Problem #1

Let $f(n)$ and $g(n)$ be two functions, defined for each positive integer n.

By definition, $f(n) = O(g(n))$ if there exist positive constants *c* and n_0 for which $0 \leq f(n) \leq c * g(n)$ for all integers $n \geq n0$. To prove that $f(n) \neq O(g(n))$ one must prove that no such constants *c* and n_0 exist.

(a) Let $f(n) = n^2 + 2n + 3$. Prove that $f(n) = O(n^2)$ and $f(n) \neq O(n)$.

Answer (a)

We're given :

- $f(n) = n^2 + 2n + 3$
- $g(n) = n^2$

To prove :

- $f(n) = O(n^2)$ and $f(n) \neq O(n)$

$Proof$:

As stated, to prove that $f(n)$ is of order n^2 we must make sure that $\forall n \geq n_0,\ g(n)$ scaled by any given constant *c* overtakes the function $f(n)$.

For the proof, let's assume $n_0 = 5$, hence we have to prove the inequality to be true $\forall n \geq 5$.

$$\text{from the assumption,}$$
$$n \geq 5$$
$$\Rightarrow n^2 \geq 5$$
$$\text{from equations (2) and (3)}$$
$$n^2 \geq n \geq 5$$

Let's multiply the $equation(1)$ by 2 to get different terms of $f(x)$

We get,

$$2n \geq 10$$
$$2n^2 \geq 2n$$
$$\Rightarrow 2n^2 \geq 2n \geq 10$$

Similarly,

$$3n \geq 6$$
$$3n^2 \geq 3n$$
$$\Rightarrow 3n^2 \geq 3n \geq 15$$

Now we can combine $(3), (4)$ and (5) and get the following :

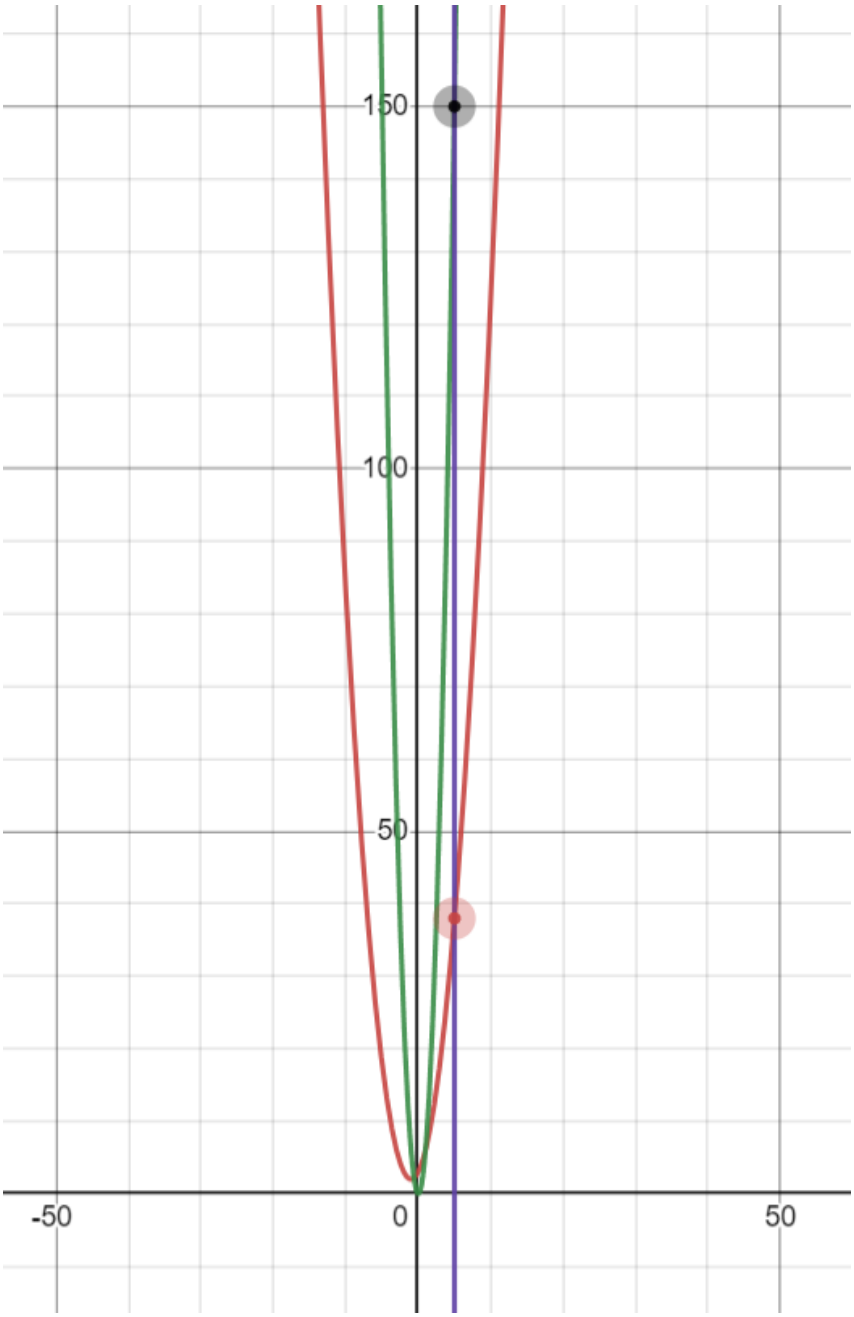
$$3n^2 + 2n^2 + n^2 \geq 3n + 2n + n \geq 15 + 10 + 5$$

