Name: Kevina Wong
ID: 109179049

CSCI 3104, Algorithms **Profs. Chen & Grochow**
Problem Set 3 – Due Thurs Feb 6 11:55pm **Spring 2020, CU-Boulder**

Advice 1: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.

Advice 2: Informal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Instructions for submitting your solutions:

- All submissions must be easily legible.
- You should submit your work through the **class Canvas page** only.
- You may not need a full page for your solutions; pagebreaks are there to help Gradescope automatically find where each problem is. Even if you do not attempt every problem, please allot at least as many pages per problem (or subproblem) as are allotted in this template.

Quicklinks: 1a 1b 2a 2b 3 4

---

1. Solve the following recurrence relations. For each case, show your work.
   (a) $T(n) = 2T(n-1) + 1$ if $n > 1$, and $T(1) = 2$.
   Answer:
   $T_n = 2T_{n-1} + 1$ *Given equation*
   $T_1 = 2$ *Given information*

   $T_2 = 2T_1 + 1$ *Equation for the second recurrence (n=2) of the loop*
   $T_2 = 2(2) + 1$ *Plug in* $T_1 = 2$
   $T_2 = 2^2 + 2^1 - 1$ *Write in exponent form*

   $T_3 = 2(T_2) + 1$ *Equation for the third recurrence (n=3) of the loop*
   $T_3 = 2(2(2) + 1) + 1$ *Plug in* $T_2 = 2(2) + 1$
   $T_3 = 2^3 + 2^2 - 1$ *Write in exponent form*

   $T_4 = 2(t_3) + 1$ *Equation for the fourth recurrence (n=4) of the loop*
   $T_4 = 2(2(2(2) + 1) + 1) + 1$ *Plug in* $T_3 = 2(2(2) + 1) + 1$
   $T_4 = 2^4 + 2^3 - 1$ *Write in exponent form*

   Closed Form Solution: $T_n = 2^n + 2^{n-1} - 1$
   Big-O = $\mathcal{O}(2^n)$

Name: Kevina Wong
ID: 109179049

(b) $T(n) = 3T(\frac{n}{2})+\Theta(n)$ if $n > 1$, and $T(1) = \Theta(1)$. Use the plug-in/substitution/unrolling method.

Answer:

$T_n = 3(T_{\frac{n}{2}}) + \Theta(n))$ *Given equation*
$T_n = 3(T_{\frac{n}{2}}) + cn$ *Replace* $\Theta(n)$ *for some constant* cn

$T_{\frac{n}{2}} = 3(T_{\frac{n}{4}}) + \frac{cn}{2}$ *Equation for the recursion when* $n=\frac{n}{2}$
$T_n = 3(3(T_{\frac{n}{4}} + \frac{cn}{2})) + cn$ *Plug in for* $T_{\frac{n}{2}} = 3(T_{\frac{n}{4}}) + \frac{cn}{2}$
$T_n = 3^2(T_{\frac{n}{4}}) + 3^1(\frac{cn}{2}) + cn$ *Simiply Exponents*

$T_{\frac{n}{4}} = 3(T_{\frac{n}{8}}) + \frac{cn}{4}$ *Equation for the recursion when* $n=\frac{n}{4}$
$T_n = 3(3(3(T_{\frac{n}{8}}) + \frac{cn}{4}) + \frac{cn}{2}) + cn$ *Plug in for* $T_{\frac{n}{4}} = 3(T_{\frac{n}{8}}) + \frac{cn}{4}$
$T_n = 3^3(T_{\frac{n}{4}}) + 3^2(\frac{cn}{4}) + 3^1(\frac{cn}{2}) + cn$ *Simiply Exponents*

$T_{\frac{n}{8}} = 3(T_{\frac{n}{16}}) + \frac{cn}{8}$ *Equation for the recursion when* $n=\frac{n}{4}$
$T_n = 3(3(3(3(T_{\frac{n}{16}}) + \frac{cn}{8}) + \frac{cn}{4}) + \frac{cn}{2}) + cn$ *Plug in for* $T_{\frac{n}{8}} = 3(T_{\frac{n}{16}}) + \frac{cn}{8}$
$T_n = 3^4(T_{\frac{n}{4}}) + 3^3(\frac{cn}{8}) + 3^2(\frac{cn}{4}) + 3^1(\frac{cn}{2}) + cn$ *Simiply Exponents*

