Algorithms Homework 2

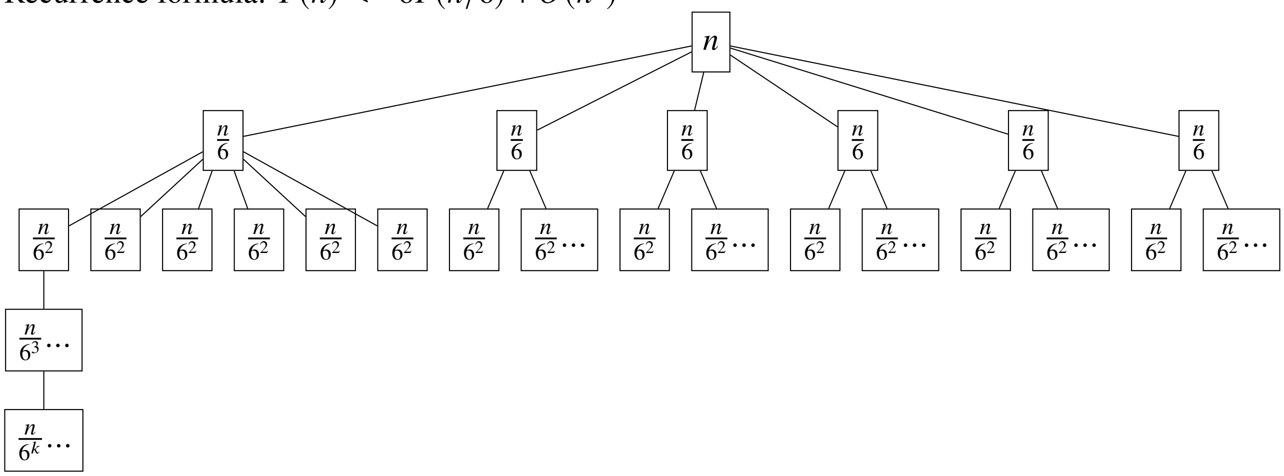
BRANDON YUEN, by161, Section 06 & RYAN LEE, rsl82, Section 07 & JAMES ALBA, jma

This is a joint assignment between Brandon Yuen, Ryan Lee, and James Alba

September 2022

Problem 1: Brandon Yuen, by161, section 06, Ryan Lee, rsl82, section07, and James Alba, jma390, section 07

1. Recurrence formula: $T(n) <= 6T(n/6) + O(n^3)$



Each layer has 6 terms from each node, could not display all nodes because the diagram would expand too far out of the page for us to see

Non-recursive works per layer
Layer 0: n^3
Layer 1: $6 \cdot (\frac{n}{6})^3 = \frac{n^3}{6^2}$
Layer 2: $6^2 \cdot (\frac{n}{6^2})^3 = \frac{n^3}{(6^2)^2}$
.
.
Layer k: $6^k \cdot (\frac{n}{6^k})^3 = \frac{n^3}{(6^2)^k}$

Value for the k
$\frac{n}{6^k} = 1$
$n = 6^k$
$k = \log_6 n$

Overall works $n^3 + \frac{n^3}{6^2} + \frac{n^3}{6^{2\cdot 2}} + ... + \frac{n^3}{6^{2k}} =$
$\sum_{k=0}^{\log_6 n}(\frac{1}{36^k} \cdot n^3) = n^3 \cdot (\frac{1-(\frac{1}{36})^{\log_6(n)+1}}{1-\frac{1}{36}})$
We assume n is big so, $= n^3 \cdot (\frac{1-0+1)}{1-\frac{1}{36}}) = C \cdot n^3 = O(n^3) = T(n)$

Induction Proof
$T(n) \le 6T(\frac{n}{6}) + O(n^3)$
Claim : $T(n) \le k \cdot n^3$
Base : $T(1), T(2) \le C' < k$
Inductive hypothesis : $T(x) \le k \cdot x^3$ for all $x < n$
Proof : $T(n) \le 6T(\frac{n}{6}) + C \cdot n^3$
$\le 6 \cdot (k \cdot (\frac{n}{6})^3) + C \cdot n^3$
$= \frac{6 \cdot k \cdot n^3}{6^3} + C \cdot n^3 = \frac{k \cdot n^3}{36} + C \cdot n^3 = (\frac{k}{36} + C) \cdot n^3 \le k \cdot n^3$ when k gets big

