Solution: Asymptotic Notations and Correctness of Algorithms

1. **Identify asymptomatic notation of a function:** Use the definitions of O, Ω and Θ to identify if the following statements are true or false. Prove your assertion by solving for the values of c and n_o. Draw a rough graph representing the c and n_o values if the statement is True.
   Notes: Used the steps provided in **this Ed discussion** to solve this problem.

For lower bound:

- remove a positive term from f(n)

$$3n^2 + 5n \geq 3n^2$$

- multiply a negative lower order term in f(n)

$$5n^2 - 2n \geq 5n^2 - 2n \times n = 3n^2$$

- split a higher order term if needed

$$2n^3 + 1 = n^3 + n^3 + 1 \geq n^3$$

For upper bound:

- remove a negative term in f(n)

$$5n^2 - 3n \leq 5n^2$$

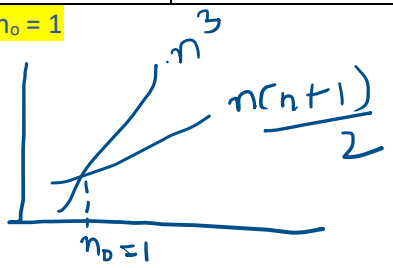
- multiply a positive lower order term in f(n)

$$3n^2 + n \leq 3n^2 + n \times n = 4n^2$$

   a. $n(n+1)/2 \in O(n^3)$ **True**

$n^2 \leq n^3 \quad \forall n \geq 1$	[Take higher order terms that we want in our inequality]
$\frac{n^2}{2} \leq \frac{n^3}{2} \quad \forall n \geq 1$	[Dividing by two on both sides, since original equation has /2]
$\frac{n^2}{2} + \frac{n}{2} \leq \frac{n^3}{2} + \frac{n^3}{2}$	[Since $\frac{n}{2} \leq \frac{n^3}{2}$, "Multiply positive lower order term", to get original equation]
$\frac{n(n+1)}{2} \leq n^3 \quad \forall n \geq 1$	

c = 1 , n_o = 1



b. $n(n+1)/2 \in \Theta(n^2)$ **True**

Proving Upper Bounds

$n^2 = n^2$	Take higher order terms that we want to prove
$\frac{n^2}{2} = \frac{n^2}{2}$	Modify it to get it in the form of f(n)'s higher order term
	Remaining term in f(n) is $\frac{n}{2}$ If we add n/2 to Left Hand Side (LHS) then LHS will become big. To make Right Hand Side bigger than LHS. Since $\frac{n}{2} \leq \frac{n^2}{2}$, "Multiply positive lower order term", to get original equation. This is true for all n >= 1
$\frac{n^2}{2} + \frac{n}{2} \leq \frac{n^2}{2} + \frac{n^2}{2} \quad \forall n \geq 1$	
$\frac{n(n+1)}{2} \leq n^2 \quad \forall n \geq 1$	

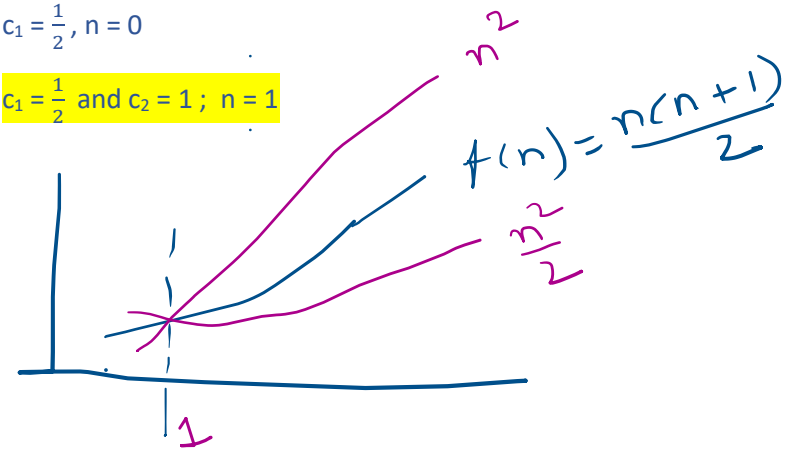
$c_2 = 1$, $n = 1$

Proving Lower Bounds

$\frac{n^2}{2} + \frac{n}{2} = \frac{n^2}{2} + \frac{n}{2}$	Take the term as is
	To get $\leq$ inequality "Remove a lower order positive term"
$\frac{n^2}{2} \leq \frac{n^2}{2} + \frac{n}{2} \quad \forall n \geq 0$	$\frac{n}{2}$ should be positive to remove it. $\Rightarrow \frac{n}{2} \geq 0 \Rightarrow n \geq 0$
$\frac{n^2}{2} \leq \frac{n(n+1)}{2} \quad \forall n \geq 0$	