$T_n = 3^k(T_{\frac{n}{2^k}}) + 3^{k-1}(\frac{cn}{2^{k-1}}) + ... + 3^1(\frac{cn}{2^1}) + cn$ *Write equation for the pattern*
$T_n = 3^k(T_{\frac{n}{2^k}}) + cn[\sum_{i=1}^{k-1}(\frac{3}{2})^i] + cn$ *Write in Summation Notation*
$T_n = 3^k(T_{\frac{n}{2^k}}) + cn[2^{1-k}3^k - 3] + cn$ *Simplify Summation*

$1 = \frac{n}{2^k}$ *Solve for constant* k
$2^k = n$ *Multiply both sides by* 2^k
$log_2 2^k = log_2 n$ *Take* log_2 *of both sides*
$k = log_2 n$ *Here we have solved for the constant* k

$T_n = 3^{log_2 n}(T_{\frac{n}{2^{log_2 n}}}) + cn[2^{1-log_2 n}3^{log_2 n} - 3] + cn$ *Plug in* $k = log_2 n$
$T_n = 3^{log_2 n}(T_1) + cn[\frac{(2)3^{log_2 n}}{n} - 3] + cn$ *Plug in Simplifications*
$T_n = 3^{log_2 n}(T_1) + (2c)3^{log_2 n} - 2cn$ *Simplify Further*

$T_n = 3^{log_2 n}(\Theta(1)) + (2c)3^{log_2 n} - 2cn$ *Plug in* $T_1 = \Theta(1)$
Note that $\Theta(1)$ *is asymptotically the same as 1*

Closed Form Solution: $T_n = 3^{log_2 n} + (2c)3^{log_2 n} - 2cn$
Big-O: $\mathcal{O}(3^{log_2 n})$

Name: Kevina Wong
ID: 109179049

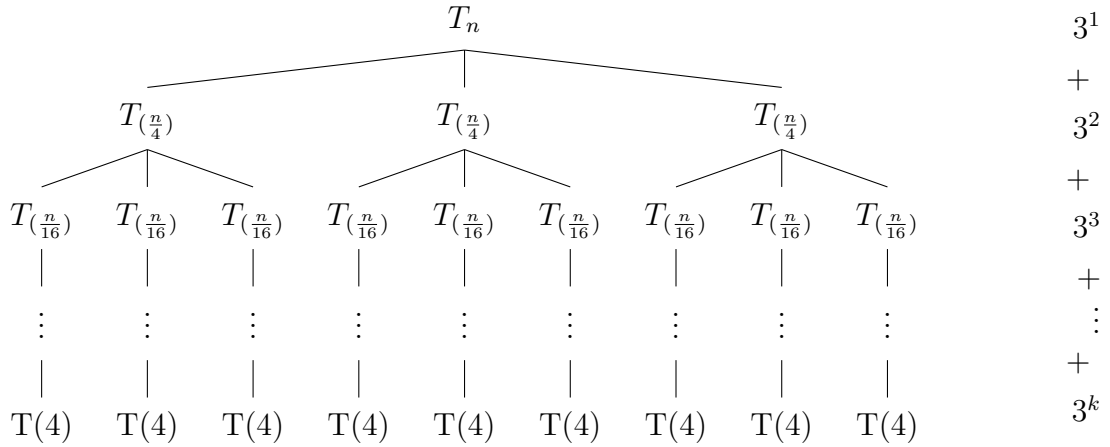
CSCI 3104, Algorithms **Profs. Chen & Grochow**
Problem Set 3 – Due Thurs Feb 6 11:55pm **Spring 2020, CU-Boulder**

---

2. Consider the following functions. For each of them, determine how many times is 'hey' printed in terms of the input n. You should first write down a recurrence and then solve it **using the recurrence tree method.** That means you should write down the first few levels of the recurrence tree, specify the pattern, and then solve.

