Homework 1

The following definition of Big-O will be used for problem 1:

f(n) = O(g(n)) means ∃c, ∃n0, ∀n > n0, 0 ≤ f(n) ≤ c\*g(n) = f(n) / g(n) ≤ c

**1a.** 7n + log(n) = O(n)

[7n + log(n)] / n ≤ C

[7n + log(n)] / n ≤ [7n + n] / n = 8n / n = 8

Note: log(n) is always strictly ≤ n for n > 1

Therefore, for all n > n0 = 1 and C ≥ 8, 7n + log(n) = O(n)

**1b.** n2 + 4n + 7 = O(n2)

[n2 + 4n + 7] / n2 ≤ C

[n2 + 4n + 7] / n2 ≤ [n2 + 4n2 + 7n2] / n2 = 12n2 / n2 = 12

Note: 4n is always strictly ≤ 4n2, 7 is always strictly ≤ 7n2 for all n > 1

Therefore, for all n > n0 = 1 and C ≥ 12, n2 + 4n + 7 = O(n2)

**1c.** n! = O(nn)

n! / nn ≤ C

n! / nn = [n\*(n-1)\*(n-2)\*(n-3)\*…\*3\*2\*1] / nn ≤ [n\*n\*n\*n\*…\*n\*n\*n] / nn = nn / nn = 1

Note: Every term after the first in a factorial sequence is strictly < n

Therefore, for all n > n0 = 1 and C ≥ 1, n! = O(nn)

**1d.** 2n = O(22n)

2n / 22n = 1 / 2n ≤ C

1 / 2n ≤ 2n / 2n = 1

Note: 1 is always strictly < 2n for all n > 1

Therefore, for all n > n0 = 1 and C ≥ 1, 2n = O(22n)

**2a.** f(n) = n2, g(n) = n3

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

Since the limit = 0, f(n) is strictly < g(n) meaning f(n) = o(g(n)). Therefore, f(n) = O(g(n))

**2b.** f(n) = log2(n), g(n) = log3(n)

f(n) = Θ(g(n))

Since the limit = 0 < 1.58496250072 < ∞, f(n) = Θ(g(n))

**2c.** f(n) = 2n, g(n) = 3n

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

Since the limit = 0, f(n) is strictly < g(n) meaning f(n) = o(g(n)). Therefore, f(n) = O(g(n))

**2d.** f(n) = 2n, g(n) = 2n+1

f(n) = Θ(g(n))

Since the limit = 0 < ½ < ∞, f(n) = Θ(g(n))

**3.**

Base Case: n = 2, T(n) = 2

Since T(n) = n\*lg(n), T(2) = 2\*lg(2) = 2 \* 1 = 2

Therefore, the base case holds true

Inductive step:

Assuming that T(n) = n\*lg(n) and T(n) = 2T(n/2) + n if n = 2k for k > 1, then for n = 2k+1:

T(2k+1) = 2T(2k+1 / 2) + 2k+1

T(2k+1) = 2T(2\*2k / 2) + 2k+1

T(2k+1) = 2T(2k) + 2k+1

Since we know that T(n) = n\*lg(n)…, then

T(2k+1) = 2\*(2k\*lg(2k)) + 2k+1

T(2k+1) = 2k+1\*lg(2k) + 2k+1

T(2k+1) = 2k+1\*(lg(2k) + 1)

T(2k+1) = 2k+1\*(k+1)

T(2k+1) = 2k+1\*(k+1)\*1

T(2k+1) = 2k+1\*(k+1)\*lg(2)

T(2k+1) = 2k+1\*(lg(2k+1)

Therefore, by the principal of mathematical induction, the recurrence has been proved.

**4a.**

O(1) if n ≤ 1

T(n-1) + O(n) if n > 1

T(n) =

**4b.**

T(n)

T(n-1) n

T(n-2) n-1

T(n-3) n-2

T(n-4) n-3

T(n-5) n-4

……

T(1)

T(0) 1

At each level of the tree, the amount of work done is the time it takes to call insertion sort on an array of size n-1, for every n, plus the time it takes to insert into the correct position, which is a linear function of n (n, n-1, n-2, n-3, n-4, …). Therefore, the total work done in this recursive version of insertion sort is 1 + 2 + 3 + … + (n-2) + (n-1) + n = [n(n+1)] / 2 = (n2 + n) / 2 = O(n2).

**5a.** The running time to merge 4 arrays is going to be a function of the sum of their respective sizes. For example, consider arr1, arr2, arr3, and arr4 of sizes n1, n2, n3, and n4. The running time to merge arr1, arr2, arr3, and arr4 into a sorted array is O(n1 + n2 + n3 + n4).

**5b.**

T(n) = 4T(n/4) + O(n)

T(n) = 4[4T(n/16) + n/4] + n = [16T(n/16) + n] + n = 16T(n/16) + 2n

Assuming that n is some power of 4, meaning n = 4k, then…

T(n) = 4kT(n / 4k) + kn

T(n) = nT(n / n) + n\*log4(n)

T(n) = nT(1) + n\*log4(n)

T(n) = n\*1 + n\*log4(n)

T(n) = n + n\*log4(n)

T(n) = O(n\*log4(n))

T(n) = O(n\*log(n))

While technically, n\*log4(n) is an improvement on n\*log2(n), in terms of Big-O complexity, modifying merge sort to split into 4 non-overlapping arrays does not improve on the traditional algorithm. Since the difference between any two logarithm bases is a constant, log4(n) is the same as log2(n) in terms of Big-O complexity since constants are not considered in asymptotic efficiency. While the sorting phase of merge sort will be faster by splitting the array into 4 sub-arrays, the merging phase will take longer, so there is really no improvement.