Name: Jeremy M. Hein

ID: 810952267

CSCI 3104, Algorithms **Charlie Carlson & Ewan Davies**

Problem Set 3 – Due Sep 17 11.59pm MDT **Fall 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:

- The solutions **should be typed using** LaTeX and we cannot accept hand-written solutions. Here's a short intro to LaTeX.
- You should submit your work through the **class Canvas page** only.
- You may not need a full page for your solutions; page breaks are there to help Gradescope automatically find where each problem is. Even if you do not attempt every problem, please submit this template of at least 6 pages (or Gradescope has issues with it). **We will not accept submissions with fewer than 5 pages**.
- **You must CITE any outside sources you use, including websites and other people with whom you have collaborated. You do not need to cite a CA, TA, or course instructor.**
- **Posting questions to message boards or tutoring services including, but not limited to, Chegg, StackExchange, etc., is STRICTLY PROHIBITED. Doing so is a violation of the Honor Code.**

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

You are welcome to use the following formula without proof:

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

Name: Jeremy M. Hein
ID: 810952267

CSCI 3104, Algorithms **Charlie Carlson & Ewan Davies**
Problem Set 3 – Due Sep 17 11.59pm MDT **Fall 2020, CU-Boulder**

Problem 1. Analyze the worst-case running time for each of the following algorithms. You should give your answer in Big-Theta notation. You *do not* need to give an input which achieves your worst-case bound, but you should try to give as tight a bound as possible.

- In the column to the right of the code, indicate the cost of executing each line once.
- In the next column (all the way to the right), indicate the number of times each line is executed.
- Below the code, justify your answers (show your work), and compute the total runtime in terms of Big-Theta notation. You may assume that $T(n)$ is the Big-Theta of the high-order term. You need not use Calculus techniques. However, you **must** show the calculations to determine the closed-form expression for a summation (represented with the $\sum$ symbol). You cannot simply say: $\sum_{i=1}^{n} i \in \Theta(n^2)$, for instance.
- In both columns, you don't have to put the *exact* values. For example, putting "c" for constant is fine. We will not be picky about off-by-one errors (i.e., the difference between n and $n-1$); however, we will be picky about off-by-two errors (i.e., the difference between n and $n-2$).

(1a) Consider the following algorithm.

```
                                                 | Cost       | # times run
1   f(A[1, ..., n][1, ..., n]):
2     let d be a copy of A                       | $c_2n^2$   | 1
3     for i = 1 to n:                            | $c_3$      | $n$
4       d[i][i] = 0                              | $c_4$      | $n-1$
5
6     for i = 1 to n:                            | $c_5$      | $n$
7       for j = 1 to n:                          | $c_6$      | $n^2$
8         for k = 1 to n:                        | $c_7$      | $n^3$
9           if (d[i][k] + d[k][j]) < d[i][j]:    | $c_8$      | $n^3-1$
10            d[i][j] = d[i][k] + d[k][j]        | $c_9$      | $\sum_{k=1}^{n} t_k$
11
12    return d                                   | $c_{10}$   | 1
```

Note: The numbers below are in reference to the lines of codes presented in the algorithm above.

2) Since we're allocating a 2D array, we need to allocate $n \times n$ elements (rows x columns). Our model dictates allocating one element costs constant time, c_2. So then we have $c_2 \times n \times n = c_2 n^2$ cost. This line only needs to be executed once.

3) The code checks the value of loop variable i, n times. The action of checking the loop variable is constant cost of c_3.

4) The element $d[i][i]$ is set to 0, $n - 1$ times (one less than the for loop). The cost of setting each element to 0 is constant c_4.

6) The code checks the value of loop variable i, n times at a cost of c_5.

7) Because this is a nested for loop, the outer loop will execute n times, and the inner loop will execute $n \times n$ times. The pattern is when $i = 1$, j loops from 1 to n once, or $1n$ times. When $i = 2$, j will loop from 1 to n again, or $2n$ times. It follows that when $i = n$, our number of iterations is $n \times n$. So this line will execute n^2 times. The cost is constant c_6.

8) This is the third level of a nested for loop, so as mentioned above our outer loop will execute n times, the second level will execute $n \times n$ times, and this level will execute $n \times n \times n$, or n^3 times. The pattern is when $i = 1$, and $j = 1$, k will loop from 1 to n once, or $1n$ times. When $i = 1$ and $j = 2$, k will loop from 1 to n again, or $2n$ times. When $i = 1$ and $j = n$, k will loop $1 \times n^2$ times. It follows that when $i = n$, our number of iterations is $n \times n^2$, or n^3. Our cost remains constant, c_7.