(a)
```
def fun(n) {
      if (n > 1) {
         print( 'hi' 'hi' 'hi' )
         fun(n/4)
         fun(n/4)
         fun(n/4)
}}
```

Recurrence Relation: $T_n = 3(T_{\frac{n}{4}}) + 3$

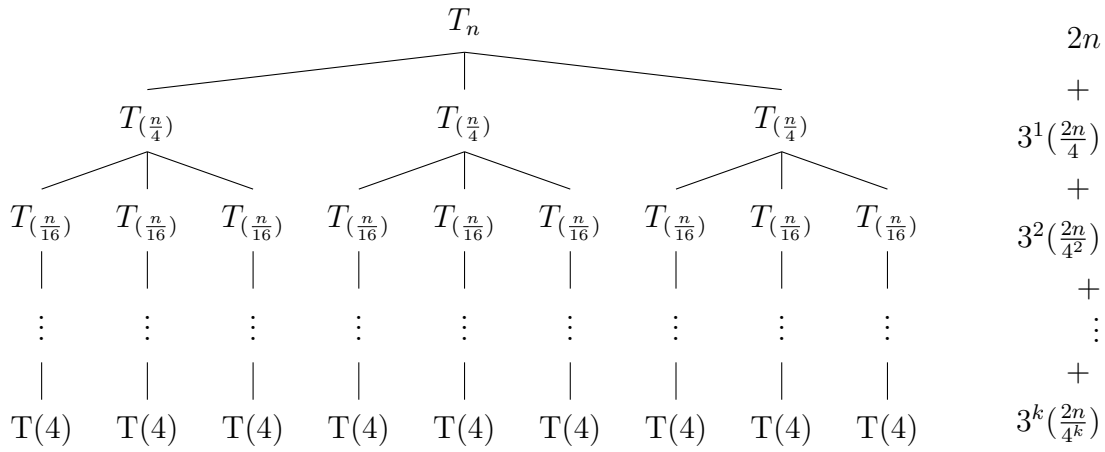


Size	Prints Per Node	Node	Prints per Level
n	3	3^0	$3(3^0) = 3^1$
$\frac{n}{4^1}$	3	3^1	$3(3^1) = 3^2$
$\frac{n}{4^2}$	3	3^2	$3(3^2) = 3^3$
⋮	⋮	⋮	⋮
$\frac{n}{4^k}$	3	3^k	(3^{k+1})

Note that k = number of tiers
Pattern of "hi"s outputted: $T_n = \sum_{i=0}^{k} 3^{i+1}$.
Solve for k = Number of Tiers = $k = log_4 n + 1$
Solution: $T_n = \sum_{i=0}^{log_4 n+1} 3^{i+1} = T_n = (\frac{3}{2})3^{log_4 n+1} - 1$.
Big-Theta = $\Theta(3^{log_4 n})$ **or** $\Theta(n^{log_4(3)})$
Note that they are equivalent.

(b)
```
def fun(n) {
      if (n > 1) {
         for i=1 to n {
           print( 'hi' 'hi' )
         }
         fun(n/4)
         fun(n/4)
         fun(n/4)
}}
```

Recurrence Relation: $T_n = 3T_{n/4} + 2n$



Size	Prints Per Node	Node	Prints per Level
n	$\frac{2n}{4^0}$	3^0	$(2\frac{n}{4^0})3^0$
$\frac{n}{4^1}$	$\frac{2n}{4^1}$	3^1	$(2\frac{n}{4^1})3^1$
$\frac{n}{4^2}$	$\frac{2n}{4^2}$	3^2	$(2\frac{n}{4^2})3^2$
⋮	⋮	⋮	⋮
$\frac{n}{4^k}$	$\frac{2n}{4^k}$	3^k	$(2\frac{n}{4^k})3^k$

Note that k = number of tiers
Pattern of "hi"s outputted: $T_n = \sum_{i=0}^{k} 2(\frac{n}{4^i})3^i$.
Solve for k = Number of Tiers = $k = log_4 n + 1$
Solution: $T_n = \sum_{i=0}^{log_4 n+1} 2(\frac{n}{4^i})3^i = 8n - 3^{log_4 n+2}$
Big-Theta = $\Theta(n)$

3. Consider the following algorithm

```
fun(A[1, ..., 4n]):
    if A.length == 0:
        return 0
    return 1 + fun(A[3, ..., 4n-2])
```

Find a recurrence for the worst-case runtime complexity of this algorithm. Then solve your recurrence and get a tight bound on the worst-case runtime complexity.