2. Recurrence formula: $T(n) = 16T(n/4) + n^2$

$\boxed{n}$

$\boxed{n/4\ldots \text{ (total of 16 nodes at this level)}}$

$\boxed{n/4^2\ldots\text{(total of } 16^2 \text{ nodes at this level)}}$

$\boxed{n/4^3\ldots\text{(total of } 16^3 \text{ nodes at this level)}}$

$\boxed{n/4^k\text{(total of } 16^k \text{ nodes at each level)}}$

non-recursive works per layer
layer 0: n^2
layer 1: $16 \cdot \frac{n^2}{4^2} = n^2$
layer 2: $16^2 \cdot \frac{n^2}{4^{2\cdot 2}} = n^2$
.
.
layer k: $16^k \cdot \frac{n^2}{4^{2k}} = n^2$
$k = \log_4(n)$
summation $= n^2 + n^2 + \ldots + n^2 = \sum_{k=0}^{\log_4 n} n^2 = (1 + log_4 n) \cdot n^2$
Therefore, $T(n) = O(n^2 \log_4(n))$

3. Recurrence formula: $T(n) = 8T(n/5) + n$

$\boxed{n}$

$\boxed{n/5\ldots \text{ (total of 8 nodes at this level)}}$

$\boxed{n/5^2\ldots\text{(total of } 8^2 \text{ nodes at this level)}}$

$\boxed{n/5^3\ldots\text{(total of } 8^3 \text{ nodes at this level)}}$

$\boxed{n/5^k\ldots\text{(total of } 8^k \text{ nodes at each level)}}$

non-recursive works per layer
Layer 0: n
Layer 1:$8 \cdot \frac{n}{5} = \frac{8n}{5}$
Layer 2:$8^2 \cdot \frac{n}{5^2} = \frac{8^2 \cdot n}{5^2}$
.
.
Layer k:$\frac{8^k \cdot n}{5^k}$
$k = \log_5(n)$

Total works
$= \sum_{k=0}^{log_5 n} n \cdot (8/5)^k$
$= n \cdot \frac{(\frac{8}{5})^{\log_5(n+1)}-1}{\frac{8}{5}-1}$
$= n \cdot \frac{8^{\log_5(n+1)}-1}{5^{\log_5(n+1)}} \cdot \frac{5}{3} = n \cdot \frac{40 \cdot 8^{\log_5 n}-5}{3(n+1)} \leq n \cdot \frac{40 \cdot 8^{log_5 n}}{3n} = \frac{40 \cdot n^{log_5 8}}{3} = O(n^{log_5 8})$
Therefore, $T(n) = O(n^{\log_5(8)})$

Problem 2: Brandon Yuen, by161, section 06, Ryan Lee, rsl82, section07, and James Alba, jma390, section 07

1. Use FindSquareRoot(k,1,k).

Algorithm 1 FindSquareRoot(k,min,max)

> **if** $max < min$ **then**
>   **return** No Solution
> **end if**
> int squareRoot $\leftarrow min + \frac{(max-min)}{2}$
> squared $\leftarrow$ squareRoot $*$ squareRoot
> **if** squared = k **then**
>   **return** squareRoot
> **else if** sqaured < k **then**
>   FindSquareRoot(k,floor(squareRoot+1),max)
> **else**
>   FindSquareRoot(k,min,ceiling(squareRoot-1))
> **end if**

Recursion formula = T(n) = T(n/2) + O(1)

Problem 3: Brandon Yuen, by161, section 06, Ryan Lee, rsl82, section07, and James Alba, jma390, section 07

1. Answer Line 1: Nothing
2. Answer Line 2: if $i \neq n : S' = S' + (n - i)$

Problem 4: Brandon Yuen, by161, section 06, Ryan Lee, rsl82, section07, and James Alba, jma390, section 07

Algorithm 2 IsFrequent(A[],x)

1. int frequencyCount $\leftarrow$ 0
    for i=0 to n-1 **do**
        if $A[i] = x$ **then**
            frequencyCount = frequencyCount + 1
        end if
    end for

    if frequencyCount $\geq \frac{n}{4}$ **then**
        return True
    else
        return False
    end if

Algorithm 3 findFrequentElement(A[])