9) This condition is inside our nested loops, so it will be checked each iteration of loop k, minus the last iteration where we exit the loop. This implies line 9 will be executed $n^3 - 1$ times. The condition itself is adding two elements in the array, and comparing to another element in the array, each with constant time. Therefore this will still be a constant cost of c_8.

10) This line will only be executed the number of times the condition in line 9 holds true. We don't know how many times this will happen, so we suppose for loop variable k, the line is executed $\sum_{k=1}^{n} t_k$ times. Setting an element in the array equal to the addition of two other elements is constant time.

12) Our model doesn't include a definition for counting the cost of the return statement, and it could vary depending on how we define it. I've included a constant cost here with the number of times executed = 1, but pointing out that this isn't defined within our model. This is also consistent with lectures from the instructors, where this line was skipped all together.

Finally, our closed form summation is:

$$T(n) = \sum_{i=1}^{n}(c_2n^2 \times 1) + c_3n + c_4(n-1) + c_5n + c_6n^2 + c_7n^3 + c_8(n-1)^3 + c_9\sum_{k=1}^{n} t_k + c_{10}$$

$$T(n) = \sum_{i=1}^{n} c_2n^2 + c_3n + c_4n - c_4 + c_5n + c_6n^2 + c_7n^3 + c_8(n-1)^3 + c_9(n-1)^3 + c_{10}$$

Our worst-case scenario would be if the condition in line 9 were true for every iteration of loop variable k, and $\sum_{k=1}^{n} t_k = n^3 - 1$. However we can also see that for any non-empty array, we will enter all three levels of the nested for loop, which runs n^3 times. n^3 is the highest order term regarless of how many times $\sum_{k=1}^{n} t_k$ executes, and therefore $T(n) = \Theta(n^3)$.

(1b) Consider the following algorithm. Note that the inner loop variable j depends on the outer loop variable i. So it is not sufficient to multiply the complexities of the two loops together.

```
                                              | Cost     | # times run
1 g(A[1, ..., n]):
2   for i = 1 to len(A):                      | $c_2$    | $n$
3     for j = 1 to len(A)-i:                  | $c_3$    | $\sum_{i=1}^{n}(n-i)$
4       if A[j+1] > A[j]:                     | $c_4$    | $\sum_{i=1}^{n}(n-i)-1$
5         // swap A[j+1] and A[j]
6         tmp = A[j+1]                        | $c_5$    | $\sum_{j=1}^{n} t_j$
7         A[j+1] = A[j]                       | $c_6$    | $\sum_{j=1}^{n} t_j$
8         A[j] = tmp                          | $c_7$    | $\sum_{j=1}^{n} t_j$
9   return A                                  | $c_8$    | 1
```

1) The first for loop will run from 1 to n at a constant cost of c_2.

2) When $i = 1$, line 2 will run $(n-1)$ times. When $i = 2$, line 2 will run $(n-2)$ times. So the pattern is when $i = n$, line two will have run $(n-1)+(n-2)+(n-3)+...+(n-n)$. Note this is an arithmetic sequence with the formula

$$\sum_{i=1}^{n} a_i = \frac{n(a_1 + a_n)}{2}$$

, where a_1 is the first term of the sequence and a_n is the last term of the sequence. Therefore:

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

4) The condition in line 4 will be checked for each iteration of the inner loop j, minus the last iteration where we exit the loop. So this line will be executed $\sum_{i=1}^{n}(n-i)-1$ times. Checking the condition is constant cost c_4.

$6, 7, 8$) Lines 6, 7, and 8 will only be executed the number of times line 4 evaluates to true. We don't know how many times this will happen, so we suppose for loop variable j, the line is executed $\sum_{j=1}^{n} t_j$ times. It follows that lines 7 and 8 are executed the same number of times. All of the operations in these lines have a constant cost of c.

9) See response to line 12, part $1(a)$.

