CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2021**

Homework #1

January 26, 2021

Name: Yinfeng Cong *Extension: No*

Homework Policy

- If you leave a question completely blank, you will receive 25% of the grade for that question. This however does not apply to the extra credit questions.
- You are allowed to discuss the homework problems with other students in the class. **But you must write your solutions independently.** You may also consult all the materials used in this course (video recordings, notes, textbook, etc.) while writing your solution, but no other resources are allowed.
- Do not forget to write down your name and whether or not you are using one of your two extensions. Submit your homework on Canvas.
- Unless specified otherwise, you may use any algorithm covered in class as a "black box" – for example you can simply write "sort the array in $\Theta(n \log n)$ time using merge sort".
- Remember to always **prove the correctness** of your algorithms and **analyze their running time** (or any other efficiency measure asked in the question). See "Practice Homework" for an example.
- The "Challenge yourself" and "Fun with algorithms" are both extra credit. These problems are significantly more challenging than the standard problems you see in this course (including lectures, homeworks, and exams). As a general rule, only attempt to solve these problems if you enjoy them.

Problem 1. This question reviews asymptotic notation. You may assume the following inequalities for this question (and throughout the course): For any constant $c \geq 1$,

$$(\log n)^c = o(n) \quad , \quad n^c = o(n^{c+1}) \quad , \quad c^n = o((c+1)^n) \quad , \quad (n/2)^{(n/2)} = o(n!) \quad , \quad n! = o(n^n).$$

(a) Rank the following functions based on their asymptotic value in the increasing order, i.e., list them as functions $f_1, f_2, f_3, \ldots, f_9$ such that $f_1 = O(f_2)$, $f_2 = O(f_3), \ldots, f_8 = O(f_9)$. Remember to write down your proof for each equation $f_i = O(f_{i+1})$ in the sequence above. **(15 points)**

$\sqrt{\log n}$	$\log \log n$	$2^{\log n}$
$100n$	10^n	$2^{2^{2^{2}}}$
2^n	$n!$	$\dfrac{n}{\log n}$

Hint: For some of the proofs, you can simply show that $f_i(n) \leq f_{i+1}(n)$ for all sufficiently large n which immediately implies $f_i = O(f_{i+1})$.

Solution. Note :$1 < \log\log n < \log n < \sqrt{n} < n < n \log n < n\sqrt{n} < n^2 < n^3 < ..... < 2^n < 3^n < ..... < n! < n^n$

$2^{2^{2^{2}}} = O(1)$, we know that it is the smallest one, so $f_1 = 2^{2^{2^{2}}}$

then we need to compare $\log\log n, \sqrt{\log n}$ to find the smaller one of them as f_2

$\log\log n = (\log n)^(1/2)$, we set $\log n = a$, the comparison becomes comparing $a^(1/2)$ and $\log a$, from the note we can easily know that $\log a < a^(1/2)$
so $\log\log n < \sqrt{\log n}$, $f_2 = \log\log n$
Then we compare $\sqrt{\log n}, \frac{n}{\log n}, 2^{\log n}$
we find a sufficient large number 10000000000 to compare them, we find that $\sqrt{\log n} < \frac{n}{\log n} < 2^{\log n}$
so we set $f_3 = \sqrt{\log n}$, $f_4 = \frac{n}{\log n}$, $f_5 = 2^{\log n}$
Then we compare $2^{\log n}, 2^n, 10^n, 100n$
$2^{\log n} = n$, and we always know that $n < 100n$
Also, we always know that $100n < 2^n < 10^n$, (because for large sufficient number $x^n > m * n$, and $10 > 2$)
Then we know $f_6 = 100n, f_7 = 2^n, f_8 = 10^n$
At last, we need to compare $10^n, n!$, and $n! = O(n^n)$ which is much bigger than 10^n
so $f_9 = n!$

(b) Consider the following four different functions $f(n)$:

$$1 \qquad \log n \qquad n^2 \qquad 4^{4^n}.$$

For each of these functions, determine which of the following statements is true and which one is false. Remember to write down your proof for each choice. **(10 points)**

- $f(n) = \Theta(f(n-1))$;
- $f(n) = \Theta(f(\frac{n}{2}))$;
- $f(n) = \Theta(f(\sqrt{n}))$;

Example: For the function $f(n) = 4^{4^n}$, we have $f(n-1) = 4^{4^{n-1}}$. Since $4^{4^{n-1}} = 4^{\frac{1}{4}\cdot 4^n} = (4^{4^n})^{1/4}$.

$$\lim_{n\to\infty} \frac{f(n)}{f(n-1)} = \lim_{n\to\infty} \frac{4^{4^n}}{4^{4^{n-1}}} = \lim_{n\to\infty} \frac{4^{4^n}}{(4^{4^n})^{1/4}} = \lim_{n\to\infty} (4^{4^n})^{3/4} = +\infty.$$

As such, $f(n) \neq O(f(n-1))$ and thus the first statement is false for 4^{4^n}.

