# CSE 321 HW 1

Elif Akgün
1801042251

## 1 pseudocode:

* L is a list of 2n-boxes, the first n of them are black and the remaining n boxes are white.

```
function Boxes ( L [1: 2n] )
    for i=1 to 2n-1 do
        temp=0
        if (L[i] = L[i+1]) then
            for j=i+1 to n+temp do
                L[j+1] = L[j]
            end for
            L[i+1] = L[j+1]
            temp = temp + 1
        end if
    end for
```

My algorithm resemble to insertion sort. According to my algorithm, first, it checks whether the first two elements are the same. If they are not same, checks the second and third element and it goes on to 2n th element. If they are same, it shifts the elements from second element to (n+temp)th element. ("temp" is zero in the beginning). Because (n+temp)th element is always white. And it goes on up to list likes BWBW...

For example

L = {B,B,B,W,W,W}
i = 1 temp = 0
L[1] = = L[2] then
j = 2
L[3] = L[2]
new list → L = {B,B,B,W,W,W}
j = 3
L[4] = L[3]
new list → L = {B,B,B,B,W,W}
j = 4 X remove from for loop
L[2] = L[4]
new list → L = {B,W,B,B,W,W}
temp=1

i = 2 temp = 1 L = {B,W,B,B,W,W}
L[2] ! = L[3] → does not into if condition

i = 3 temp = 1 L = {B,W,B,B,W,W}
L[3] = = L[4] then
j = 4
L[5] = L[4]
new list → L = {B,W,B,B,B,W}
j = 5 X → remove from for loop
L[4] = L[6]
new list → L = {B,W,B,W,B,W}
temp = 2

i = 4 temp = 2 L = {B,W,B,W,B,W}
L[4] ! = L[5]
⋮

* from i = 4 to i = 2n-1 it does not into the first if condition and ends the for loop. List is sorted.

Best case : In the best case, we have sorted list and it can only be L={B,W} Because we have 2n elements and n can be 1 at least. If n=2 we have L={B,B,W,W} that is unsorted. Assume that our list is L={B,W}

According to my algorithm it enters the for loop and compare the first element and second element. Then exits the for loop. So there are one comparison and complexity is O(1). But if we generalize, for all situation it enters the for loop from i=1 to i=2n-1. So our complexity is $O(2n-1) \in \underline{O(n)}$

Worst case : In worst case, we have unsorted list. It can be L={B,B,B,W,W,W} In this situation, it enters the both for loops until the array is sorted.

$$\sum_{i=1}^{2n-1}\sum_{j=i+1}^{n+temp} 1 = \sum_{i=1}^{2n-1}(n+temp-i-\cancel{1}+\cancel{1}) = \sum_{i=1}^{2n-1}(n+temp-i)$$

$$= \underbrace{(n-1+temp)+(n-2+temp)+\ldots+(n+temp-2n+1)}_{\#\text{ of element is } 2n-1} \approx (2n-1)*n \approx 2n^2 \in \underline{\underline{O(n^2)}}$$

Average case:

1 comparison if 2n=2

3 comparision if 2n=4

5 comparision if 2n=6

⋮

2n-1 comparision if 2n

$$\left.\begin{matrix} \sum_{i=1}^{2n-1}(n+temp-i) \rightarrow \text{all possible cases.} \\ \approx 2n^2 \end{matrix}\right\}$$

\# of cases = 2n-1

$\therefore$ Avg. case complexity $= \dfrac{2n^2}{2n-1} \approx n \in \underline{\underline{O(n)}}$

(2) In this problem, most popular solution is dividing coins into two part. There are two parts $n/2$, $n/2$ or $n/2$, $n/2+1$; $n$ is number of coin. Since dividing into three parts is more efficient, my algorithm divides $n$ coins into three part if $n \geq 3$. If $n=1$ and if there are exactly one fake coin, this coin is fake. If $n=2$ then lightier is fake. (We assume that fake coin is lighter). If $n=3$, divide three part these coins and each part has one coin. To find heavier coin, compares first two part. If they are equal, third is fake; else heavier is fake. Thinking like this, we can create an algorithm for large number of coins. For $n>=3$; $n$ can be $3k$, $3k+1$ or $3k+2$. These comes moduler arithmetic. Let's analyze these three case:

case 1) If $n=3k$ each part has $\frac{n}{3}=k$ coins $(k, k, k)$

case 2) If $n=3k+1$, parts are $k, k, k+1$

case 3) If $n=3k+2$, parts are $k, k, k+2$.