$$\text{from } (3),(4),(5) \text{ and } (6)$$
$$\Rightarrow 3n^2 + 2n^2 + n^2 \geq n^2 + 2n + 3$$
$$= n^2 + 2n + 3 \leq 6n^2$$
$$\because f(n) = O(g(n)), \text{ if } 0 \leq f(n) \leq c * g(n)$$

$\text{when c} = 6, n^2 + 2n + 3 \leq 6 * n, \text{ as per } (7)$

Therefore it is proved that $f(x) = n^2 + 2n + 3$ is of order $O(n^2)$.

Here we can also see points at which the function $6n^2$ exceeds $f(x)$



Now to disprove that $f(n) \neq O(n)$

Let's assume that $f(n)$ is indeed $O(n)$ $\forall n \geq n_0$, where $n_0 = 1$.

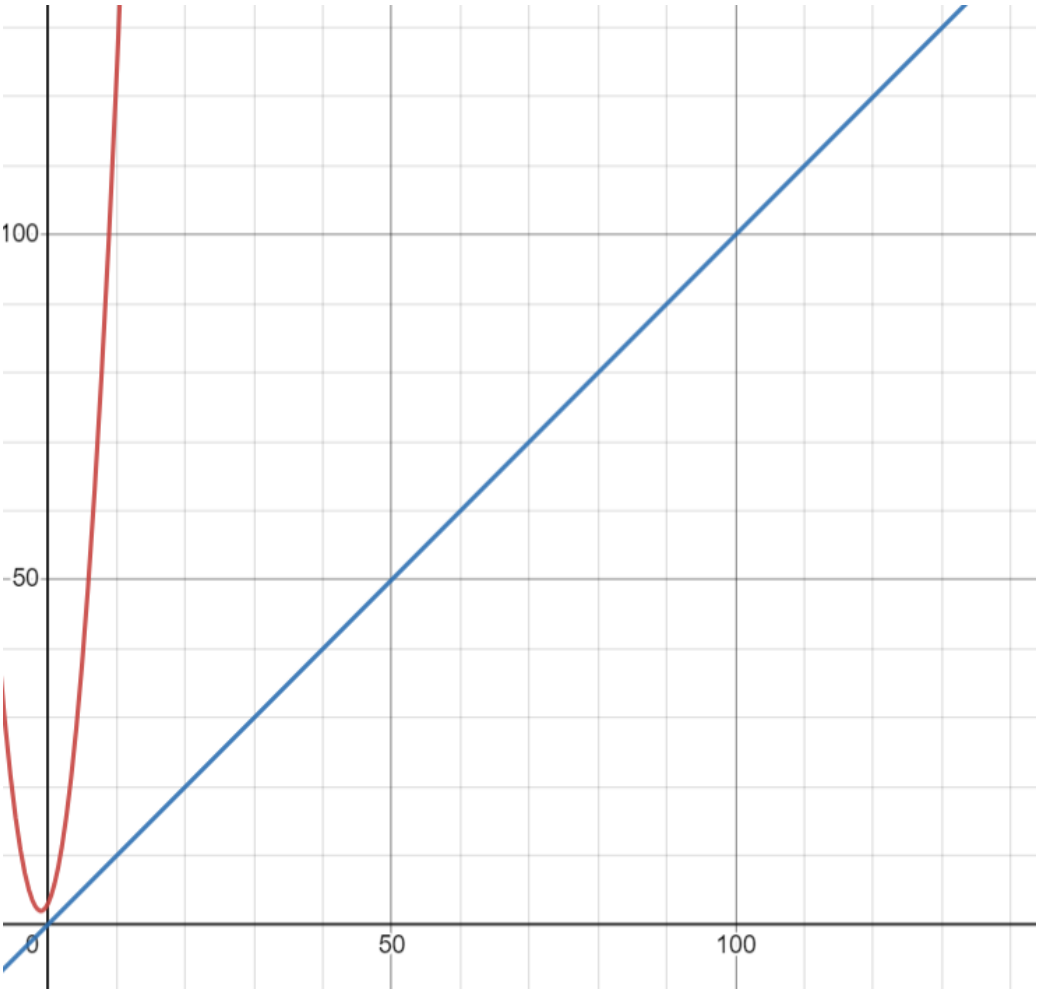
Therefore,

$n^2 + 2n + 3 \leq c * n$
Some constant c should exist where the above is true

It is understood that $n^2 \leq n^2 + 2n + 3$