Now we have:

$$T(n) = \sum_{i=1}^{n} c_1 n + c_2 \frac{n(n-1)}{2} + c_3 \frac{n(n-1)}{2} - 1 + c_5 \sum_{j=1}^{n} t_j + c_6 \sum_{j=1}^{n} t_j + c_7 \sum_{j=1}^{n} t_j + c_8$$

$$T(n) = \sum_{i=1}^{n} \frac{n(n-1)}{2}(c_2 + c_3 + c_5 + c_6 + c_7) + c_1 n + c_8 - (c_3 + c_5 + c_6 + c_7)$$

Our worst-case scenario would be if the condition in line 4 is true for every iteration of loop j, which would imply $\sum_{j=1}^{n} t_j = \frac{n(n-1)}{2} - 1$. Substituting this into $T(n)$, we can see that our highest order term is n^2. Furthermore, we note that for any non-empty array the algorithm will enter the nested for loop and check the condition for line 4. Therefore, our highest order term will always be n^2 for any non-empty array, and we can conclude $T(n) = \Theta(n^2)$.

Name: $\boxed{\text{Jeremy M. Hein}}$

ID: $\boxed{810952267}$

CSCI 3104, Algorithms **Charlie Carlson & Ewan Davies**

Problem Set 3 – Due Sep 17 11.59pm MDT **Fall 2020, CU-Boulder**

Problem 2. Solve the following recurrence relations using the unrolling method (sometimes known as plug-in or substitution method). Show all work.

(2a) $T(n) = \begin{cases} 5\,T(n-2)+3 & : \text{if } n > 1, \\ 2 & : \text{otherwise} \end{cases}.$

$$\begin{aligned} T(n) &= 5T(n-2)+3 \\ &= 5(5T(n-4)+3)+3 \\ &= 5^2T(n-4)+(5\times 3)+3 \\ &= 5(5^2T(n-6)+(5\times 3)+3)+3 \\ &= 5^3T(n-6)+(5^2\times 3)+(5\times 3)+3 \end{aligned}$$

So it looks like our pattern is in the form $5^iT(n-2i)+\sum_{k=1}^{i} 3(5^{i-1})$. We'll hit the base case when $n=1$, so $n-2i=1 \implies i=\frac{n-1}{2}$ and $T(1)=2$. Substituting into our relation, we have:

$$\begin{aligned} T(n) &= 5^{\frac{n-1}{2}}(2)+3\sum_{k=1}^{i} 5^{i-1} && (1) \\ &= 5^{\frac{n-1}{2}}(2)+3\left(\frac{5^{\frac{n-1}{2}}-1}{5-1}\right) && (2) \\ &= 5^{\frac{n-1}{2}}(2)+\frac{3(5^{\frac{n-1}{2}})-3}{4} && (3) \\ &= \Theta(5^{\frac{n-1}{2}}) && (4) \end{aligned}$$

Recognize that (1) is a geometric series, so apply the formula $\frac{a(r^n-1)}{r-1}$ in step (2).

See in step (3) we have a $5^{\frac{n-1}{2}}$ in each term. So our highest order term must be $5^{\frac{n-1}{2}}$, and we can make our conclusion in step (4).

Name: $\boxed{\text{Jeremy M. Hein}}$

ID: $\boxed{810952267}$

CSCI 3104, Algorithms **Charlie Carlson & Ewan Davies**

Problem Set 3 – Due Sep 17 11.59pm MDT **Fall 2020, CU-Boulder**

(2b) $T(n) = \begin{cases} 7\,T(n/2) + \Theta(n) & : \text{if } n > 1, \\ \Theta(1) & : \text{otherwise} \end{cases}.$

$$\begin{aligned}
T(n) &= 7T(\frac{n}{2}) + cn && \text{(replace } \Theta(n) \text{ w/ } cn\text{, where } c > 0\text{)} \\
&= 7(7T(\frac{n}{4}) + \frac{cn}{2}) + cn \\
&= 7^2T(\frac{n}{4}) + \frac{7cn}{2} + cn \\
&= 7(7^2T(\frac{n}{8}) + \frac{7cn}{4} + \frac{cn}{2}) + cn \\
&= 7^3T(\frac{n}{8}) + \frac{7^2cn}{4} + \frac{7cn}{2} + cn
\end{aligned}$$

So it looks like our pattern is in the form $7^iT(\frac{n}{2^i}) + \sum_{k=1}^{i}(\frac{7}{2})^{i-1}cn$. We'll hit the base case when $n = 1$, so $\frac{n}{2^i} = 1 \implies i = log(n)$ and $T(1) = c$. Substituting into our relation, we have:

$$\begin{aligned}
T(n) &= 7^{log(n)}c + cn\sum_{k=1}^{i}\left(\frac{7}{2}\right)^{i-1} && (1) \\
&= 7^{log(n)}c + cn\left(\frac{\frac{7^{\log(n)}}{2^{\log(n)}} - 1}{\frac{7}{2} - 1}\right) && (2) \\
&= 7^{log(n)}c + cn\left(\frac{\frac{7^{\log(n)}}{n} - 1}{\frac{5}{2}}\right) && (3) \\
&= 7^{log(n)}c + \frac{7^{\log(n)}c - cn}{\frac{5}{2}} && (4) \\
&= 7^{log(n)}c + \frac{2(7^{\log(n)}c - cn)}{5} && (5) \\
&= \Theta(7^{log(n)}) && (6)
\end{aligned}$$

Recognize that (1) is a geometric series, so apply the formula $\frac{a(r^n-1)}{r-1}$ in step (2).