For all cases:

→ First, compare weight of first two part.

→ If they are equal, fake coin is in part 3. Then continue part 3 and do the same steps.

→ If they are not equal, lighter part has fake coin, then continue with the lighter part and do the same steps.

Let's write a pseudocode for this algorithm:

```
function  findFakeCoin ( L[1:n] )
    if n=1 then
        return L[1]
    end if
    if n=2 then
        if L[1].weight() < L[2].weight() then
            return L[1]
        end if
        else
            return L[2]
        end else
    end if
```

```
else
    // sums of elements of each parts
    Sum 1 = 0
    sum 2 = 0
    sum 3 = 0

    for i = 1 to n/3 do
        sum1 = sum1 + L[i].weight()
    end for
    for i = n/3 + 1 to 2n/3 do
        Sum 2 = sum 2 + L(i).weight()
    end for
    for i = 2n/3 + 1 to n do
        sum 3 = sum 3 + L[i].weight()
    end for
    if sum 1 = sum 2 then
        findFakeCoin ( L[ 2n/3 + 1 : n] )
    end if
    else if sum 1 < sum 2 then
        findFakeCoin ( L[1 : n/3] )
    end else if
    else
        findFakeCoin ( L[ n/3 + 1 : 2n/3 ] )
    end else
end else
```

Worst case: We can easily set up a recurrence relation for the number of weighing $W(n)$ needed by this algorithm in the worst case:

$$W(n) = W\left(\frac{n}{3}\right) + 1 \qquad n > 1 \quad W(1) = 0$$

It is almost identical to the one for the worst-case number of comparisons in binary search. (The differences are initial condition and dividing 2 parts in binary search) This is more efficient. Complexity is $O(\log_3 n)$

Average case is $\Omega(\log n)$.

Best case is $O(\log_3 n)$ because if we generalize, all the time it divide the 3 parts.

③ Insertion sort (Average case)

Let's L be the list that will sort. ($L[1:n]$)

1 comparison $\leftarrow$ $x = L[1] > L[i-1]$

2 comparison $\leftarrow$ $L[i-2] < x < L[i-1]$

⋮

i comparison $\leftarrow$ $x < L[1]$

* i+1 possible position and we assume that each interval is equally likely.

$$P(T_i = j) = \begin{cases} \frac{1}{i+1} , & 1 \le j \le i-1 \\ \frac{2}{i+1} , & j = i \end{cases}$$

* P is probability that there are j comparisions.

$$\frac{1}{i+1} \sum_{j=1}^{i+1} j = \frac{1}{i+1} \frac{(i+1)(i+2)}{2} = \frac{i+2}{2}$$

$$\Rightarrow \sum_{i=1}^{n-1} \frac{i+2}{2} = \sum_{i=1}^{n-1} \frac{i}{2} + \underbrace{\sum_{i=1}^{n-1} 1}_{n-1} = n-1 + \frac{1}{2} \sum_{i=1}^{n-1} i = n-1 + \frac{1}{2} \frac{(n-1)(n-1+1)}{2}$$

(for ($i=1$ to $n-1$))

$$= \frac{n(n-1)}{4} + n - 1 = \frac{n^2 - n + 4n - 4}{4} = \frac{n^2+3n-4}{4} \in \underline{\underline{\Theta(n^2)}}$$

Quick Sort (Average case)

Let's L be the list that will sort. ($L[1:n]$)

Assume that pivot ($L[low]$) will be any position after Rearrange.

$$T = \underbrace{T_1}_{\text{Rearrange}} + \underbrace{T_2}_{\text{recursive call.}}$$

