Andrew Ha – 40088148
Nora

Question 1:

1A) Note: This is under the assumption **that I will have to return two values back**. I wouldn't have used an array otherwise and used two variables instead.

**Algorithm** sumSmallerSumHigher(A, x)

**Input:** array A of integers
integer value x to compute higher/lower value sums

**Output:** array **res** with results of higher/lower value sums with index [0] containing the lower sum and index [1] containing the higher sum

**//Initialize Variables**

**//Contains the sum of value lower then X**

res[0] ← 0

**//Contains the sum of value higher then X**
res[1] ← 0

**//Looping in the and calculating the values**

**for** i ← 0 **to** A.length() - 1 **do**

**//If the value is greater than X then add it to res[1]**

**if** A[i] > x **then**
res[1] ← res[1] + x

**//If the value is not greater than X then add it to res[0]**

**else if** A[i] < x **then**

res[0] ← res[0] + x

**return** res

1B) The time complexity of my algorithm in terms of Big-O is O(n). There is a single loop that goes to the length of the array.

1C) The space complexity of my algorithm is also O(n).

Question 2:

2A) 5000000(n^5) log n + n^7 is O((n^7) log n)

This is **false** because:

f(n) = 5000000(n^5) log n + n7 is O((n^7) log n)

5000000(n^5) log n ≤ ((n^7) log n) for **n** ≥ 0
n^7 ≤ ((n^7) log n) for **n** ≥ 0 **This is false**

**For any n** ≥ 0 and c > 0

Real complexity:

f(n) = 5000000(n^5) log n + n7 is O((n^7) log n)

5000000(n^5) log n ≤ n^7 for **n** ≥ 0
n^7 ≤ n^7 for **n** ≥ 0

So, **for any n** ≥ 0

Consider n = 1 and c = 5000000 for 5000000(n^5) log n + n^7
g(n) = n^7
Consequently the above f(n) is O(n^7)

2B) (10^22)(n^21) + 5(n^5) + 120(n^3) is Θ(n^29)

**Taking the limit:** $\lim_{n\to\infty} f(x)/g(x)$

f(x) = (10^22)(n^21) + 5(n^5) + 120(n^3)
g(x) = (n^29)

Now we have: $\lim_{n\to\infty}(10^{22})(n^{21}) + 5(n^5) + 120(n^\wedge 3)/(n^{29})$

= $\lim_{n\to\infty}(10^{22})(n^{21})/(n^{29})$ + $\lim_{n\to\infty} 5(n^5)/(n^{29})$ + $\lim_{n\to\infty} 120(n^\wedge 3)/(n^{29})$

= 0 + 0 + 0

The limit = 0. This means that (10^22)(n^21) + 5(n^5) + 120(n^3) is not Θ(n^29). Rather it should be that it is O(n^29). Therefore, this is false.

2C) n^n is Ω (n!)

**Taking the limit:** $\lim_{n\to\infty} f(x)/g(x)$

f(x) = n^n
g(x) = (n!)

$\lim_{n\to\infty} (n^n)/(n!) = \infty$

Since the limit is Infinity, it is sufficient to write that f(n) = Ω (g(n)).
This means that n^n is Ω (n!) is **True.**

2D) 0.01n^3 + 0.0000001n^7 is Θ(n^3)
**Taking the limit:** $\lim_{n\to\infty} f(x)/g(x)$

f(x) = 0.01n^3 + 0.0000001n^7
g(x) = (n^3)

= $\lim_{n\to\infty} 0.01n^3 + 0.0000001n^7/(n^3)$

= $\lim_{n\to\infty} 0.01n^3/(n^3)$ + $\lim_{n\to\infty} 5(0.0000001\text{n}^7)/(n^3)$

= 0.01 + ∞

$= \infty$

Since the limit is Infinity it is only sufficient to say that f(n) = Ω (g(n)).

Therefore this statement **is false**.

2E) n^6 + 0.0000001(n^5) is Ω(n^5)

**Taking the limit:** $\lim_{n\to\infty} f(x)/g(x)$

f(x) n^6 + 0.0000001(n^5)
g(x) = (n^5)

= $\lim_{n\to\infty} \text{n}^6 + 0.0000001(\text{n}^5) /(n^5)$

= $\lim_{n\to\infty} \text{n}^6/(n^5)$ + $\lim_{n\to\infty} (0.0000001\text{n}^5)/(n^5)$

= $\infty + 0.0000001$

$= \infty$

Since the limit is Infinity it is only sufficient to say that f(n) = Ω (g(n)).

Therefore this statement is **true**.

2F) n! is Θ(2^n)

**Taking the limit:** $\lim_{n\to\infty} f(x)/g(x)$

f(x) =  n!
g(x) = (2^n)

= $\lim_{n\to\infty} n!/(2^n)$

= ∞

Since the limit is Infinity, it is only sufficient to say that f(n) = Ω (g(n)).

Therefore this is **false.**

Question 3:

3A) **Algorithm** isThreeOccurence(A, x)
**Input:** array A of integers
integer value x to compare to see if there are exactly 3 occurrences of

**Output:** Boolean isThree, that returns whether or not there is three occurrences of the specified number in array A.

**//Initialize variables**

isThree ← false

amountOfOccurence ← 0

**for** i ← 0 **to** A.length() - 1 **do**

**//If the value is equal to the value entered then increment amountOfOccurence**

**if** A[i] = x **then**
amountOfOccurence ← amountOfOccurence + 1

**end for //end of For Loop**

**if** amountOfOccurrence = 3 **then**

isThree ← true

**return** isThree

3B) The time complexity of my algorithm would be O(n).

3C) The space complexity of my algorithm would be O(n).

3D) If the algorithm was a sorted array the algorithm will change.

**Algorithm** isThreeOccurence(A, x)
**Input:** array A of integers
integer value x to compare to see if there are exactly 3 occurrences of

**Output:** Boolean isThree, that returns whether or not there is three occurrences of the specified number in array A.

**//Initialize variables**

isThree ← false

startIndex ← findFirstOccurence(A, x, 0, A.length() - 1)

lastIndex ← findLastOccurence(A, x, 0, A.length() - 1)

amountOfOccurence ← lastIndex – startIndex + 1

**if** amountOfOccurrence = 3 **then**

isThree ← true

**return** isThree

**Algorithm** findFirstOccurence(A, x, start, end)

**Input:** array A of integers
integer value x to compare
integer start which is the starting index of the array to search
integer start which is the ending index of the array to search

**Output:** return int which contains index of first 'x' in the sorted array

**If** (end >= start) **then**

mid ← (start + end)/2
**if** ((mid = 0 or (A[mid] < x)) and A[mid] = x) **then**
**return** mid

**else if (**A[mid] < x**) then**

**return** findFirstOccurence(A, x, mid + 1, end)
**else**

**return** findFirstOccurence(A, x, start, mid - 1)

**else**

**return** -1

**Algorithm** findLastOccurence(A, x, start, end)

**Input:** array A of integers
integer value x to compare
integer start which is the starting index of the array to search
integer start which is the ending index of the array to search

**Output:** return int which contains index of last 'x' in the sorted array

```
If (end >= start) then

    mid ← (start + end)/2
    if ((mid = A.length() - 1 or (A[mid + 1] > x)) and A[mid] = x) then
        return mid

    else if (A[mid] > x) then

        return findLastOccurence(A, x, start, mid - 1)
    else

        return findLastOccurence(A, x, mid + 1, end)

else

    return -1
```

The time complexity of this new algorithm would be O(logn). This is because we are doing a binary search and cutting down the complexity by half every time we search for the index.