$c_1 = \frac{1}{2}$, $n = 0$

$c_1 = \frac{1}{2}$ and $c_2 = 1$; $n = 1$



c. $n(n+1)/2 \in \Theta(n^3)$ **False**

This is not true. To prove this, you can take limits on f(n)/g(n). Here $g(n) = n^3$

$$\lim_{n\to\infty} \frac{\frac{n(n+1)}{2}}{n^3}$$

$$= \frac{1}{2} \lim_{n\to\infty} \frac{n^2+n}{n^3}$$

$$= \frac{1}{2} \lim_{n\to\infty} \frac{2n+1}{3n^2}$$ Apply L'Hospital rule

$$= \frac{1}{2} \lim_{n\to\infty} \frac{2}{6n}$$ Apply L'Hospital rule

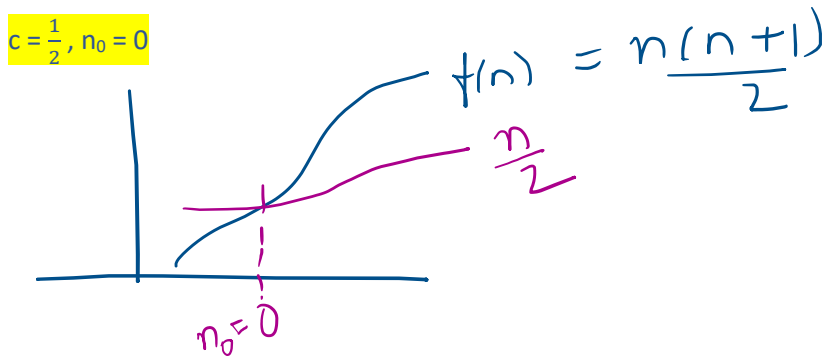
$$= \frac{1}{2}\left(\frac{2}{\infty}\right) = \infty$$

$\Rightarrow n(n+1)/2 \in O(n^3)$ Not Theta.

d. $n(n+1)/2 \in \Omega(n)$

$\frac{n^2}{2} + \frac{n}{2} = \frac{n^2}{2} + \frac{n}{2}$	Take the term as is
	To get $\leq$ inequality "Remove a lower order positive term or any positive term to get g(n)". Here we can remove $\frac{n^2}{2}$ to get n in the LHS
$\frac{n}{2} \leq \frac{n^2}{2} + \frac{n}{2}$ $\forall n \geq 0$	$\frac{n^2}{2}$ should be positive to remove it. $\Rightarrow \frac{n^2}{2} \geq 0 \Rightarrow n \geq 0$

$c = \frac{1}{2}$, $n_0 = 0$



2. **Compare order of growth**: For each of the following, indicate whether f = O(g), f = Ω (g) or f = Θ (g).

   a. f(n) = 10n-6, g(n) = 12345678n + 2020

$$\lim_{n\to\infty} \frac{10n-6}{12345678n+2020}$$

$$= \lim_{n\to\infty} \frac{10}{12345678}$$ Applying L'Hospital rule

= Constant

$\Rightarrow$ f(n) $\in \Theta$ (g(n))

   b. f(n) = n!, g(n) = 0.00001n

$$\lim_{n\to\infty} \frac{n!}{0.00001n}$$

$$= \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{0.00001n}$$ using Stirling's Formula

$$= \frac{\sqrt{2\Pi}}{0.00001} \lim_{n\to\infty} \frac{\sqrt{n}}{n}\left(\frac{n}{e}\right)^n$$

$$= \frac{\sqrt{2\pi}}{0.0000^1} \lim_{n\to\infty} \frac{\sqrt{n}}{(e)^n} x \frac{n^n}{n}$$

