Problem 1)

1. =

Problem 2)

1. True.
   1. If f(n) = O(g(n)), then g(n) ≥ c \* f(n) for a large enough number.
   2. If g(n) = Ω(f(n)), then c \* g(n) ≥ f(n) for a large enough number.
   3. Given that the constants don’t matter, O(c \* g(n)) = O(g(n)) and Ω(c \* f(n)) = Ω(f(n))
   4. So since both statements rely on the same inequality, then if one is true, then both are true
2. True.
   1. If f(n) = O(g(n)), then the big-O complexity of f(n) can never be greater than the big-O complexity of g(n), given the definition.
   2. The big-O complexity of a polynomial is equal to the highest degree function in that polynomial (additivity rule)
   3. Therefore, given 100f(n) + g(n), the highest complexity function in this polynomial will always be g(n).
   4. It is a given that g(n) = O(g(n)) and that the big-O complexity of a polynomial is always based on the highest complexity function in that polynomial, so 100f(n) + g(n) = O(g(n)).
3. False.
   1. For example, f(n) = (x2 + x) and g(n) = x2.
   2. x2 + x = Θ(x2) is true because x2 + x = O(x2) and x2 + x = Ω(x2).
   3. However, 2x^2 + x = 2x^22x =/= O(2x^2) and so =/= Θ(2x^2).
4. True.
   1. If f(n) = O(g(n)), then g(n) > c \* f(n) towards infinity
   2. If g(n) > c \* f(n), then g(n)2 > c2 \* f(n)2 as well, since c = c2 here
   3. So f(n)2 = O(g(n)2).
5. True.
   1. f(n) = Θ(g(n)) means that f(n) = O(g(n)) and f(n) = Ω(g(n))
   2. That means f(n) and g(n) can only be off by a constant c
   3. i.e. f(n) = c \* g(n) in terms of big-Theta
   4. 1/f(n) = 1/(c \* g(n)) therefore
   5. 1/c = c when it comes to constants here
   6. So 1/f(n) = Θ(1/g(n))

Problem 3)

The baseis for being able to perform a kind of binary search on the given array in this problem, in order to make an algorithm faster linear time, are the assumptions we make about each half of the input array based on the center value. In a standard binary search algorithm, given an array sorted in ascending order, you can assume that each value in the bottom half of the array will be less than or equal to the given middle value, and the top half will be greater than or equal to the given middle value. We can make the same assumption in this case.

However, this algorithm requires the extra assumption that each integer value is distinct, which the standard binary search algorithm doesn’t require. Take this array, for example:

1 3 4 5 6 7 8 9 10

The midpoint of this array is the value 6 at index 5, meaning A[k] > k, and we can see that, even as the values on the right side of the midpoint increase by 1, the minimum an integer can increase by, it is impossible for A[k] to ever be equal to k if A[k] > k is true at any element before it. However, on the right side, A[1] = 1, so we only need to worry about the right side. However, if we take an array of integers that isn’t distinct:

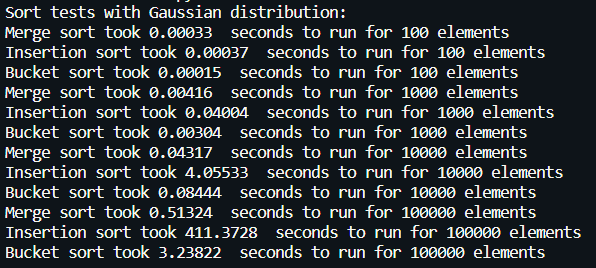
-1 1 4 5 6 7 7 9 10

We can see that the assumption made about the upper half is no longer necessarily true. Our binary algorithm would discard the top half and eventually conclude that there is no case where A[k] = k in this array, when in fact A[7] = 7. That is because having duplicate integer values effectively allows the array to “skip” an index. We can no longer be sure that the value of an element will be at least 1 greater than the element that came before, which is the key assumption an array of distinct integers allows us to make. For one final example, a perfect array starting at 1 and ascending by 1s:

1 2 3 4 5 6 7 8 9

This works because A[k] = k at the midpoint. Whether the value at the midpoint is greater than, less than, or equal to the midpoint index, the algorithm works, but only if the array is sorted and every integer in it is greater than its predecessor by at least 1 (distinct).

Problem 4)



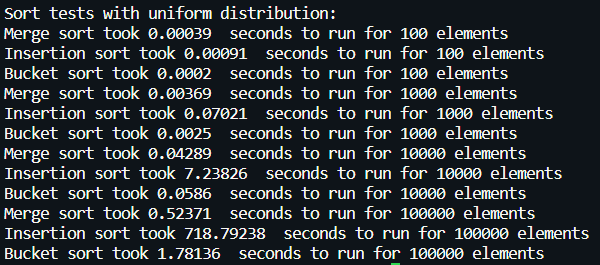
In all four cases, insertion sort is the slowest, which is to be expected. Insertion sort is, on average, O(n2), while merge sort is O(nlog(n)) very consistently. Merge sort increasingly outperforms insertion sort as the number of elements get larger and larger. Bucket sort manages to perform about as well as merge sort for the first two cases, but it begins performing much worse as the number of elements in the list increases. In pure time complexity, bucket sort has an average running time of O(n), but clearly merge sort’s O(nlog(n)) beats it in this case. Clearly this has something to do with bucket sort’s worst case of O(n2).

The worst list for bucket sort would be a list with a skewed distribution, where the vast majority of the elements fit into only a few buckets. My particular implementation of bucket sort uses insertion sort to sort each bucket before combining them together. If only a few buckets hold all the elements, bucket sort relies on insertion sort to pick up the slack, which will obviously slow the algorithm down severely. Bucket sort performs far better on a uniformly-distributed dataset, which is where its best case of O(n) comes from.

That being said, a Gaussian distribution is not uniformly distributed. It is unimodal, with most values appearing at the center of the distribution, forming a bell curve. For bucket sort, this means that most of the elements end up in the center buckets with increasingly fewer elements in each bucket as you travel further from the center. This can be demonstrated by this output:



Each integer represents the number of elements in each bucket, from the smallest to greatest interval, for a random list of 1000 Gaussian-distributed values with the specified mean and standard deviation. My particular algorithm creates sqrt(n) buckets (rounded down), so it created 31 buckets for a list of size 1000, yet only 10-12 of them have a significant number of elements in them. This is how the sorts run with a uniform distribution, rather than a Gaussian one:



Bucket sort certainly performs faster, but with 10,000 and 100,000 elements, merge sort still beats it. Interestingly, insertion sort becomes slower with a uniform distribution, likely because the lists are less ordered to begin with. Insertion sort works best with a nearly-ordered list (in a list that is only one pass-through away from being sorted, insertion sort works in O(n) time), and a uniform distribution is likely further away from being ordered than a Gaussian one would be my guess.