases, being best, worst, and the average case. Function for our algorithm:
$T(n) = T(n-4) + 1$ and $T(0) = 1$

$$T(n) = T(n-4) + 1 \quad (1)$$
$$T(n) = T(n-8) + 1 + 1 \quad (2)$$
$$T(n) = T(n-12) + 1 + 1 + 1 \quad (3)$$
$$... = ... \quad (4)$$
$$T(n) = T(0) + n/4 \quad (5)$$
$$T(n) = n/4 \quad (6)$$

Hence, for size n, $T_{best}(n) \in \mathcal{O}(n)$.

Worst Case: $T_{worst}(n) \in \mathcal{O}(n)$.

Average Case: $T_{avg}(n) \in \mathcal{O}(n)$.

Question 2

1. Fake coin problem. We can do this by using **decrease-by-factor-2** algorithm. But in that case we need to know in advance, if the fake coin will be lighter or heavier than the normal coins. Therefore we can use weighbridge regarding to that knowledge. Since we don't know that we can only go with **decrease-by-factor-3** algorithm. With this way we can ignore if it's heavier or lighter but we can focus the coin list having a different weight other than two lists. Here is the algorithm:

Algorithm 2 Finding the counterfeit coin algorithm.

```
1: procedure findFakeCoin(list)
2:     if list has 3 coins then
3:         return the coin having different weight among them.
4:     end if
5:     divide list to 3 list: A, B, C
6:     if A = B then                          ▷ Weighbridge is used here.
7:         findFakeCoin(C)                    ▷ Find the different one among them.
8:     else if A = C then
9:         findFakeCoin(B)
10:    else then
11:        findFakeCoin(A)
12: end procedure
```

But with this way we have an assumption that is total coin number must be $n = 3^m$.

2. Analysis of complexities.

   Best Case: In the best case scenario, 'til it's found fake coin always remains in the C. We are saved from only more check. Which makes our recurrence relation as: $T(n) = T(\frac{n}{3}) + 2$. Using the master theorem: $T_{best}(n) \in \mathcal{O}(logn)$.

   Worst Case: In the worst case scenario, 'til it's found fake coin always remains in the A. Which makes our recurrence relation as: $T(n) = T(\frac{n}{3}) + 3$. Using the master theorem: $T_{worst}(n) \in \mathcal{O}(logn)$.

   Average Case: $T_{best}(n) = T_{worst}(n) \implies T_{avg} \in \mathcal{O}(logn)$.

Question 3

1. Implementation of Insertion Sort and Quick Sort.

```
1  def insertion_sort(lst):
2      for i in range(1, len(lst)):
3          item = lst[i]
4          sorted_idx = i - 1
5          while sorted_idx >= 0 and lst[sorted_idx] > item:
6              lst[sorted_idx + 1] = lst[sorted_idx]  # swap operation
                   performed
7              sorted_idx -= 1
8          lst[sorted_idx + 1] = item
```

```
1  def quick_sort(lst, low_idx, high_idx):
2      if low_idx < high_idx:
3          partition_idx = partition(lst, low_idx, high_idx)
4          quick_sort(lst, low_idx, partition_idx - 1)
5          quick_sort(lst, partition_idx + 1, high_idx)
6
7
8  def partition(lst, low_idx, high_idx):
9      smaller_idx = low_idx - 1
10     pivot = lst[high_idx]
11     for j in range(low_idx, high_idx):
12         if lst[j] < pivot:
13             smaller_idx = smaller_idx + 1
14             swap(lst, smaller_idx, j)
15     swap(lst, smaller_idx + 1, high_idx)
16     return smaller_idx + 1
```

2. **Average Case**:
Let's analyze insertion sort:

$$E[I_{i,j}] = \frac{1}{2} \tag{1}$$
$$E[I] = \sum_{i<j} E[I_{i,j}] \tag{2}$$
$$= \sum_{i<j} \frac{1}{2} \tag{3}$$
$$= \frac{1}{2}.\binom{n}{2} \tag{4}$$