$$= \frac{\sqrt{2\pi}}{0.00001} \lim_{n\to\infty} \frac{\sqrt{n}}{e^n} x^{n^{n-1}}$$

$$= (something) * (\infty)^{\infty} = \infty$$

$\Rightarrow$ f(n) $\in \Omega$ (g(n))

   c. f(n) = $n^{1.34}$, g(n) = log n

$$\lim_{n\to\infty} \frac{n^{1\cdot 34}}{log\, n}$$

$$= \lim_{n\to\infty} 1 \cdot \frac{34 \times n^{0.34}}{log_2^e\left(\frac{1}{n}\right)}$$

$$= \frac{1\cdot 34}{log_2^e} \lim_{n\to\infty} n^{0.34} \times n = \infty$$

$\Rightarrow$ f(n) $\in \Omega$ (g(n))

d. $f(n) = n + n^4$, $g(n) = n^3 + \log n$

$$\lim_{n\to\infty} \frac{n+n^4}{n^3+\log n}$$

$$= \lim_{n\to\infty} \frac{1+4n^3}{3n^2+\frac{1}{n\log_e 2}}$$

$$= \lim_{n\to\infty} \frac{1+4n^3}{3n^3\times\log_e 2+1} * n\log_e 2$$

$$= \lim_{n\to\infty} \frac{12n^3\log_e 2}{9n^2\log_e 2}$$

$$= \lim_{n\to\infty} \frac{12n}{9\log_e 2} = \infty$$

$\Rightarrow f(n) \in \Omega(g(n))$

3. **Identify asymptomatic notation of an operation:**
If you want to search for a value 'x' in the following data structures and return True if the value is present otherwise return False. What would be the time complexity for this operation? Assume that the values are sorted. (Use an appropriate notation among O, Ω and Θ to mention the time complexity). Provide an explanation for your answer.
   a. Array

   To search for a value in an array if we are checking each element one after another starting from the first element, then in the worst case we have to check the entire array. This would be equal to the size of the array. So, in the worst-case time complexity for this search operation will be O(n), where n is the size of the array.

   b. Linked List

   The same logic applies that was used for searching an element in an array applies to the linked list. Worst case time complexity will be O(n).

4. **Read and Analyse Pseudocode:** Consider the following algorithm

```
Classified(A...n-1):
    minval = A[0]
    maxval = A[0]
    for i = 1 to n-1:
        if A[i] < minval:
            minval = A[i]
        if A[i] > maxval
            maxval = A[i]
    return maxval – minval
```

   a. What does this algorithm compute? The difference between the largest and the smallest element in an array.
   b. What is its basic operation (i.e. the line of code or operation that is executed maximum number of times)? The if conditions
   `if A[i] < minval; if A[i] > maxval`

c. How many times is the basic operation executed?
$\sum_{i=1}^{n-1} 2 = 2 \cdot (n-1)$

d. What is the time complexity of this algorithm?
$\theta(n)$

5. **Using mathematical induction prove below non-recursive algorithm:**
Any number greater than 8 can written in terms of three or five.

a. Write a pseudocode of algorithm that that takes a number greater than 8 and returns a tuple (x,y); where x represents number of threes and y represent number of fives make that number

If number = 8 your pseudocode should return (1,1)

```
def ThreeAndFive(n):
    t=1, f=1, x=8 # to begin with 3t+5f= 8; one three coin, and
one five coin to make 8, minimum possible amount

    while(x<n):
        x = x+1 #solving for x+1 amount
        if we have at least one 5 coin already, we can remove it
add two 3 coins

        else we can remove three coins and add two five coins
        #We will have at least three 3 coins, if we have only two
3*2=6 not possible as min amount is 8.

    return t, f
```

b. Prove correctness of your pseudocode using induction

Loop Invariant: 3t + 5f will make the amount x. (x is the amount that current iteration is solving for)

Base case: Before the loop starts 3t+5f = 3*1+5*1 = 3+5 = 8. Before the loop begins x = 8. The loop invariant holds true.