Extrapolating this to our assumption
$\Rightarrow n^2 \leq c * n$
$= n \leq c$

Hence, this is a direct contradiction to our assumption since the assumption must hold true for $\forall n \geq n0 = 1$.

So our assumption must be false.



We can see in the image above that $f(n)$ scales much higher than $g(n) = n$.

(b) Let $f(n) = nlogn + 100n$. Prove that $f(n) = O(nlogn)$ and $f(n) \neq O(n)$.

Answer 1 (b)

We're given:

$f(n) = nlogn + 100n$

$g(n) = nlogn$

To Prove:

$f(n) = O(nlogn)$

Proof:

Let's assume n_0 = 1

$$\Rightarrow n \geq 1$$
$$\Rightarrow nlogn \geq logn$$

It is understood that $\forall n \geq 1$

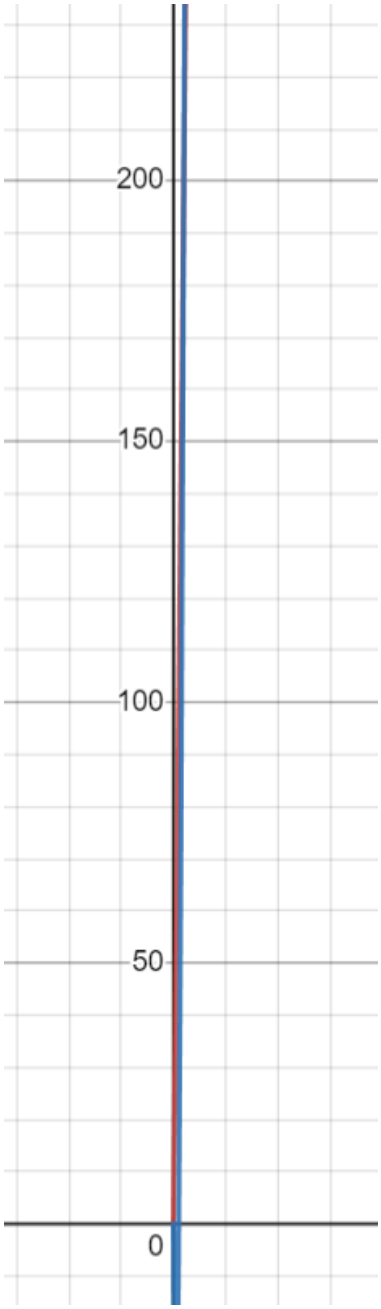
$$nlogn \geq n$$
$$\Rightarrow nlogn \geq n \geq 1$$
$$2(nlogn) \geq nlogn$$