Solution. For 1, we have

$$\lim_{n\to\infty} \frac{f(n)}{f(n-1)} = \lim_{n\to\infty} \frac{1}{1} = 1$$

so

$f(n) = \Theta(f(n-1))$

$$\lim_{n\to\infty} \frac{f(n)}{f(n/2)} = \lim_{n\to\infty} \frac{1}{1} = 1$$

so

$f(n) = \Theta(f(\frac{n}{2}))$;

$$\lim_{n\to\infty} \frac{f(n)}{f(\sqrt{n})} = \lim_{n\to\infty} \frac{1}{1} = 1$$

so

$f(n) = \Theta(f(\sqrt{n}))$;

For $\log n$, we have

$$\lim_{n\to\infty} \frac{f(n)}{f(n-1)} = \lim_{n\to\infty} \frac{\log n}{\log(n-1)} = \lim_{n\to\infty} \frac{1/n}{1/(n-1)} = 1$$

so

$f(n) = \Theta(f(n-1))$

$$\lim_{n\to\infty} \frac{f(n)}{f(n/2)} = \lim_{n\to\infty} \frac{\log n}{\log (n/2)} = \lim_{n\to\infty} \frac{1/n}{2/n} = 1/2$$

so

$f(n) = \Theta(f(\frac{n}{2}))$;

$$\lim_{n\to\infty} \frac{f(n)}{f(\sqrt{n})} = \lim_{n\to\infty} \frac{\log n}{1/2 \log n} = 2$$

so

$f(n) = \Theta(f(\sqrt{n}))$;

For n^2, we have

$$\lim_{n\to\infty} \frac{f(n)}{f(n-1)} = \lim_{n\to\infty} \frac{n^2}{(n-1)^2} = \lim_{n\to\infty} \frac{2n}{2(n-1)} = 1$$

so

$f(n) = \Theta(f(n-1))$

$$\lim_{n\to\infty} \frac{f(n)}{f(n/2)} = \lim_{n\to\infty} \frac{n^2}{(n/2)^2} = 4$$

so

$f(n) = \Theta(f(\frac{n}{2}))$;

$$\lim_{n\to\infty} \frac{f(n)}{f(\sqrt{n})} = \lim_{n\to\infty} \frac{n^2}{(\sqrt{n})^2} = +\infty$$

As such, $f(n) \neq O(f(n-1))$ and thus the statement is false

For 4^{4^n}, we have $f(n-1) = 4^{4^{n-1}}$. Since $4^{4^{n-1}} = 4^{\frac{1}{4}\cdot 4^n} = (4^{4^n})^{1/4}$.

$$\lim_{n\to\infty} \frac{f(n)}{f(n-1)} = \lim_{n\to\infty} \frac{4^{4^n}}{4^{4^{n-1}}} = \lim_{n\to\infty} \frac{4^{4^n}}{(4^{4^n})^{1/4}} = \lim_{n\to\infty} (4^{4^n})^{3/4} = +\infty.$$

As such, $f(n) \neq O(f(n-1))$ and thus the first statement is false for 4^{4^n}.

$$\lim_{n\to\infty} \frac{f(n)}{f(n/2)} = \lim_{n\to\infty} \frac{4^{4^n}}{4^{4^{n/2}}} = \lim_{n\to\infty} 4^{4^{(n-(n/2))}} = +\infty$$

As such, $f(n) \neq O(f(n/2))$ and thus the second statement is false for 4^{4^n}.

$$\lim_{n\to\infty} \frac{f(n)}{f(\sqrt{n})} = \lim_{n\to\infty} \frac{4^{4^n}}{4^{4^{\sqrt{n}}}} = \lim_{n\to\infty} 4^{4^{n^{(1-1/2)}}} = +\infty$$

As such, $f(n) \neq O(f(\sqrt{n}))$ and thus the third statement is false for 4^{4^n}.

Problem 2. Your goal in this problem is to analyze the *runtime* of the following (imaginary) recursive algorithms for some (even more imaginary) problem:

(A) Algorithm A divides an instance of size n into 4 subproblems of size $n/4$ each, recursively solves each one, and then takes $O(n)$ time to combine the solutions and output the answer.