Inductive Case: Assume that invariant is true for k^{th} iteration. [Which means we have solved for 8+k amount; 3t+5f = 8+k]. Let us see what happens in the $(k+1)^{th}$ iteration. Since we have if – else condition in the code. We can take two cases.

Case 1 (if condition execution): When we have at least one 5 coin. The if condition in the code will be executed. In if condition we are removing one 5 coin and adding two 3 coins. Two 3 coins = 6, which be thought as adding (6-5 = 1) a total of 1 amount more the coins total worth. This correctly satisfies the increment of 1 in the value of x. So, 3t+5f = 8+k+1, our new amount for the $(k+1)^{th}$ iteration.

Case 2 (else condition execution): When we don't have any 5 coins but all three coins; This also means we have atleast three 3 coins, because two 3 coins = 6, 6 is not possible amount as we have a minimum of 8.

We are removing three 3-coins and adding two 5-coins; 2*5-3*3 = 1; an increment of 1 in our total coins worth. This correctly satisfies the increment of 1 in the value of x. So, 3t+5f = 8+k+1, our new amount for the $(k+1)^{th}$ iteration.

In both if and else conditions the loop invariant is maintained.

Termination: The Loop terminates when x = n; in the previous iteration of the loop, we have calculated for an amount of x+1, which was (n); So we are done will finding coins for n amount.

c. What is the time complexity of your pseudocode?
The while loop runs for a maximum of (n-8) number of times. Giving a time complexity of theta(n)

d. Code your pseudocode into python and name your file ThreeAndFive.py

```
def ThreeAndFive(n):
    t=1
    f=1 #to begin with 3t+5f= 8; one three coin, and one five
coin
    x= 8 # n >= 8;

    while(x<n):
        x = x+1 # in current iteration we solving for x+1 amount
        if(f>=1): # we remove a 5 coin and add two three coins
            f = f - 1
            t = t + 2

        else: # we can remove 3 three coins and add two five
coins
            f = f + 2
            t = t - 3

    return t, f
```

6. **Visualizing time complexity for best case, worst case and average case**:
   a. Implement insertion sort to sort an array of numbers in descending order. Name your function insertionsort and your file EvaluateInsertionSort.py.
   b. Modify your code to add another function timeSortingRandomNumbers() that creates arrays of different sizes ranging from 0 to 10,000 with random unsorted numbers and collects the time that insertionsort function takes to sort the arrays. Collect at least 10 running times.
   c. Modify your code to add another function timeSortingAscendingNumbers() that creates arrays of different sizes ranging from 0 to 10,000 with numbers sorted in ascending

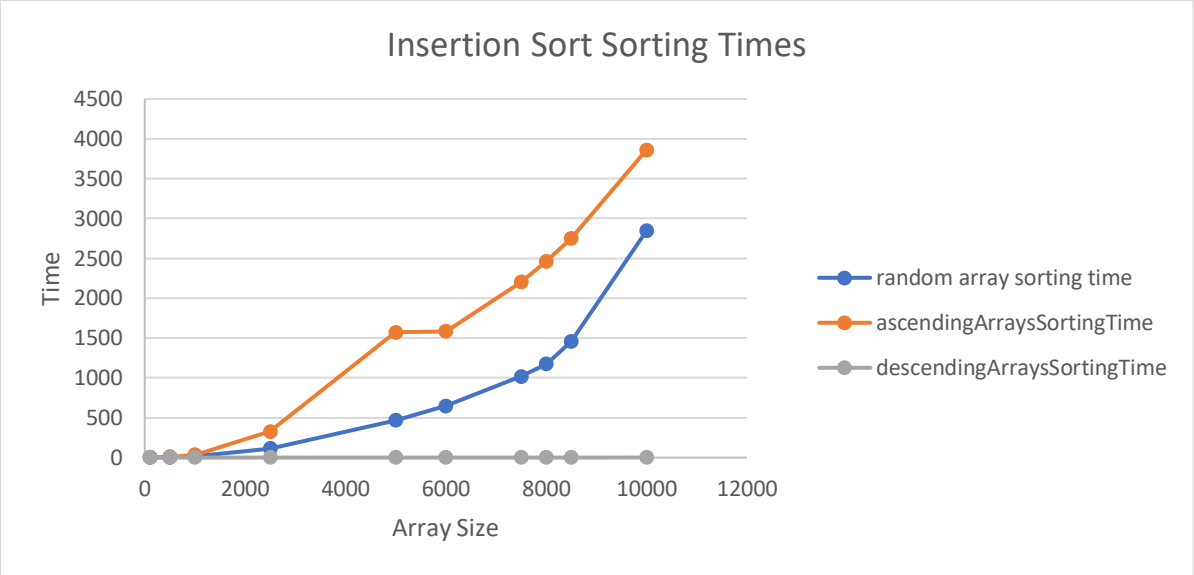
order and collects the time that insertionsort function takes to sort the arrays. Collect at least 10 running times.

