Yusuf Patoglu 151044027

CSE321 Introduction to Algorithm Design – HW3

1-) Box Problem

In box problem we have pre-defined box position and box count which is 2n. Simply what problem tells us there are 2n boxes and n boxes ordered in n black and n white in order.

In order to solve this problem we need to simulate this with an array and fill the array with `b` and `w` which is black and white. After filling array with `b` and `w` we can start to create ‘b` ‘w` ‘b` ‘b` ‘w` ‘b`... order.

```
def boxProblem(boxCount):
    boxList = list();
    for x in range(boxCount // 2):
        boxList.append('b')
    for y in range(boxCount // 2):
        boxList.append('w')
```

Filling whole array with required order.

In order to solve this problem with Decrease and Conquer, I decided to implement Selection-Sort like logic to this. As we know Selection-Sort is decreases problem and finds the solution, in our version we’ll have two pivots, first one starts at first index and assumes that this is the sorted part. Second pivot is the middle index of list. This cursor will help us to get white boxes properly.

As you can see here pivot_1 increases by 2 and pivot1 increases by 1.

Because pivot 1 is the one which keeps our assumed sort part like:
(‘b’), (‘b’ ‘w’ ‘b’), (‘b’ ‘w’ ‘b’ ‘w’ ‘b)..

```
pivot_1 = 1
pivot_2 = boxCount // 2
for i in range(boxCount // 2, len(boxList)):
    swapCount += 1
    swapBoxes(boxList, pivot_1, pivot_2)
    pivot_1 += 2
    pivot_2 += 1
```

For the sake of clarity we can check box’ position for each iteration like:

```
root@kali:~/Python-3.6.5# python3.6 part1.py
Input size of boxes:  6
Before swapping boxes:  ['b', 'b', 'b', 'w', 'w', 'w']
['b', 'w', 'b', 'b', 'w', 'w']
['b', 'w', 'b', 'w', 'b', 'w']
['b', 'w', 'b', 'w', 'b', 'w']
After swapping boxes:  ['b', 'w', 'b', 'w', 'b', 'w']
Number of swaps:  3
```

Since we’re incrementing first pivot by two till end of box count we can say that for n boxes the required operations will be n / 2 and for our problem there are 2n boxes so the **number of swaps will be n in.** Since this problem is predefined in terms of input sequence

and input value, for my design of algorithm the worst,average and best case will be same, Θ(n).

2-) Fake Coin Problem

In Fake Coin Problem there are n coins and one of them is fake. We need to find the fake coin with Decrease and Conquer Algorithm. In my design standart coins have value of 5 and fake coin has value of 3. You can think they're 5 gr and 3 gr. In order to simulate fake coin problem we need to fill whole list with standart coins and place fake coin in random place. This implemented as follows:

```
#Assume that standart coin is 5 gr and fake one is 3 gr.
def fakeCoin(coinNumber):
    STANDART_COUNT = 5
    FAKE_COIN = 3

    #First create whole list with n standart coins.

    coinList = list();
    for x in range(coinNumber):
        coinList.append(STANDART_COUNT)

    #Then place the fake coin in random location.
    randomIndex = random.randrange(coinNumber)
    coinList[randomIndex] = FAKE_COIN

    print("Coin list is created as: ", coinList)
```

randrange function used for creating random index

After creating single list with coins we can think about solution. Applying a logic like Binary Search works here but there are better solution: "Dividing the coins by 3". In order to achieve that we need to write generalized code because not always the coin count will be divided by 3 without a remainder. Therefore our rules follows as:

while(Coin Count is equals to one in any of three sublists)
- *if coins % 3 = 0*
  - *then simply seperate them by 3.*
- *if coins %3 = 1*
  - *seperate them into 3 pieces again but one of them will have extra coin.*
- *if coins %3 = 2*
  - *seperate them into 3 pieces. Two seperated parts will extra one coin and other one won't get it.*