Solution. Solution to part (a) goes here.

$$T\left(n\right) \leq 4 \cdot T\left(\frac{n}{4}\right) + O\left(n\right)$$

on level 1 and 2 it both have $C * n$ in total
the number of levels: $\log_4\left(n\right)$
level i:

$$4^{i-1} \cdot C \cdot \frac{n}{4^{i-1}} = C \cdot n$$

sum:

$$\log_4\left(n\right) \cdot C \cdot n$$

$$T\left(n\right) \leq O\left(n \cdot \log\left(n\right)\right)$$

(B) Algorithm B divides an instance of size n into 2 subproblems, one with size $n/4$ and one with size $n/5$, recursively solves each one, and then takes $O(n)$ time to combine the solutions and output the answer.

Solution. Solution to part (b) goes here.

$$T\left(n\right) \leq T\left(\frac{n}{4}\right) + T\left(\frac{n}{5}\right) + O\left(n\right)$$

level 1 : Cn
level 2: $(9/20)Cn$
level 3:$(9/20)^2Cn$
number of levels:

$$\frac{n}{4^k} = 1,\ k\ =\ \log_4\left(n\right)$$

level i :

$$\left(\frac{9}{20}\right)^{i-1} \cdot C \cdot n$$

sum:

$$C \cdot n \cdot \sum_{i=1}^{\log_4(n)} \left(\frac{9}{20}\right)^{i-1} = \frac{1}{1-\frac{9}{20}} \cdot C \cdot n = O\left(n\right)$$

$$T\left(n\right) \leq O\left(n\right)$$

(C) Algorithm C divides an instance of size n into 4 subproblems of size $n/3$ each, recursively solves each one, and then takes $O(n^2)$ time to combine the solutions and output the answer.