Adding equation (1)

$$2(nlogn) + 100(nlogn) \geq nlogn + 100n$$
$$nlogn + 100n \leq 102(nlogn)$$

$c = 100, n_0 = 1$

$\therefore Proved$



Now to prove that $f(n)$ is not $O(n)$

Let's assume that it is $O(n)$, $\forall n \geq n_0 = 1$. Then,

$$nlogn + 100n \leq c * n$$
$$\Rightarrow logn + 100 \leq c$$
$$logn \leq c - 100$$
$$n \leq e^{c-100}$$

For any positive value of c, say $c = 1$, $e^{-99} = 1.0112215e - 43$, which is less than 1. Since we assumed the proposition to be true for $n \geq 1$, this is a contradiction.

1(c) Let $f(n) = 2n^2 + 4$ and $g(n) = 4n^2 + 2$.

Prove that $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Answer 1 (b)

We're given:

$f(n) = 2n^2 + 4$

$g(n) = 4n^2 + 2$

To Prove:

$f(n) = O(g(n))$

$g(n) = O(f(n))$

Proof

Let's assume $n \geq n_0 = 1$

Let's also assume that $f(n) \neq O(g(n))$ and $g(n) \neq O(f(n))$

If so, then $\forall n,\ 2n^2 + 4 \leq c * (4n^2 + 2)$ is invalid

However, for $c \geq 1$, we can observe that the equation is indeed valid. For example, if we fix $c = 2$:

$$2n^2 + 4 \leq 2(4n^2 + 2)$$
$$2n^2 + 4 \leq 8n^2 + 4$$

Similarly, we can show that $g(n) = O(f(n))$

$$4n^2 + 4 \leq 2(2n^2 + 4)$$
$$4n^2 + 4 \leq 4n^2 + 8$$

Hence Proved.

1(d) Let $f(n)$ and $g(n)$ be any two functions for which $f(n)$ and $g(n)$ are positive numbers, for each integer n ≥ 1.

Prove or disprove: at least one of these two statements must be true: $f(n) = O(g(n))$, $g(n) = O(f(n))$. Clearly and carefully justify your answer.

I will attempt to disprove that *both* statements are true through counterexample.

Let's assume the first statement $f(n) = O(g(n))$ is true. Then, let's also assume that $f(n) = n^2, g(n) = n^{10}$.

We can easily prove that $f(n) = O(g(n))$ since $n^2 < n^{10}$.

Now let's try the other statement. If $g(n) = O(f(n))$, then,

$$n^{10} \leq c * n^2$$
$$\Rightarrow n^8 \leq c$$
$$n \leq c^{-8}$$

This means for a constant c that is positive enough, the inequality will be invalid since we assumed this to be true $\forall n \geq 1$.

External References for Problem 1:

1. Lecture 9: November 8, 2018, CS 5002: Discrete Structures by Adrienne Slaughter, Tamara Bonaci

Problem #2

2 (a) Prove that $2^{n+1} = O(2^n)$.

To prove this we have $f(n) = 2^{n+1}$. Let's assume $n_0 = 1$.

Let's consider the equality $2^n = 2^n$

If we multiply this by 2, we can represent it as :

$$2^{n+1} \leq 2^n * 2$$

If we put $c = 2$ then we can satisfy the condition $f(n) = O(g(n))\ if\ f(n) = c * g(n)$

2(b) Prove or disprove: $2^{2n} = O(2^n)$?

Let's assume that $2^{2n} \leq O(2^n), n_0 = 1$

Then, since $f(n) \leq c * (g(n)),\ 2^{2n} \leq c * 2^n$

Now, let's try to solve for c and cancel out terms from both sides. We can spot that 2^n is common on both sides.

Hence the equation becomes :

$2^n \leq c$

$n \leq \frac{log(c)}{log(2)}$

Hence this contradicts our assumption given a large enough c.