Hence, $T(n) \in \mathcal{O}(n^2)$

Let's analyze quick sort. After the do the partitioning, we recursively call our algorithm on two partitioned lists. Regarding to Hoare Partition algorithm, Since i is between 0 to $n-1$, average value of $T(i)$ is:

$$E(T(i)) = \frac{1}{n}\sum_{j=0}^{n-1} T(j) \tag{5}$$
$$E(T(n-i)) = E(T(i)) \therefore T(n) = \frac{2}{n}\left(\sum_{j=0}^{n-1} T(j)\right) + c.n \tag{6}$$
$$n.T(n) = 2.\left(\sum_{j=0}^{n-1} T(j)\right) + n^2 \qquad \text{Multiply by } n \tag{7}$$
$$(n-1).T(n-1) = 2.\left(\sum_{j=0}^{n-1} T(j)\right) + (n-1)^2 \qquad \text{put } n-1 \text{ for } n \tag{8}$$
$$n.T(n) = (n+1).T(n-1) + 2.c.n \qquad \text{substract 4 from 5} \tag{9}$$

Solving this recurrence relation:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1} \tag{10}$$
$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n} \tag{11}$$
$$... = ... \tag{12}$$

Hence, we get:

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{j=3}^{n+1}\frac{1}{j} \tag{13}$$

$$\sum_{j=3}^{n+1}\frac{1}{j} \to \ln n + \gamma \therefore \frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c.\ln n + 2c\gamma \tag{14}$$

Hence, $T(n) \in \mathcal{O}(n \log n)$

3. Comparative analysis of both sorting algorithms: Theoretically quick sort is better, since $T_{qui}(n) \in \mathcal{O}(n \log n)$ and $T_{ins}(n)\mathcal{O}(n^2)$. And with experiments, I have added a counter in python code to a bunch of relevant places to see actually which one is better in practically:

```
1 def swap(lst, idx1, idx2, count=0):
2     count += 1
3     temp = lst[idx1]
4     lst[idx1] = lst[idx2]
5     lst[idx2] = temp
```

If we are to count the number of swap operations in the sorting algorithms respectively, insertion sort performs more swap operations:

```
Same arrays sorted via Insertion Sort, Count : 20, 36, ...
Same arrays sorted via Quick Sort, Count : 6, 13, ...
```

Similar patterns observed during the experiments. Therefore we can say, quick sort is both better in theory and practice.

Question 4

1. We can do this by sorting and taking the middle element, but assuming we are using merge sort we can only get $T(n) \in \mathcal{O}(logn)$ from that. But using the *quickselect* algorithm, we can get k_{th} element, i.e. we can directly get the median from it. Therefore, we used *quickselect* algorithm which is **decrease-and-conquer** to find the median. Pseudo code of finding the median of an unsorted array:

Algorithm 3 Finding the median algorithm.

```
1: procedure FINDMEDIAN(list)
2:     if list has odd number of items then
3:         return QUICKSELECT(lst, len(lst)/2)
4:     else
5:         return (QUICKSELECT(lst, len(lst)/2 − 1) + QUICKSELECT(lst, len(lst)/2))/2
6:     end if
7: end procedure
```

Algorithm 4 Quickselect algorithm.

```
1: procedure QUICKSELECT(list, k)
2:     if list has one item then
3:         return lst[0]
4:     pivot ← random item among the list
5:     smallers ← items smaller than pivot in list
6:     highers ← items higher than pivot in list
7:     pivots ← items equal to pivot in list
8:     if k < len(smallers) then
9:         return QUICKSELECT(smallers, k)
10:    else if k < len(smallers) + len(pivots)
11:        return pivots[0]                          ▷ Found the kth item.
12:    else
13:        return QUICKSELECT(highers, k − len(smallers) − len(pivots))
14: end procedure
```