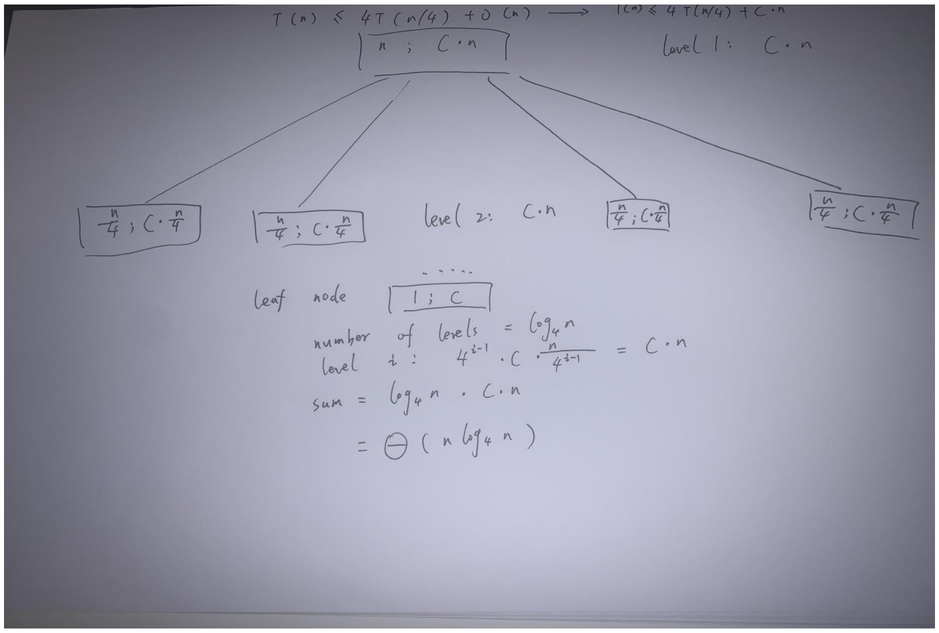
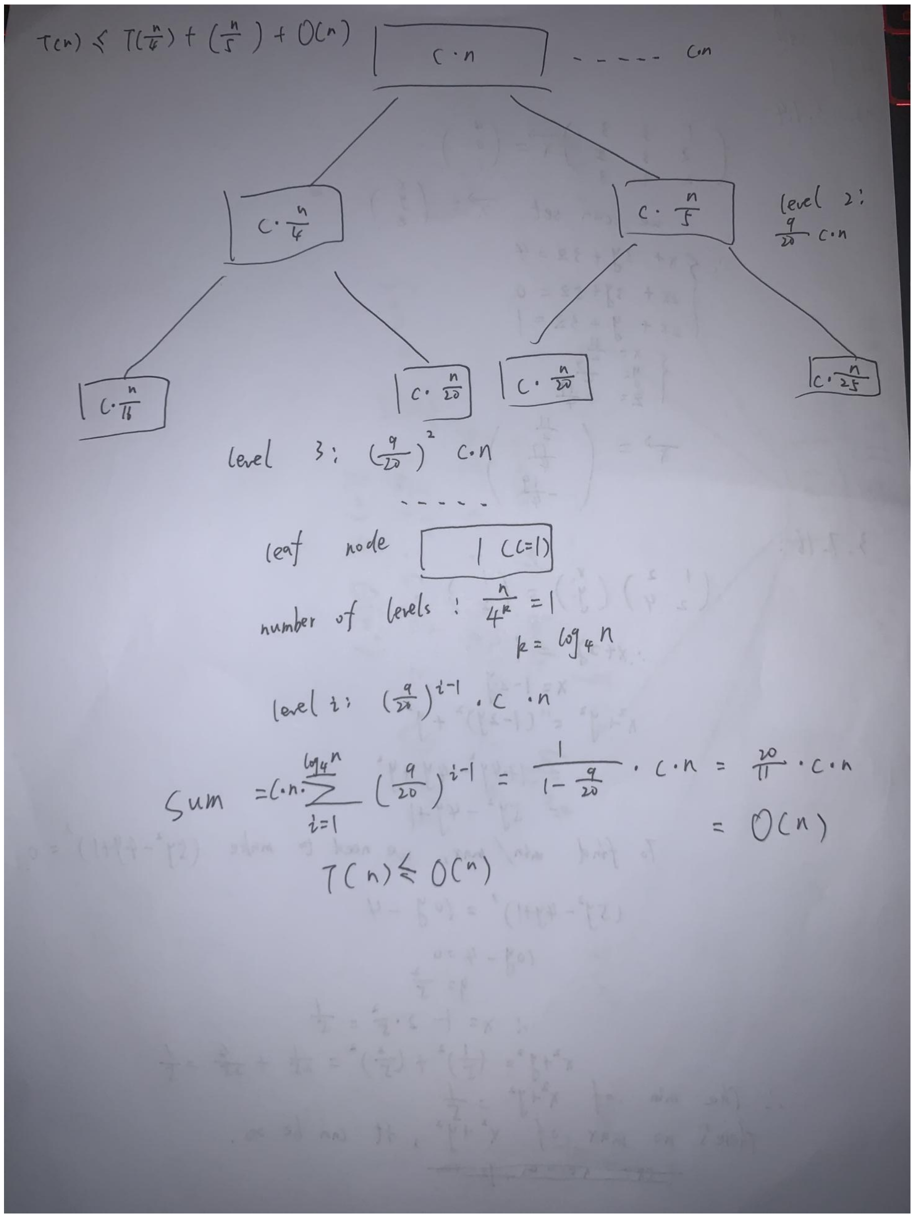

$T(n) \leq 4T(n/4) + O(n) \longrightarrow T(n) \leq 4T(n/4) + C \cdot n$
$n \,;\, C \cdot n$
level 1: $C \cdot n$
$\frac{n}{4} \,;\, C \cdot \frac{n}{4}$
$\frac{n}{4} \,;\, C \cdot \frac{n}{4}$
level 2: $C \cdot n$
$\frac{n}{4} \,;\, C \cdot \frac{n}{4}$
$\frac{n}{4} \,;\, C \cdot \frac{n}{4}$
leaf node $1 \,;\, C$
number of levels $= \log_4 n$
level i: $4^{i-1} \cdot C \cdot \frac{n}{4^{i-1}} = C \cdot n$
sum $= \log_4 n \cdot C \cdot n$
$= \Theta(n \log_4 n)$

Figure 1: Recursion tree for algorithm A.



$T(n) \leq T(\frac{n}{4}) + (\frac{n}{5}) + O(n)$

level 3: $(\frac{9}{20})^2 c \cdot n$

- - - - -

leaf node $\boxed{1 \ (c=1)}$

number of levels : $\frac{n}{4^k} = 1$

$k = \log_4 n$

level i: $(\frac{9}{20})^{i-1} \cdot c \cdot n$

$$\text{Sum} = c \cdot n \cdot \sum_{i=1}^{\log_4 n} \left(\frac{9}{20}\right)^{i-1} = \frac{1}{1-\frac{9}{20}} \cdot c \cdot n = \frac{20}{11} \cdot c \cdot n = O(n)$$

$T(n) \leq O(n)$

Figure 2: Recursion tree for algorithm B.