In order to divide list by three sub-lists, I calculated three index values for seperating lists as follows:

```
while(len(coinList) > 1):
    startPoint1 = 0
    startPoint2 = math.ceil(coinNumber / 3)
    startPoint3 = math.ceil(2 * coinNumber / 3)
    #Split list into 3 sublist.
    coinList1 = coinList[startPoint1 : startPoint2]
    coinList2 = coinList[startPoint2 : startPoint3]
    coinList3 = coinList[startPoint3 : len(coinList)]
    #Split evenly if the mod 3 is 0.
    if coinNumber % 3 == 0:
    #Split n/3 n/3 n/3 + 1 if mod 3 is 1.
    elif coinNumber % 3 == 1:
    #Split n/3 n/3 + 1 n/3 + 1 if mod 3 is 2.
    elif coinNumber % 3 == 2:
```

I implemented this iterative method for reducing space overhead on stack. Here, there are calculations for each index as starting points. There are also three sublists named

with coin lists. Inside of if functions there are necessary calculations for dividing the list.
(Note that I didn't show whole code block inside of if conditions for sake of clarity.

Since we're dividing 3 sub-problems we'll have logn (base of 3) *time complexity.* Because for n coins we'll divide the sub-problems like n, n/3, n/9, n/27, ... where x = 3 ^ n. Therefore complexity will be logn(base3).

While doing this seperation also we need to check total size of each sublist. For example let's think about we have n coins which is divisible by 3 without a remainder. Then we can seperate it into n/3 pieces. We'll need to iterate whole sublists to get total weight value. So for our example sum will be (n / 3) + (n / 3)+ (n / 3) +(n / 9) + (n / 9) + (n / 9) ... it will be O(n) for calculating sublists size. Because soon or later we need to iterate every element of list once. At last if we don't think about getting summation of lists it will be ***O(logn)*** but if we mind about getting each sublist's sum it will be ***O(logn) + O(n) = O(n).***

3-) Quick Sort and Insertion Sort

Before comparing these two sort techniques I will talk about main idea behind those sorts and explain how did I implement them in Python. Also I will find each of their best case complexities. Let's start with Insertion Sort:

Insertion Sort is a decrease and conquer sorting algorithm. It's a sort technique that divides the working list into two pieces, sorted piece and unsorted piece. Every iteration we pick up from unsorted part and insert the element in sorted part(logically).

Here I implemented insertion sort on Python, with tracing number of swaps:

```
def insertionSort(l):
    counter = 0
    listLength = len(l)
    for x in range(1, listLength):
        pivot = l[x]
        pos = x

        while l[pos - 1] > pivot and pos > 0:
            counter += 1
            l[pos] = l[pos - 1]
            pos = pos - 1
        l[pos] = pivot
    return counter
```

Yusuf Patoglu 151044027

Average Case Analysis of Insertion Sort:

We know that Insertion Sort has O(n^2) average case complexity. There is a theorem that says: *"The average number of inversions in an array of N distinct elements is n (n – 1) / 4"*

In order to prove that we can assume *"each element is about halfway in order"*

$$\sum_{i=1}^{N-1} \frac{i}{2} = \frac{1}{2}(1+2+3...+(N-1)) = \frac{(N-1)N}{4}$$

*Which gives us **O(n^2)** complexity.*

Quick Sort:

For my implementation of Quick Sort, I returned both list and count values. Count value is the counter that traces how many swaps performed in sort operation. As we know that Quick Sort has partition method that sorts elements with 2 pivots. So whole code is looks like:

```
def partition(l, low, high):
    counter = 0
    mid = (high + low) // 2
    pivot = high
    if l[low] < l[high]:
        pivot = low
    elif l[low] < l[mid]:
        if l[mid] < l[high]:
            pivot = mid


    index = pivot
    currentElement = l[index]
    l[index], l[low] = l[low], l[index]
    limit = low


    for x in range(low, high + 1):
        counter+=1;
        if l[x] < currentElement:
            limit += 1
            l[x], l[limit] = l[limit], l[x]
    l[low], l[limit] = l[limit], l[low]

    return limit,counter


def quickSort(l, low, high):
    counter = 0
    if low < high:
        limit, counter = partition(l, low, high)
        counter += quickSort(l, low, limit - 1)
        counter += quickSort(l, limit + 1, high)
    return counter
```

