Computer Science 2210A: Assignment 1

1. We need positive constant c and integer n_0 such that:
$$3n^3 \leq cn^4 \quad \forall n \geq n_0$$
By manipulating the above inequality:
$$3n^3 < 3n^4 \leq cn^4, \forall n \geq 1 = n_0$$
Therefore, choosing $c = 3,\ n_0 = 1$ suffices. ■

2. Assume $n^3 + n^2 \in O(n^2)$. By the definition of 'Big Oh', we obtain the following inequality for certain constants $c \in \mathbb{R}, c > 0, n_0 \in \mathbb{N}$:
$$n^3 + n^2 \leq cn^2,\ \forall n \geq n_0$$
By manipulating the above inequality:
$$n^3 + n^2 \leq n^3 + n^3 = 2n^3 \leq cn^2 \Leftrightarrow 2n \leq c, \forall n \geq n_0, n \neq 0$$
However, consider choosing $n_0 = 1$. Then the inequality implies that no matter how large n gets, this constant always remains greater than $2n$. This is absurd, because $n \in \mathbb{N}$, and $\mathbb{N}$ is not bounded above, so at some point, $2n$ must exceed c. By induction on $\mathbb{N}$, the same argument holds for all $n_0 \in \mathbb{N}$. The contradiction implies that the premise is false, and $n^3 + n^2 \notin O(n^2)$. ■

3. We need positive constants c and n_0 such that:
$$f(n) - g(n) \leq cf(n),\ \forall n \geq n_0$$
By the definition of 'Big Oh', it follows that:
$$f(n) \leq c_i g(n), \quad \forall n \geq n_i \quad (1)$$
$$g(n) \leq c_j f(n), \quad \forall n \geq n_j \quad (2)$$
By summing (1) and (2):
$$f(n) + g(n) \leq c_i g(n) + c_j f(n), \quad \forall n \geq \max(\{n_i, n_j\}) = n_0$$
Subtracting $2g(n)$ from the inequality:
$$f(n) - g(n) \leq (c_i - 2)g(n) + c_j f(n) \leq c_k g(n) + c_j f(n), \qquad c_k = max(\{1, c_i - 2\})$$
Finally, noting that $g(n) \in O(f(n))$, i.e. using (2):
$$f(n) - g(n) \leq c_k c_j f(n) + c_j f(n) = c_j(c_k + 1)f(n)$$
Which is exactly what we need to prove, with $n_0 = \max(\{n_i, n_j\}), c = c_j(c_k + 1)$. ■

4.

Algorithm: RepeatedLinearSearch(A, B, n)
Input: Arrays A and B of size of n to search for common values.
Output: True if neither array contain a common value, False otherwise.

```
i ← 0
while (i < n) do:
    j ← 0
    while (j < n and A[i] ≠ B[j]) do:
        j ← j + 1
    if (A[i] = A[j]) then: return False
    else: i ← i + 1
return True
```

To show that the algorithm above is correct, we first show that it terminates. Note that i is initialized to 0. Then, for every iteration of the outer loop, i increases by 1. Eventually, $i = n$ and the condition of the outer loop is false, so the loop terminates. Next, we show that the inner loop terminates. Note that j is initialized to 0. After each iteration, it increases by 1. Eventually, $j = n$, but when this happens, the condition of the inner loop is false, and the loop terminates.

To show the algorithm produces the correct output, we first explain what it is doing. Note that the outer loop marks each element of A for comparison with all of the elements of B. We first mark $A[0]$ for comparison. Then, the inner loop searches linearly within array B element by element to see if there is a match. The inner loop terminates for two reasons: all of B has been searched and no match exists, or a match exists. The condition in the body of the outer loop checks if the inner loop has terminated because of a match, and otherwise the outer loop goes into the next iteration. Finally, if the outer loop terminates and $A[i] \neq A[j]$, then the algorithm returns True.

Assume there is a common value, then the following must hold:

$$\exists p < n, \exists q < n : A[p] = B[q]$$

When $i = p, j = q$, the inner loop is false because $A[p] = B[q]$, and it terminates. Next, the condition $A[i] = B[j]$ is true, so the algorithm returns False.

Assume there is no common value. Then, in every iteration, $A[i] \neq B[j]$, but if that is the case, then for every iteration of the outer loop, the inner loop reiterates $j = 0, 1, 2, 3, \dots, n$. Every time $j = n$, the outer loop re-iterates, and $i = 0, 1, 2, 3, \dots, n$. That is, at the last iteration, $j = n, i = n$. Since all loops terminate, the last line executes, and the algorithm returns True – i.e. the arrays are indeed unique. ■

The algorithm is $O(n^2)$ at worst-case scenario. To show this, consider the case when no common element exists. Assume each iteration of the inner loop costs a_2 amount of time. Then, in total, the inner loop costs $a_2 n$. Consider the outer loop. Aside from the inner loop, its body costs a constant time amount a_1. Then, for each iteration of the outer loop, the cost is $a_2 n + a_1$. Since the outer loop reiterates n times, then total cost of the outer loop is $n(a_2 n + a_1) = a_2 n^2 + n a_1$. Finally, assume the rest of the algorithm body (aside from the inner and outer loop) cost a_0 amount of time, then in total, the algorithm costs:

$$t(n) = a_2 n^2 + a_1 n + a_0$$

By the definition of 'Big Oh', $t(n) \in O(n^2)$. ■

5. An experiment was run on a machine with following properties:

```
Operating System: Windows 10 Pro.
Processor: Intel Core i5-7300U CPU @ 2.60GHz 2.71 GHz.
RAM: 8.00 GB.
System Type: 64-bit Operating System, 64-based processor.
```

The experiment entails running tests on different solutions to the searching problem with varying input sizes. All time measurements here indicate a worst-case scenario. The following table summarizes the findings of the experiment:

Linear Search		Quadratic Search		Factorial Search	
Input Size (n)	Time (T in ms)	Input Size (n)	Time (T in ms)	Input Size (n)	Time (T in ms)
5	441	5	544	7	2813500
10	101	10	620	8	12244100
100	616	100	5884	9	48581700
1000	2349	1000	260755	10	936542000
2000	804	2000	1150361	11	8165503900
10000	4356			12	102631347600

Interestingly, for the Linear Search, the times seem to be oscillating albeit the centre of oscillation is not constant, i.e. increasing similarly to the function $f(x) = x + \sin x$. The oscillation can be attributed to the fact that since Linear Search takes very short time to execute, the timer's precision is not high enough to capture the measurements to the nearest millisecond. What matters is that the centre of oscillation is increasing as observed. Note that Quadratic Search takes significantly more times compared to Linear Search for $n > 1000$. Most importantly, Factorial Search takes the longest time and by far exceeds any other type of solution by the millions for input sizes as small as 7.