Solution. Solution to part (c) goes here.

$$T(n) \leq 4 \cdot T\left(\frac{n}{3}\right) + O\left(n^2\right)$$

level 1: Cn^2
level 2: $4/3Cn^2$
number of levels: $\log_3(n)$
level i :

$$\left(\frac{4}{9}\right)^{i-1} \cdot C \cdot n^2$$

sum:

$$\sum_{i=1}^{\log_3(n)} \left(\frac{4}{9}\right)^{i-1} \cdot C \cdot n^2 = O\left(n^2\right)$$

$$T(n) \leq O\left(n^2\right)$$

(D) Algorithm D divides an instance of size n in to 2 subproblems of size $n-1$ each, recursively solves each one, and then takes $O(1)$ time to combine the solutions and output the answer.

Solution. Solution to part (d) goes here.

$$T(n) \leq 2 \cdot T(n-1) + O(1)$$

level 1: C
level 2: $2C$
level 3: $4C$
number of levels: n
level i :

$$2^{i-1} \cdot C$$

Sum:

$$\sum_{i=1}^{n} 2^{i-1} \cdot C = 2^{n-1} \cdot C = O\left(2^n\right)$$

$$T(n) \leq O\left(2^n\right)$$

For each algorithm, write a recurrence for its runtime and *use the recursion tree method* of Lecture 3 to solve this recurrence and find the *tightest asymptotic* upper bound on runtime of the algorithm. **(25 points)**

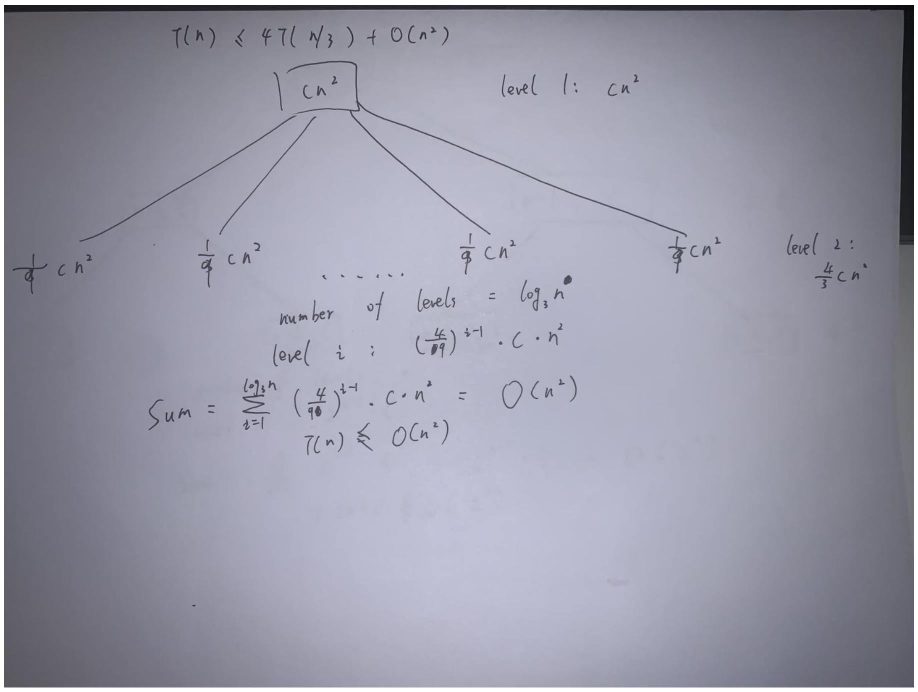

$T(n) \leq 4T(n/3) + O(n^2)$
cn^2
level 1: cn^2
$\frac{1}{9}cn^2$
$\frac{1}{9}cn^2$
$\frac{1}{9}cn^2$
$\frac{1}{9}cn^2$
level 2: $\frac{4}{9}cn^2$
number of levels = $\log_3 n$
level i : $(\frac{4}{9})^{i-1} \cdot c \cdot n^2$
Sum = $\sum_{i=1}^{\log_3 n} (\frac{4}{9})^{i-1} \cdot c \cdot n^2 = O(n^2)$
$T(n) \leq O(n^2)$

Figure 3: Recursion tree for algorithm C.