2.

```
n ← A.length
if n ≤ 4 then
    return A
end if
left ← first half of A = A[0] to A[n/2 -1]
right ← second half of A = A[n/2] to A[n-1]
aLeft ← findFrequentElement(left)
aRight ← findFrequentElement(right)
freq ← []
count ← 0
for i in aLeft do
    if isFrequent(A,i) = true then
        if checkDup(freq,i) = true then
            freq[count] ← i
            count ← count + 1
        end if
    end if
end for

for i in aRight do
    if isFrequent(A,i) = true then
        if checkDup(freq,i) = true then
            freq[count] ← i
            count ← count + 1
        end if
    end if
end for
if count = 0 then
    return No Solution
else
    return freq
end if
```

Algorithm 4 checkDup(A[],x)

```
for i in A do
    if i = x then
        return false
    end if
end for
return true
```

I added helper method checkDup for easier implementation. It checks duplication in freq array.

Non-recursive works for findFrequentElement(A[])
line 5-6 : split array = O(1)
line 9-10: O(1)
line 12: isFrequent(A,i) = O(n)
line 13 16: checkDup(freq,i) = O(1) because length of array freq is at most 4 because number of frequent elements is at most 4.
line 11: for loop = O(1) itself, because the length of aLeft is at most 4 for same reason above.
Therefore, line 11 18 : O(1) * O(n) = O(n)

line 21: isFrequent(A,i) = O(n)
line 22 25: checkDup(freq,i) = O(1) because length of array freq is at most 4 because number of frequent elements is at most 4.
line 20: for loop = O(1) itself, because the length of aRight is at most 4 for same reason above.
Therefore, line 20 27 : O(1) * O(n) = O(n)
line 28 32: O(1)

So, Overall non-recursive work is O(n)

Recurrence formula = T(n) = 2T(n/2) + O(n)
= O(nlogn)

Problem 5: Brandon Yuen, by161, section 06, Ryan Lee, rsl82, section07, and James Alba, jma390, section 07

Algorithm 5 frequentMoore(A[])

```
element1 ← null
count1 ← 0
element2 ← null
count2 ← 0
element3 ← null
count3 ← 0
element4 ← null
count4 ← 0

for i in A do
    if count1 = 0 then
        element1 ← i
        count1 ← count1 + 1
    else if element1 = i then
        count1 ← count1 + 1
    else if count2 = 0 then
        element2 ← i
        count2 ← count2 + 1
    else if element2 = i then
        count2 ← count2 + 1
    else if count3 = 0 then
        element3 ← i
        count3 ← count3 + 1
    else if element3 = i then
        count3 ← count3 + 1
    else if count4 = 0 then
        element4 ← i
        count4 ← count4 + 1
    else if element4 = i then
        count4 ← count4 + 1
    else
        count1 ← count1 - 1
        count2 ← count2 - 1
        count3 ← count3 - 1
        count4 ← count4 - 1
    end if
end for
```

Algorithm 6 continue..

```
count1 ← 0
count2 ← 0
count3 ← 0
count4 ← 0
for i in A do
    if i = element1 then
        count1 ← count1 + 1
    else if i = element2 then
        count2 ← count2 + 1
    else if i = element3 then
        count3 ← count3 + 1
    else if i = element4 then
        count4 ← count4 + 1
    end if
end for
frequentArr ← []
elementCount ← 0
n ← A.length
if count1 ≥ n/4 then
    frequentArr[elementCount] ← element1
    elementCount ← elementCount + 1
end if
if count2 ≥ n/4 then
    frequentArr[elementCount] ← element2
    elementCount ← elementCount + 1
end if
if count3 ≥ n/4 then
    frequentArr[elementCount] ← element3
    elementCount ← elementCount + 1
end if
if count4 ≥ n/4 then
    frequentArr[elementCount] ← element4
    elementCount ← elementCount + 1
end if

if elementCount = 0 then
    return No Solution
else
    return frequentArr
end if
```

We have 4 placeholders for elements(element1 4), because frequent elements are at most 4. After first for loop, elements1 4 should contain 4 most frequent elements in there.
After that, we check if this most frequent elements are fulfill the condition. We count the frequencies of these elements again and see if these elements are equal or more than 4/n.
This algorithm just go through A twice with for loop in line 10 in first half and in line 5 in second half. And everything else is O(1), so overall big O is O(n).