Step (3) is applying the logarithm rule $2^{\log_2(n)} = n$

Step (5) is a polynomial of the highest degree $7^{\log_2(n)}$. Note that $7^{\log_2(n)} = n^{\log_2(7)} = n^{\frac{\log(7)}{\log(2)}} > n$.

Name: Jeremy M. Hein
ID: 810952267

CSCI 3104, Algorithms | **Charlie Carlson & Ewan Davies**
Problem Set 3 – Due Sep 17 11.59pm MDT | **Fall 2020, CU-Boulder**

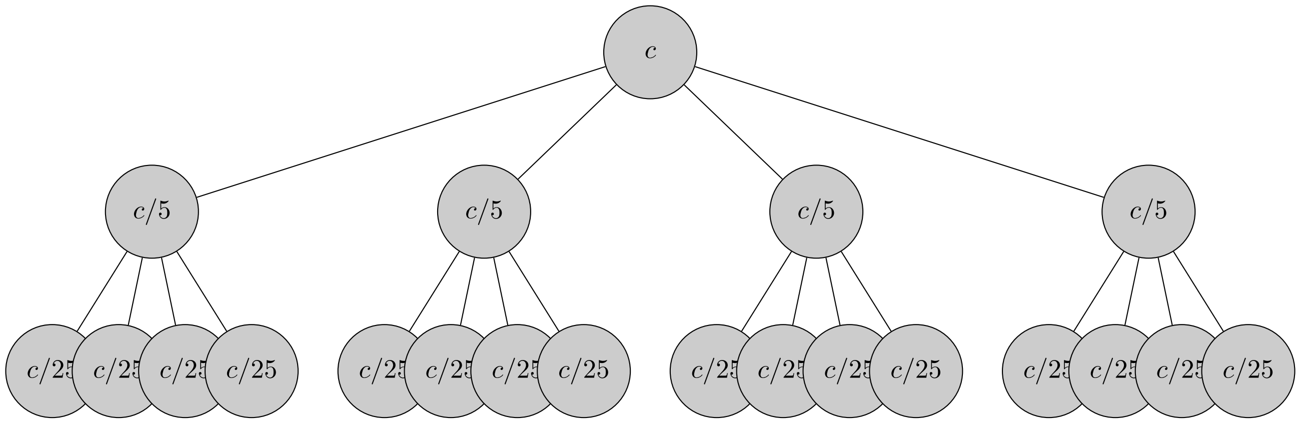
Problem 3. Consider the following functions. For each of them, determine how many times is 'hi' printed in terms of the input n. You should first write down a recurrence and then solve it **using the recursion tree method.** That means you should write down the first few levels of the recursion tree, specify the pattern, and then solve. **Your final answer should be an asymptotic expression in terms of Big-Theta. That is, you do not need to give a precise numerical count depending on n.**

(3a)
```
def fun(n) {
     if (n > 1) {
        print( ‘hi’ ‘hi’ ‘hi’ )
        fun(n/5)
        fun(n/5)
        fun(n/5)
        fun(n/5)
}}
```

Let $T(n)$ be the run time on an input of size n. Then we have:

$$T(n) = \begin{cases} 4\,T(n/5) + \Theta(1) & : \text{if } n \geq 2, \\ \Theta(1) & : \text{otherwise} \end{cases}.$$

Note that the cost of the condition if $(n > 1)$ is constant run time and the cost of the line print('hi' 'hi' 'hi') is also constant run time. So the non-recursive cost is $\Theta(1)$. We have four recursive calls, each with a cost of $T(n/5)$.



The top node has constant cost c because the first call to the function does constant units of work, and there haven't been any recursive calls yet. The nodes on the second layer all have cost $\frac{c}{5}$ because there are four recursive calls on problems of size $n/5$. This pattern continues until we reach our base case.

