Assignment #1 (Ternary Search)

PART 1

B) ternarySearch([1,3,5,10,12,15,32,91,125,132], 18)

It will calculate n/3 = (3) and 2n/3 = (6) to determine segmentation then the function will check if K is less than a[3] = 10 then between 10 and a[6] = 32. K = 18 is between 10 and 32 so it will call ternarySearch from 10-15 like this ternarySearch([10,12,15], 18). It will go through this process again and discover that 18 is greater than 12 so it will call ternary search with the last element ternarySearch([15], 18). This will reach the base case of len(a) = 1 and will check if 15 = k. Since it does not, the function will return false.

C) Proof: Ternary Search will return True if and only if k is in the list A. If the list A is empy. It will return False.

Proof by induction: The base cases will produce correct results. A[] = empty will return False, and a list of 1 element will match on K and return either true or false. A list of 2 elements will also match on K and return true or false.

A list of 3 elements will recursively call the algorithm on a list of one element within the bounds of K.

Now using strong induction assume that this function works for all values up to a list of size M. If k < A[n/3] and K is in A. Then K has to be in sublist A’ composed of elements 0, A[n/3]. By induction, we can assume that trinarySearch(A’, k) will produce the correct result since A’ is a list smaller than size M. This can be used to prove the other two subcases of A[n/3] <= k <A[2n/3] and k >= A[2n/3] since both of those cases will produce lists smaller than size M. QED

D) every time the function is called it reduces the size of the list by a factor of 3. An array sized 3 finishes in 2 steps (one to reduce and one to match a base case). An array of size 9 takes 3 steps. An array of size 15 also takes 3 steps, and an array of size 27 takes 4 steps. It would appear that the BigO for this function is Log3(n) + 1.

PART2

A)

Insertion sort works by comparing each element to the previous element and cycling through until the current element is in its proper place. With a list that is fully sorted this will take O(n) iterations to compare each element with the element before it(Base Case). With a list that is all but K sorted, this big O will be increased for each element K that is out of place. Each element will take (worst case) O(n) operations to be sorted because it will have to be compared with each element in the list before it till it is moved from the last place in the list to the first. So for each out of place element an extra O(n) operations will be needed. This will hold true for any “all but K+1 sorted list” as well and leads to the expression O(n X k) for each k element out of place. QED

B) The merge sort presented in class doesn’t match on anything or check for correctness, it simply breaks down and reorders lists into a correct order. This means that even with a sorted array it will go through the same number of steps as an unsorted array. This could be fixed simply by adding in a check to see if the pieces of the list are in order before calling the merge sort algorithm. However, this wouldn’t add much efficiency and would rarely be of much use.