* To sort large sequence of keys that can’t fit in memory **MERGESORT** is the recommended.
* In real time scenario where a sort must be completed within a fixed amount of time and the input can fit in memory, then **HEAPSORT** is the recommended choice.
* The standard bucket sort distributes the keys into buckets, sorts each bucket, then concatenate the buckets. This algorithm runs in linear time no matter how the keys are distributed over the buckets **FALSE**
* A sorting algorithm is considered in place if it use some memory in addition to the memory used by the input sequence but only a constant amount example not an amount that increases as n increases **TRUE**
* In a self-balancing binary search tree, to remove a key element pair, the removeElement(K) method calls expandExternal(v) **FALSE**
* In a binary search tree, every internal node has either one or two children **FALSE**
* the lower bound on sorting by key comparisons is O(n) since we can always do a bucket or radix sort **FALSE**
* All implementations of an unordered dictionary are necessarily inefficient for finding items since the entire dictionary might have to be scanned to find a key **FALSE**
* In reading start, the key is divided into components and bucket sort is run on each component, starting with the most significant high order component down to the list signal significant component load order **FALSE**
* Quicksort is an example of the wine and conquer approach with worst case time complexity that is no better than selection sort or incessant sort **TRUE**
* in a heap, external nodes cannot appear on more than one level **FALSE**
* Post order traversal off a binary tree means the parent left child and right hand notes are visited in the following order **left child right child and parent**
* in altered travels of diner it remains the parent left child and right child notes our visit in the following order **left child parent right child**
* The maximum depth of any external node of tree tea is called the **height** of T
* In a circular growable area implementation of the key ADT the increase and decrease operations running **n** and **n** immortalized time if the earlier size is increased by constant C each time it has to be enlarged
* In a growable area based implementation of stack ADT the push operation run sing 01 immortalized worst case time when the area site increases by **double** each time resizing is necessary
* in a hash table implementation of the dictionary ADT insert item find element and element operation running **1 1 and #1** expected time respectively
* In a sorted growable array implement of the dictionary ADT, insert item, find element, and remove element operations run in **n**,, and **n** time respectively
* In a unsorted grabbable area implementation of a dictionary ADT insert item find element and remove element operations running **1, n** and **n** time respectively

Our midterm was almost the same as PreviousMidterm\_Ruby especially fill in the blank and true false questions.

The algorithm design questions were different, we had 4 questions:

1. You have single list of elements contains duplicates. Design in-place algorithm to remove the duplicates.
2. You have unsorted stack of numbers. Sort the stack, you can only use **ONE** extra stack.
3. Problem to use ordered dictionary to get the max count. (same as book/publisher problem with ruby).
4. You have {a,a,a,a,b,b,c,d,d,d,a,a} we need to have a compression algorithm. So the output should be {a,4,b,2,c1,d,3,a,2} First you need find the size of the new array, if the size is bigger than the size of the original array -> compression inefficient and do not do it, otherwise do the compression in that way.