d. Modify your code to add another function timeSortingDescendingNumbers() that creates arrays of different sizes ranging from 0 to 10,000 with numbers sorted in descending order and collects the time that insertionsort function takes to sort the arrays. Collect at least 10 running times.

```
# Insertion sort in Python
import random
import time

def insertionsort(array):
    for step in range(1, len(array)):
        key = array[step]
        j = step - 1
        while j >= 0 and key > array[j]:
            array[j + 1] = array[j]
            j = j - 1
        array[j + 1] = key


def timeSortingRandomNumbers(arraysize):
    randArray = []
    #create array of random numbers
    for i in arrysizes:
        randArray.append([random.randint(0, 10000) for x in range(0,
i)])

    #collect sorting time
    sortingTime = []
    for i in randArray:
        tic = time.perf_counter()
        insertionsort(i)
        toc = time.perf_counter()
        #print("arraysize: ", len(i),  " timetaken: ",  (toc-tic)*1000
)
        sortingTime.append("{:0.2f}".format((toc-tic)*1000))

    return sortingTime

def timeSortingAscendingNumbers(araysize):

    ##create array of random numbers in ascending order
    ascendingArray = []
    for i in arrysizes:
        rand = [random.randint(0, 10000) for x in range(0, i)]
        rand.sort()
        ascendingArray.append(rand)

    # collect sorting time
    sortingTime = []
    for i in ascendingArray:
        tic = time.perf_counter()
        insertionsort(i)
        toc = time.perf_counter()
```

```
        # print("arraysize: ", len(i),  " timetaken: ",  (toc-tic)*1000
)
        sortingTime.append("{:0.2f}".format((toc - tic) * 1000))

    return sortingTime


    return ascendingArray

def timeSortingDescendingNumbers(arraysize):

    ##create array of random numbers in descinding order
    descendingArray = []
    for i in arrysizes:
        rand = [random.randint(0, 10000) for x in range(0, i)]
        rand.sort(reverse=True)
        descendingArray.append(rand)

    # collect sorting time
    sortingTime = []
    for i in descendingArray:
        tic = time.perf_counter()
        insertionsort(i)
        toc = time.perf_counter()
        # print("arraysize: ", len(i),  " timetaken: ",  (toc-tic)*1000
)
        sortingTime.append("{:0.2f}".format((toc - tic) * 1000))

    return sortingTime


if(__name__ == "__main__"):
    arrysizes = [100, 500, 1000, 2500, 5000,  6000, 7500, 8000, 8500,
10000]

    randArraySortingTime = timeSortingRandomNumbers(arrysizes)
    ascendingArraysSortingTime = timeSortingAscendingNumbers(arrysizes)
    descendingArraysSortingTime =
timeSortingDescendingNumbers(arrysizes)
```

e. Plot the running time obtained 6.b, 6.c, 6.d using excel. On y axis: running time and on X axis: array sizes



f. What is your observation about the insertion sort algorithm on random array, array with numbers in ascending order and array with numbers in descending order?

When an array that is already sorted in ascending order is sorted in descending order by the insertion sort; this has taken the maximum time for the same number of elements. This looks like the worst-case scenario for this problem where almost all the elements are not in their correct positions. This justifies that it took maximum time in this case.

When the arrays are already sorted in the descending order, the insertionSort function does not have to do any work; hence the time taken is almost constant. This shows the best-case scenario for this problem where not much work has to be done by the function.

When the arrays contain random numbers. The time taken is between the best case and worst case. This shows the average case of solving this problem. This graph curve lies between other two curves.