First we find the height of the tree. We can see from our tree above that the subproblem size at level $i = \frac{n}{5^i}$, and we will hit $n = 1$ when $\frac{n}{5^i} = 1$, or $i = \lceil \log_5(n) \rceil$ levels.

Now we need to find the cost of each level of the tree. At level i, we have 4^i nodes at a cost of $\frac{c}{5^i}$. So the cost for each level is $\left(\frac{4}{5}\right)^i c$. At our last level, we have $4^{\log_5(n)} = n^{log_5(4)}$ nodes at cost T(1). Lastly we sum all of the levels together:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_5(n)-1} \left(\frac{4}{5}\right)^i c + \Theta(n^{\log_5(4)}) && (1)\\
&= c\left(\frac{1-\left(\frac{4}{5}\right)^{\log_5(n)}}{1-\frac{4}{5}}\right) + \Theta(n^{\log_5(4)}) && (2)\\
&= 5c\left(1-\frac{4^{\log_5(n)}}{5^{\log_5(n)}}\right) + \Theta(n^{\log_5(4)}) && (3)\\
&= 5c\left(1-\frac{n^{\log_5(4)}}{n}\right) + \Theta(n^{\log_5(4)}) && (4)\\
&= \Theta(n^{\log_5(4)}) && (5)
\end{aligned}$$

1) Note this is the sum of a geometric series from $i = 0$ to $\log_5(n) - 1$

2) Here $a = 1$ and $r = \frac{4}{5}$, and apply the formula $\frac{a(1-r^n)}{1-r}$.

3) Logarithm properties $5^{log_5(n)} = n$ and $4^{log_5(n)} = n^{log_5(4)}$

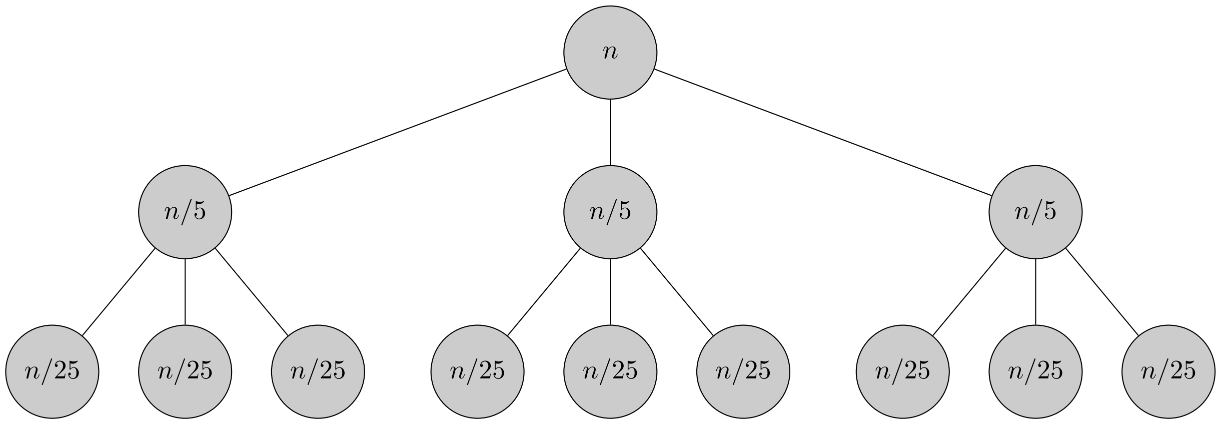
4) Note that $\log_5(4) = \frac{\log(4)}{\log(5)} < 1 \implies n > n^{log_5(4)}$. So that means the term $\frac{n^{\log_5(4)}}{n} < 1$, and so our largest term must be $\Theta(n^{\log_5(4)})$.

(3b)
```
def fun(n) {
     if (n > 1) {
        for i=1 to n {
          print( 'hi' 'hi' )
        }
        fun(n/5)
        fun(n/5)
        fun(n/5)
}}
```

Let $T(n)$ be the run time on an input of size n. Then we have:

$$T(n) = \begin{cases} 3\,T(n/5) + \Theta(n) & : \text{if } n \geq 2, \\ \Theta(1) & : \text{otherwise} \end{cases}.$$

Note that before our recursive calls, we loop from 1 to n and print ('hi' 'hi'), so our non-recursive cost is $\Theta(n)$. We have three recursive calls, each with a cost of $T(n/5)$.



The top node has constant cost n because the first call to the function does n units of work, and there haven't been any recursive calls yet. The nodes on the second layer all have cost $\frac{n}{5}$ because there are three recursive calls on problems of size $n/5$. This pattern continues until we reach our base case.