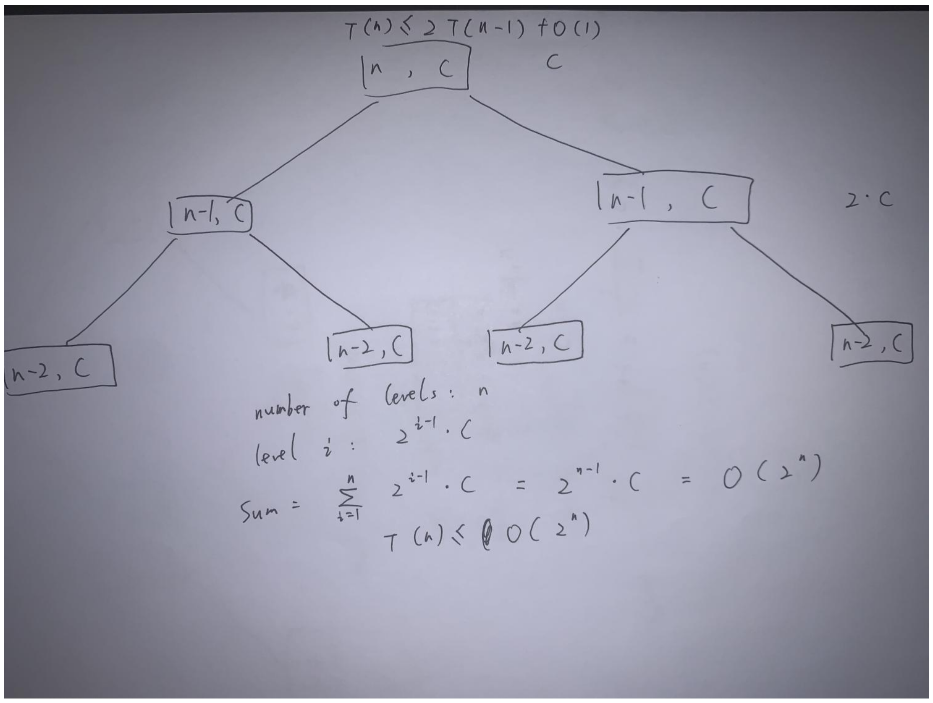

$T(n) \leq 2T(n-1) + O(1)$
n, c
c
$n-1, c$
$n-1, c$
$2 \cdot c$
$n-2, c$
$n-2, c$
$n-2, c$
$n-2, c$
number of levels: n
level i: $2^{i-1} \cdot c$
Sum $= \sum_{i=1}^{n} 2^{i-1} \cdot c = 2^{n-1} \cdot c = O(2^n)$
$T(n) \leq O(2^n)$

Figure 4: Recursion tree for algorithm D.

Problem 3. In this problem, we consider a non-standard sorting algorithm called the *Silly Sort.* Given an array $A[1:n]$ of n integers, the algorithm is as follows:

- **Silly-Sort**($A[1:n]$):

1. If $n < 5$, run merge sort (or selection sort or insertion sort) on A.
2. Otherwise, run **Silly-Sort**($A[1:3n/4]$), **Silly-Sort**($A[n/4:n]$), and **Silly-Sort**($A[1:3n/4]$) again.

We now analyze this algorithm.

(a) Prove the correctness of **Silly-Sort**. **(10 points)**

Solution. Solution to part (a) goes here.
To solve this problem, we need to utilize induction (for proof of correctness)
I use the way professor prove merge sort in lecture to prove this problem.
base case: when n< 5,
we use merge sort as we are taught in the lecture, it is correct, and the runtime of it will be O(1) [becacuse it is a specific number < 5 here, so it will not be nlogn]
inductive step:
Suppose it is true for n=m, then we need to prove it is also correct for n=m+1
So for [1,3/4m], [m/4,m], and [1,3/4m], the answer is correct, we need to prove for [1,3/4m +1], [m/4, m+1], and [1, 3/4m +1]
For n=m, the area [1/4m,3/4m] was the part that got sorted again and again
For n = m+1, the area becomes [m,3/4m +1] and the array becomes m+1
It is proving the correctness

(b) Write a recurrence for **Silly-Sort** and use the recursion tree method of Lecture 3 to solve this recurrence and find the *tightest asymptotic* upper bound on the runtime of **Silly-Sort**. **(10 points)**

Solution. Solution to part (b) goes here.
$T(n) \leq 3 \cdot T\left(\frac{3}{4}n\right) + O(1)$
level 1: C
level 2: 3 C
number of levels:$\log_{\frac{3}{4}}\left(\frac{5}{n}\right)$
level i: 3^{i-1}
sum:$\sum_{i=0}^{\log_{\frac{3}{4}}\left(\frac{5}{n}\right)} 3^{i-1} \cdot C = 3^{\log(n)}$

(c) Suppose we like to change the second line of the algorithm to

Silly-Sort($A[1:m]$), **Silly-Sort**($A[n-m:n]$), and **Silly-Sort**($A[1:m]$)

for some other value of m instead (in the original algorithm, $m = 3n/4$).