* The array can be split from any index. ($0 \le i \le n-1$). In total ($n+1$) comparisons are made. After the partition, two split has $i$ and $n-i-1$ elements. probability of splitting from any index is $1/n$. So:

$$\frac{1}{n} \sum_{i=0}^{n-1} [(n+1) + Avg(i) + Avg(n-i-1)]$$

* position of pivot $\rightarrow L[1]$ $\Rightarrow$ subsets: $L[1:0]$, $L[2:n]$

" " " $L[2]$ $\Rightarrow$ " $L[1:1]$, $L[3:n]$

" " " $L[3]$ $\Rightarrow$ " $L[1:2]$, $L[4:n]$

"

⋮

$Avg(0) = 0$, $Avg(1) = 0$

$\Rightarrow Avg(n) \approx n\log n \rightarrow \underline{\underline{\Theta(n\log n)}}$

* When list is sorted, in quick sort, this is worst case. Because, when the list is sorted, there is a empty subarray. All elements bigger or smaller than pivot. So there are n recursive call and n checking between all elements and pivot. ($O(n^2)$)

When list is sorted, in insertion sort, this is the best case. Because only one comparision required for each insertion. ($O(n)$)

∴ When list is sorted or nearly sorted so not very complex, using insertion sort is better than quick sort. Counter of insertion sort less than counter of quick sort when the list is sorted.

* when list is reverse order, in insertion sort, this is worst case. Maximum comparison needed in each iteration. ($O(n^2)$)

When list is reverse order, in quick sort there is no differences from above situation.

(*) When the list is not very complex, the quick sort is better than insertion sort. So it is more efficient.

For example

→ when the list is L=[1,2,3,4,5,6] counters are:

insertion - counter = 0
quick - counter = 5

→ L = [6,5,4,3,2,1]

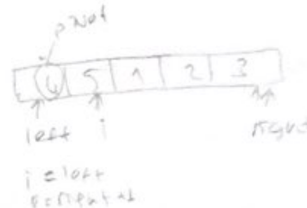
insertion _ counter = 15
quick _ counter = 5

→ L = [2,1,4,3,5,6]

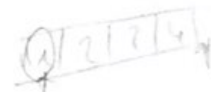
insertion - counter = 2
quick - counter = 3

(4) The middle value is called <u>median.</u>

In my algorithm, it solves with partition. The output is ith smallest element. We want to find medium, so we should find $(length/2)^{th}$ element.




```
function findMedian(L[0:n-1], k]
    left = 0
    right = n-1
    while left ≤ right do
        pivot = L[left]
        i = left
        j = right + 1
        repeat
            repeat i = i+1 until L[i] ≥ pivot
            repeat j = j-1 until L[j] ≤ pivot do
            repeat j = j-1 until L[j] ≤ pivot do
            swap (L[i], L[j]
        until i ≥ j
        swap (L(i], L(j])
        swap (L(left], L(right])
        if j > k-1 then
            right = j-1
        else if j < k-1
            left = j+1
        else
            return L(k-1]
```




* This algorithm resemble to quick sort. In this algorithm, like quicksort we take an pivot (L[left]). Then we search the array both from beginning and from end. compares the pivot and elements in the indeks, then according to circumstance swap elements.

* like the quicksort, if the list is sorted, this is the worst case. There is n checking between all elements and pivot ≈ n times.

complexity is $\underline{\underline{\Theta(n^2)}}$

(5) In this question, I used helper resursive functions. In my algorithm, first, it found all subarray with Subarrays function. This function returns an array that include all subarrays. Then it found subarrays that's sum of elements bigger then or equal $\frac{(min+max)*n}{4}$ and returns an array that includes providing this condition. subarrays. (conditionalSubarrays does these). Then, multiplicationOfConditionalSubarrays returns an array that includes multiplication of providing this condition subarrays. finally, findOptimalSubarray function finds index of min. multiplication index then returns an element of returning conditionalSubarrays function, in the index. This helper function runs O(n) So worst case is O(n)