2. Implementation.

```python
def find_median(lst):
    if len(lst) % 2 == 1:  # odd number of items
        return quick_select(lst, len(lst) / 2)
    else:
        return (quick_select(lst, len(lst) / 2 - 1) + quick_select(lst, len(
            lst) / 2)) / 2

def quick_select(lst, k):
    if len(lst) == 1:
        return lst[0]

    pivot = random.choice(lst)  # picks a pivot among the items
    smallers = [item for item in lst if item < pivot]
    highers = [item for item in lst if item > pivot]
```

```
    pivots = [item for item in lst if item == pivot]

    if k < len(smallers):
        return quick_select(smallers, k)
    elif k < len(smallers) + len(pivots):
        return pivots[0]
    else:
        return quick_select(highers, k - len(smallers) - len(pivots))
```

3. **Worst Case**: Our worst case is when the array length is even. Because in that case, *quickSelect* called twice. And for worst case to happen in *quickSelect* array must be sorted. In that case since partitioning done recursively, it will partition n times and iterate n times, meaning it will be n^2. Which can be written as:

$$T(n) = 2 \times T_{quickselect}(n/2) + 1 \tag{1}$$
$$T(n) = 2 \times n^2 \tag{2}$$
$$T(n) = n^2 \tag{3}$$

   Hence, $T(n) \in \mathcal{O}(n^2)$

Question 5

1. Finding the optimal sub array algorithm with **exhaustive search**. Our algorithm creates all subsets as B and checks if it is better than current optimal subset. Since it's **exhaustive search** it will generate all subsets of sets and pick optimal one among them.

2. Implementation.

```
def find_optimal_sub(lst, index, sub_lst, opt_sub):
    if index == len(lst):
        if len(sub_lst) != 0:
            if sum(sub_lst) >= find_sum(lst):
                if len(opt_sub) == 0 or find_mult(sub_lst) < find_mult(
                    opt_sub):
                    opt_sub = sub_lst
    else:
        opt_sub = find_optimal_sub(lst, index + 1, sub_lst, opt_sub)
        opt_sub = find_optimal_sub(lst, index + 1, sub_lst + [lst[index]],
            opt_sub)
    return opt_sub

def find_sum(A):
```

```
    return (max(A) + min(A)) * (len(A) / 4)

def find_mult(A):
    mult = 1
    for el in A:
        mult *= el
    return mult
```

Algorithm 5 Finding the optimal sub-array algorithm.

```
index ← 0
B ← [ ]
optimalB ← [ ]
procedure FINDOPTIMALSUB(A, index, B, optimalB)
    if index = len(A) then                                  ▷ Leaf, reached to B
        if sum(B) >= (max(A) + min(A))× len(A)/4 then
            if B is optimal than optimalB then
                optimalB ← B
    else
        optimalB ← FINDOPTIMALSUB(A, index + 1, B, optimalB)
        B ← B + [A[index]]
        optimalB ← FINDOPTIMALSUB(A, index + 1, B, optimalB)
    end if
    return optimalB
end procedure
```

Worst Case: Since we stated, it runs always in worst case. With *sum* operation we have n times run in each recursion:

$$T(n) = 2 \times T(n-1) + n \tag{1}$$
$$T(n) = 2^2 \times T(n-1) + 2(n-1) + n \tag{2}$$
$$\ldots = \ldots \tag{3}$$
$$2^{n-1}T(n) = T(0) \times 2^n + 2^{n-1} \tag{4}$$
$$T(n) = T(0) \times 2^n + \sum_{i=0}^{n}(n-i) \times 2^i \tag{5}$$
$$T(m) = 2^n + n \times (2^{n+1} - 1) - 2(1 - 2^n + n2^n) \tag{6}$$
$$T(n) = n \times 2^n \tag{7}$$

Hence, $T(n) \in \mathcal{O}(n2^n)$.