What is the smallest number we could pick while still maintaining the correctness of the algorithm? What would be the runtime of the resulting algorithm? For this part of the question, you can simply write a few lines for proof of correctness and runtime analysis by pointing out how your proofs and calculations in parts (a) and (b) should be changed. **(5 points)**

Solution. Solution to part (c) goes here.
the smallest number we should pick is $\frac{2}{3}n$
the runtime of it will not be changed, because 3/4 is the base of log, and when we change the base of

log to m, the runtime does not change it is still $3^{\log(n)}$
For the reason that in (b) it was $\left(\frac{9}{4}\right)^{i-1}$, and here it becomes $(3m)^{i-1}$, that is the only difference
The only difference of proof of correctness is that the area that got sorted becomes [n-m,m] (because m must be bigger than 2/3, so it can not be [m,n-m])

Problem 4. You are given an array $A[1:n]$ which includes the scores of n players in a game. They are ranked in the following way: Rank of a player is an integer r if there are exactly $r-1$ *distinct* scores strictly smaller than the score of this player (irrespective of the number of players).

(a) Design and analyze an algorithm that given the array A, can find the rank of all players in the array in $O(n \log n)$ time. **(15 points)**

Example. Suppose the input array is $A = [1, 7, 6, 5, 2, 4, 5, 2]$ for 8 players; then the rank of players is:

- Player $A[1]$ has rank 1 (as $A[1] = 1$ is the smallest number);
- Player $A[2]$ has rank 6 (as $A[2] = 7$ has 5 distinct smaller numbers: $\{1, 6, 5, 4, 2\}$);
- Player $A[3]$ has rank 5 (as $A[3] = 6$ has 4 distinct smaller numbers: $\{1, 5, 4, 2\}$);
- Player $A[4]$ has rank 4 (as $A[4] = 5$ has 3 distinct smaller numbers: $\{1, 4, 2\}$);
- Player $A[5]$ has rank 2 (as $A[5] = 2$ has 1 distinct smaller number: $\{1\}$);
- Player $A[6]$ has rank 3 (as $A[6] = 4$ has 2 distinct smaller numbers: $\{1, 2\}$);
- Player $A[7]$ has rank 4 (as $A[7] = 5$ has 3 distinct smaller numbers: $\{1, 4, 2\}$);
- Player $A[8]$ has rank 2 (as $A[8] = 2$ has 1 distinct smaller number: $\{1\}$);

Solution. Solution to part (a) goes here.
To solve this question I apply what we are taught in the lecture (merge sort) to help me solve it.
algorithm:
First, we do merge sort on array A
Separate A[1,n] to two parts : B[1,n/2] and C[n/2+1,n]
Create two pointers p and q
For $i = 1$to n
— If $B[p] < C[q]$, let $A[i] = B[p]$ and set $p = p + 1, i = i + 1$
— Else, let $A[i] = C[q]$ and set $q = q + 1, i = i + 1$
return A
Second, we need to make sure $A[a] = x$ is the first x appeared in the array
while $(A[a] = A[a-1])$
— set a = a-1
return a
print the final a we get after the while loop, and it would be the first x appeared in the array
Also, this a is the rank of player we want to search
Proof of correctness:
we have proved the correctness of merge sort in the lecture, so we do not need to prove it again.
For the second part, we have base case min[A[0],A[1]], return a is true
we suppose A[m] is correct, and we need to prove it for A[m+1]
For A[m+1], it is find [A[m],A[m+1]]
proving the induction step
Runtime analysis:
we have also known from the lecture that the runtime of merge sort is O(n logn)
For this algorithm we add a while part, so the new big O is O(n logn + n) = O(n (logn+1)) = O(n logn)

(b) Suppose you are additionally given an array $B[1:m]$ with the score of m new players. Design and analyze an algorithm that given both arrays A and B, can find the rank of each player B inside the array A, i.e., for each $B[i]$, determines what would be the rank of $B[i]$ in the array consisting of all elements of A plus $B[i]$. Your algorithm should run in $O((n+m)\cdot\log n)$ time. **(10 points)**

Example. Suppose the input array is $A = [1,7,6,5,2,4,5,2]$ as before and $B = [3,9,4]$; then the correct answer for each player in B is:

- Player $B[1]$ will have rank 3 (as $B[1] = 3$ has 2 distinct smaller numbers in A: $\{1,2\}$);
- Player $B[2]$ will have rank 7 (as $B[2] = 9$ has 6 distinct smaller numbers in A: $\{1,7,6,5,4,2\}$);
- Player $B[3]$ will have rank 3 (as $B[3] = 4$ has 2 distinct smaller numbers in A: $\{1,2\}$);

Solution. Solution to part (b) goes here.
To solve this problem, we need to apply merge sort again.
First, we do merge sort on array A (the same as (a))
Separate A[1,n] to two parts : C[1,n/2] and D[n/2+1,n]
Create two pointers p and q
For $i = 1$to n
— If $C[p] < D[q]$, let $A[i] = C[p]$ and set $p = p+1, i = i+1$
— Else, let $A[i] = D[q]$ and set $q = q+1, i = i+1$
return A
Second, we do merge sort on array B
Separate B[1,m] to two parts : E[1,m/2] and F[m/2+1,m]
Create two pointers x and y
For $j = 1$to m
— If $E[x] < F[y]$, let $A[j] = E[x]$ and set $x = x+1, j = j+1$
— Else, let $A[j] = F[y]$ and set $y = y+1, j = j+1$
return B
Third, we need to compare the score in the array B with the score in the array A to figure out the rank of scores in B
For $z = 1$ to m and o =1 to n
— if $B[z] > A[o]$, set $o = o+1$,
— if $B[z] <= A[o]$, return o and set $z = z+1$
The final o we print is the rank we need to get.
Proof of correctness:
we have proved the correctness of merge sort in the lecture, so we do not need to prove it again.
For the second part, base case: B[1]¡ A[1],the answer is the 1 for the rank
inductive step: we set it is correct for B[n], and we want to prove it for B[n+1]
which is finding min[A[n],B[n+1]], the correctness is proving
Runtime analysis:
we have also known from the lecture that the runtime of merge sort is O(n logn)
For this algorithm we let array B go through the array A to find its rank in A, so the runtime of it is O(m logn)
In total, the runtime is O(m+n logn)

Challenge Yourself. Let us revisit the community detection problem but with an interesting twist. Remember that we have a collection of n people for some odd integer n and we know that strictly more than half of them belong to a hidden community. As before, when we introduce two people together, the members of the hidden community would say they know the other person if they also belong to the community, and otherwise they say they do not know the other person. The twist is now as follows: the people that do not belong to the community *may lie*, meaning that they may decide to say they know the other person even though in reality only people inside the hidden community know each other.

Concretely, suppose we introduce two people A and B, then what they will say would be one of the following (first part of tuple is the answer of A and second part is the answer of B):

- if both belong: (*know* , *know*);
- if A belongs and B does not: (*does not know* , *know/does not know*);
- if B belongs and A does not: (*know/does not know* , *does not know*);
- if neither belongs: (*know/does not know* , *know/does not know*);

Design an algorithm that finds all members of the hidden community using $O(n)$ greetings. **(+10 points)**

Fun with Algorithms. We have an n-story building and a series of magical vases that work as follows: there is some unknown level L in the building that if we throw these vases down from any of the levels $L, L+1, \ldots, n$, they will definitely break; however, no matter how many times we throw the vases down from any level below L nothing will happen them. Our goal in this question is to determine this level L by throwing the vases from different levels of the building (!).

For each of the scenarios below, design an algorithm that uses asymptotically the smallest number of times we throw a vase (so the measure of efficiency for us is the number of vase throws).

(a) When we have only one vase. Once we break the vase, there is nothing else we can do. **(+2 points)**

Solution. Solution to part (a) goes here. For this situation, we only have one vase, and we can not make it break to test the magnitude of L
So we need to go through from the bottom, to the level L (1,2,3.....L)
As a result, the runtime would be O(n)

(b) When we have four vases. Once we break all four vases, there is nothing else we can do. **(+4 points)**

Solution. Solution to part (b) goes here. This time we have 4 vases, which means that we can waste 3 times of vase.
So we can apply the bianry search at the first time
we throw the vase at n/2 to see if it breaks (to find the worst case, we assume it breaks)
Then we throw the second vase at n/4 to see if it breaks(again to find the worst case, we assume it breaks)
Then we throw the second vase at n/8 to see if it breaks(again to find the worst case, we assume it breaks)
So what we need to find is searching from 1st floor to the n/8th floor.
The runtime would be O(n/8)

(c) When we have an unlimited number of vases. **(+4 points)**

Solution. Solution to part (c) goes here. This time we have infinite chances to test the floor, so we can use binray search from the beginning to the end of the test
As we are taught in the lecture, the runtime takes for binary search is O(log n)
So the answer would be O(log n)