2(c) Let $f(n) = lg(lg^k n)$ and $g(n) = lg^k(lgn)$. Prove which one is asymptotically larger.

$f(n) = lg(lg^k n)$

$g(n) = lg^k(lgn)$

If we take $lg^k n = x^k$ and $lgn = x.

$f(x) = lg(x^k), g(x) = lg^k(x)$

Now, we know that lg grows slower than lg^k enough that $lg(x^k)$ will be significantly smaller than $lg^k(x)$.

(d) Let $f(n) = lg_3 n$ and $g(n) = lg_9 n$. Prove the relationship between f(n) and g(n) in terms of upper bound (big O), lower bound (Ω) and tight bound (Θ)

Upper bound is denoted by Big Oh O().

Since $log_b(a) = log_x(a)/log_x(b)$

$\Rightarrow log_9(n) = log_3(n)/log_3(9) = \frac{1}{2} f(n)$

$\therefore g(n) = O(f(n)) \; where \; c = 1/2$

Lower Bound is denoted by Omega (Ω).

We can use the big Oh of g(n) to look for Ω() for g(n)

$g(n) \geq c * \frac{1}{2} f(n)$

$\frac{1}{2} f(n) \geq c * \frac{1}{2} f(n)$

to satisfy this we can take an arbitrary value of c as 1/2

$\frac{1}{2} f(n) \geq \frac{1}{4} f(n)$

Hence this g(n) has lower bound to f(n)

Tight Bound (Θ)

To validate a tight bound, bound f(n) must lie between two constants c_1 and c_2

$c_1 n \leq f(n) \leq c_2 n$

Hence $g(n) = \Theta(f(n))$

Problem #3

```
def criticalsort(arr, t):
    def partition(arr, low, high):
        pivot = arr[high]
        i = low - 1
        count = 0

        for j in range(low, high):
            if arr[j] > t * pivot: # I implement the count here.
                count += 1
                arr[i + 1], arr[j] = arr[j], arr[i + 1]
                i += 1
            else:
                arr[j], arr[i + 1] = arr[i + 1], arr[j]
                i += 1

        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        return i + 1, count

    def quicksort(arr, low, high):
        count = 0
        if low < high:
            pivot_index, events = partition(arr, low, high)
            count += events
            count += quicksort(arr, low, pivot_index - 1)
            count += quicksort(arr, pivot_index + 1, high)
        return count

    return quicksort(arr, 0, len(arr) - 1)

A = [1,2,3,4,3,2,1]

threshold = 1
result = criticalsort(A, threshold)
print("Array A : ", A)
print("Number of critical events for array A : ", result, "\n \n ")

B = [3, 6, 2, 9, -12, 4]

threshold = -1
result = criticalsort(B, threshold)
print("Array B : ", B)
print("Number of critical events for array B : ", result)
```

Analysis :

1. A pivot can be chosen in O(1) time since we choose it directly from the array.

2. The partitioning can be done in O(n) time since we iterate through the entire array once.

3. Recursively calling quicksort is done in O(nlogn) in the best case if the pivot can divide the array into equal parts by picking the mean as the pivot. If the split array becomes skewed because all the elements are the say (for ex), then the time complexity can degrade to $O(n^2)$.

Problem #4

(a) Demonstrate the Bubble Sort algorithm on the input list {4, 3, 2, 1, 5}. Clearly show your steps.

We have the array : [4,3,2,1,5]

Note : numbers in parentheses like this (0) mean the array index

Iteration 1:

Comparing (0) and (1) 4>3 so array becomes : [3,4,2,1,5]

Compare (1) and (2), 4>2 so array becomes : [3,2,4,1,5]

Compare (2) and (3), 4>1 so array becomes : [3,2,1,4,5]

Compare (3) and (4), 4<5 so leave the array the same

Iteration 2:

Compare (0) and (1), 3>2 so array becomes : [2,3,1,4,5]

Compare (1) and (2), 3>1 so array becomes : [2, 1, 3, 4,5]

Compare (2) and (3), 3<4 so array stays the same

Iteration 3:

Compare (0) and (1), 2>1 so array becomes : [1,2,3,4,5]

Compare (1) and (2), 2<3 so array stays the same.