Yusuf Patoglu 151044027

Average Case Analysis of Quick Sort:

-Quick Sort has recurrence relation as:

$$C_N = N + 1 + \frac{1}{N} \sum_{1 \le k \le N} (C_{k-1} + C_{N-k})$$ where N>2

-If we multiply then rearrange this, we'll get:

$$NC_N = (N+1)C_{N-1} + 2N.$$

-If we divide by N each side of equation we'll get this form:

$$\begin{aligned} \frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\ &= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\ &= \vdots \\ &= \frac{C_2}{3} + \sum_{3 \le k \le N} \frac{2}{k+1}. \end{aligned}$$

-Now it's clear that we have telescoping series so we can apply integral to it to find complexity:

$$\frac{C_N}{N+1} \approx 2 \sum_{1 \le k < N} \frac{1}{k} \approx 2 \int_1^N \frac{1}{x} dx$$

This will equal to 2N *ln N which is **O(n*logn)**

(Image Sources: https://courses.cs.washington.edu/courses/cse312/11wi/slides/08sort.pdf)

Discussion:

We have found that Insertion Sort has O(n^2) complexity and Quick Sort has O(n * logn) complexity. For larger arrays without a doubt Quick Sort is more logical to use. Bad side of Quick Sort is it's space complexity, it can consume a lot stack while recursing. Actually in some cases where n input is small value insertion sort can work faster because it's iterative.

Yusuf Patoglu 151044027

4-) Median of an Unsorted Array

Our problem is simple, we have an unsorted array and we want to find median of it. There is a restriction that algorithm must be divide and conquer. In order to solve that algorithm I need decrease and conquer algorithm. I could choose Insertion Sort or Binary Search like algorithm but since our input is small I will choose an iterative approach.

In order to solve that I decided to go with Insertion Sort because it's decrease and conquer algorithm. Simply I sorted the array with Selection Sort then returned the middle element which is median.

```
def decreaseMedian(l):
    listLength = len(l)
    for x in range(1, listLength):
        pivot = l[x]
        pos = x

        while l[pos - 1] > pivot and pos > 0:
            l[pos] = l[pos - 1]
            pos = pos - 1
        l[pos] = pivot

    return l[listLength // 2]
```

Worst Case is where input is reversely sorted here, so it's **O(n^2)** because it's triggers both outer and inner loop.

Yusuf Patoglu 151044027

4-) Exhaustive Subset:

In order to find all subsets of a list I used a recursive code which is have time complexity of **O(2^n)**. Also I needed to iterate two times to get optimal sub array. These iterations cost **O(n^2)** but since we know that **O(2^n)** is asymptotically bigger so the total time complexity is **O(2^n)**

```
def subs(l):
    if l == []:
        return [[]]

    x = subs(l[1:])

    return x + [[l[0]] + y for y in x]
def exhaustiveSearch(n):
    exList = []
    for x in range(n):
        exList.append(random.randrange(n))

    minVal = min(exList)
    maxVal = max(exList)
    limit = (minVal + maxVal) * (n // 4)
    allSubsets = subs(exList)
    lenAllSubsets = len(allSubsets)
    specifiedSub = []
    minx = 99999
    for x in range(lenAllSubsets):
        sum = 0
        y = 0
        for y in range(len(allSubsets[x])):
            if sum < limit:
                specifiedSub.append(allSubsets[x])


    for x in range(len(specifiedSub)):
        sum = 1

        for y in range(len(specifiedSub[x])):
            sum = sum * specifiedSub[x][y]
            if(sum < minx):
                sum = minx
                index = x
    print("The optimal sub-array is", specifiedSub[x])
```