First we find the height of the tree. We can see from our tree above that the subproblem size at level $i = \frac{n}{5^i}$, and we will hit $n = 1$ when $\frac{n}{5^i} = 1$, or $i = \lceil \log_5(n) \rceil$.

Now we need to find the cost of each level of the tree. At level i, we have 3^i nodes at a cost of $\frac{n}{5^i}$ for each node. So the cost for each level is $\left(\frac{3}{5}\right)^i n$. At our last level, we have $3^{\log_5(n)} = n^{log_5(3)}$ nodes at cost T(1). Lastly we sum all of the levels together:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_5(n)-1} \left(\frac{3}{5}\right)^i n + \Theta(n^{log_5(3)}) && (1)\\
&= n\left(\frac{1-\left(\frac{3}{5}\right)^{\log_5(n)}}{1-\frac{3}{5}}\right) + \Theta(n^{log_5(3)}) && (2)\\
&= \frac{5}{2}n\left(1-\frac{3^{\log_5(n)}}{5^{\log_5(n)}}\right) + \Theta(n^{log_5(3)}) && (3)\\
&= \frac{5}{2}n\left(1-\frac{n^{\log_5(3)}}{n}\right) + \Theta(n^{log_5(3)}) && (4)\\
&= \frac{5}{2}n\left(1-n^{\log_5(3)-1}\right) + \Theta(n^{log_5(3)}) && (5)\\
&= \Theta(n) && (6)
\end{aligned}$$

1) Note this is the sum of a geometric series from $i = 0$ to $\log_5(n) - 1$

2) Here $a = 1$ and $r = \frac{4}{5}$, and apply the formula $\frac{a(1-r^n)}{1-r}$.

4) Logarithm properties $5^{log_5(n)} = n$ and $4^{log_5(n)} = n^{log_5(4)}$

5) Note that $\log_5(3) = \frac{\log(3)}{\log(5)} < 1 \implies n > n^{log_5(3)} > n^{log_5(3)-1}$. So we can conclude that n is our largest term and thus $T(n) = \Theta(n)$.

Name: Jeremy M. Hein

ID: 810952267

CSCI 3104, Algorithms **Charlie Carlson & Ewan Davies**
Problem Set 3 – Due Sep 17 11.59pm MDT **Fall 2020, CU-Boulder**

Problem 4. Given an array $A = [a_1, a_2, \ldots, a_n]$, 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 $\mathcal{O}(n \log n)$ 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:

1. Your algorithm has to be a divide and conquer algorithm that is modified from the mergesort implementation given as Algorithm 1 and Algorithm 2 below. Recall that pseudocode from the lecture notes (sometimes) uses 1-based indexing and inclusive range notation which may differ from how Python 3 works. You must explain how your algorithm works, including pseudocode where you specify any necessary semantics of *your* pseudocode such as 0-based or 1-based indexing and whether ranges are inclusive.
2. Implement your algorithm in Python 3. **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.**
3. 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.

Name: Jeremy M. Hein
ID: 810952267

CSCI 3104, Algorithms **Charlie Carlson & Ewan Davies**
Problem Set 3 – Due Sep 17 11.59pm MDT **Fall 2020, CU-Boulder**

Algorithm 1 Pseudocode for merge.

Input: Two sorted lists A and B (in ascending order)
Output: A sorted list containing the elements of both A and B

```
procedure MERGE(A,B)
    let $C$ be a new array of length len($A$) + len($B$)
    let $i = 1$ and $j = 1$
    while $i + j \leq$ len($A$) + len($B$) :
        if $i >$ len($A$) :                     #true $\iff$ we have copied all of $A$
            $C[i + j - 1] = B[j]$
            $j = j + 1$
        else if $j >$ len($B$) :                #true $\iff$ we have copied all of $B$
            $C[i + j - 1] = A[i]$
            $i = i + 1$
        else if $A[i] < B[j]$ :
            $C[i + j - 1] = A[i]$
            $i = i + 1$
        else
            $C[i + j - 1] = B[j]$
            $j = j + 1$
    return $C$
```

Algorithm 2 Pseudocode for mergesort.

Input: List A
Output: The list A sorted in ascending order

```
procedure MERGESORT(A)
    $n$ = len($A$)
    if $n \leq 1$ :
        return $A$
    else
        $H_1$ = MERGESORT($A[1..n/2]$)
        $H_2$ = MERGESORT($A[n/2 + 1..n]$)
        return MERGE($H_1, H_2$)
```