Answer:

$$\begin{cases} T_0 = \Theta(1) & n = 0 \\ T_{4n} = T_{4n-4} + c & n > 0 \end{cases}$$

Let $m = 4n$

$T_m = T_{n-4} + c$
$T_{m-4} = T_{m-4-4} + c$

$T_m = T_{n-8} + c + c$
$T_{m-8} = T_{m-4-4-4} + c$

$T_m = Tm - 12 + c + c + c$
$T_m = T_{m-4k} + kc$ *Pattern*

$0 = m - 4k$ *Solve for constant* k
$k = \frac{m}{4}$ *Constant* k *solved*

$T_m = T_{m-4(\frac{m}{4})} + (\frac{m}{4})c$ *Plug in for constant* $k = \frac{m}{4}$
$T_m = \Theta(1) + \frac{mc}{4}$
$T_{4n} = \Theta(1) + \frac{4nc}{4}$ *Plug in* $m = 4n$
$T_{4n} = nc$
$T_{4n} = \mathcal{O}(n)$

$\lim_{n\to\infty} \frac{n}{nc} = \frac{1}{c}$ *Applying the limit comparison test, since the limit is a constant number, we can conclude that a tight bound can be represented as* $\Theta(n)$.

Tight Bound = $\Theta(n)$

4. (Recall Problem 4 in Problem Set 1) Given an array $A = [a_1, a_2, \cdots, a_n]$, a reverse is a pair (a_i, a_j) such that $i < j$ but $a_i > a_j$. Design a divide-and-conquer algorithm with a runtime of $O(n \log n)$ for computing the number of reverses in the array. Your solution to this question needs to include both a written explanation and an implementation of your algorithm, including:

   (a) Your algorithm has to be a divide and conquer algorithm that is modified from the Merge Sort algorithm. Explain how your algorithm works, including pseudocode.

   (b) Implement your algorithm in Python, C, C++, or Java. **You MUST submit a runnable source code file. You will not receive credit if we cannot compile your code. Do NOT simply copy/paste your code into the PDF.**

   (c) Randomly generate an array of 100 numbers and use it as input to run your code. Report on both the input to and the output of your code.

Answer (4a):

The way that divide and conquer algorithms work is that it is split into three processes: *Divide, Conquer, and Combine*. The way that this algorithm works is similar to merge sort. In the beginning, we must *divide*. First, the array needs to be divided in half, then those arrays need to be divided in half, and so on until we have arrays that are small enough to solve simply. Then, we need to solve each of the "broken down steps". After the broken down arrays are all sorted in order, all of the elements needs to be recombined into the original array, but in a sorted order. For more details, refer below for the pseudocode and comments associated with the pseudocode.

```
def mergeSort(toSort):
    #create length variable for convenience
    length=len(toSort)
    #split if the length of the array is greater than 1

    #DIVIDE
    if length > 1:
        #get index of half the array
        half = length//2

        #left array is the first half of the original array
        left = toSort[0:half]
        #right array is the first half of the original array
        right = toSort[half:length]

        #call the function recursively to keep splitting the array in half
        mergeSort(leftList)
        mergeSort(rightList)

        #counter: left array index
        i=0
        #counter: right array index
        j=0
        #counter: original array index
        k=0

        #arrange "nodes" in the correct order

        #CONQUER
        while i < len(left) and j < len(right):
            #if the right number is larger than the left number.
            if right[j] > left[i]
                #Note that we don't need to increase count here because nothing woul
                toSort[k] = left[i]
                i=i+1
                k=k+1
            else:
                #increase count by the size of the right array to account for every
```

```
                count=count+len(right)-i
                toSort[k] = right[j]
                j=j+1
                k=k+1
        #COMBINE
         while i < len(left):
             toSort[k] = left[i]
             i=i+1
             k=k+1
         while j < len(right):
             toSort[k] = right [j]
             j=j+1
             k=k+1
return toSort, count
```

Answer (4b and 4c:)

Refer to the mergesort.py file that is turned in with this PDF.

References and Collaboration:

- Office Hours (Luke, Nick, and Grochow)
- Collaborated and worked with other students in Office Hours
- Week3.pdf