Condition exited.

Array Sorted.

(b) Let C(n) and S(Wn) be the total number of comparisons and swaps required by Bubble Sort when the input list has n numbers. For example, in the list {1, 7, 4, 5, 2} above, Bubble Sort requires 10 comparisons and 5 swaps. Suppose the input list is {n, n − 1, n − 2, . . . , 3, 2, 1}, where the numbers appear in reverse order. In this worst-case scenario, determine the exact formulas for C(n) and S(n). Clearly show all of the steps in your proof.

We assume the worst case scenario where all the elements are sorted in descending order and we have to sort the array into ascending order.

The array looks like [n, n-1, n-2,..., 3,2,1]. In this worst case array we can see that each element will be compared with the adjacent one - (n) with (n-1), (n-1) with (n-2) all the way to 1. This will lead to (n-1) comparisons. We can verify this with an example.

Suppose we have an unsorted array : [5,4,3,2,1], we have 5 > 4, 5>3, 5>2, 5>1 so n-1 comparisons.

Next we have comparisons from element (n-1) to element 1. This is because n is already sorted : [n-1, n-2, ..., 3,2,1,n]. Since one element is lesser, we are left with (n-2) comparisons. This pattern of reducing comparisons will go on till we have 1 comparison left.

Hence we can just add the total number of comparisons : (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1

This is the series progression whose sum is $\frac{n(n-1)}{2} = \frac{n^2-n}{2}$

Since a swap will happen every time there is a comparison in the worst case, C(n) = S(n).

(c) Determine a precise “loop invariant” for Bubble Sort, clearly stating your Initialization, Maintenance, and Termination statements. Prove that your loop invariant holds, clearly and carefully justifying each step in your proof.

Bubble sort came to me as a unique problem to find a loop invariant for because it has two loops, hence there must be a proof for each loop.

```
for i = n down to 1
    for j = 1 to i-1
        if a[j]>a[j+1]
            swap(a[j],a[j+1])
```

We can try to tackle each loop one by one, first proving the invariant holds true for the inner loop for iterating j at a fixed i and then the outer loop for all i.

Inner Loop

Loop Invariant : When each iteration starts for the inner loop, `a[j-1]` is the largest value in the subarray `a[1:j]` (where j is not inclusive so the array goes from 1 to j-1).

Initialization : Before the start of the first iteration, according to line 2, `j=1`. So the subarray `a[1:j]` is always sorted.

Maintenance : Now we iterate j across to i-1. According to the `if` statement, if `a[j]>a[j+1]`, we swap the numbers. If the numbers are already sorted, we go on to the next index. Hence `a[j-1]` is always sorted before the start of the next iteration. Hence the loop invariant holds true here.

Termination : Now when j = i, the loop will terminate. At this point, all the values from 1 to j-1 have been iterated through, hence the last number `a[j-1]` is the largest still. The loop invariant holds true.

Outer Loop

Loop Invariant : When each iteration starts for the outer loop, the subarray `a[i:n+1]` is sorted.

Initialization: When at the start of the iteration, i = n, the loop is sorted because it only has 1 element.

Maintenance : When i is decremented, `a[i]` is always less than `a[i+1]` since `a[i+1]` was already sorted. This is because the inner loop's invariant holds true as proved above.

Termination: After `i = 1` is reached, all the elements in the array are compared multiple times through the inner loop running again and again. This mean that the entire array is sorted and all the elements in subarray `a[1:n]` are sorted.

(d) Consider a random permutation of {1, 2, 3, . . . , n}. Determine an exact formula for the average expected number of swaps required by Bubble Sort. Clearly and carefully justify your answer.

Consider the worst case scenario for swapping from part (b), which is equal to s(n) = $\frac{n(n-1)}{2}$.

Now for each element i , we either swap or do not swap with i + 1 because there is an equal chance of the element being bigger than the number and smaller than the i+1.

So E(i) for the probability of where i is an element is $\frac{1}{2}$.

So the overall probability over all number of swaps is = s(n) * E(i) = $\frac{n(n-1)}{2} * \frac{1}{2} = \frac{n(n-1)}{4}$