CPSC 331 Lecture 1, Tutorial T04
Written by Bruce Laird 10158951
Assignment 2
Question 5

	Insert Sort	Heap Sort	Quicksort	Improved Quicksort
Decreasing Input 128	0	1	0	1
Decreasing Input 1024	4	1	2	0
Decreasing Input 16384	81	6	33	3
Decreasing Input 65536	1183	12	576	10
Increasing Input 128	0	1	0	0
Increasing Input 1024	0	2	3	0
Increasing Input 16384	0	12	40	2
Increasing Input 65536	1	17	384	4
Random Input 128	0	1	0	0
Random Input 1024	3	2	0	0
Random Input 16384	42	7	3	3
Random Input 65536	590	14	9	11

*All values above are in milliseconds

Insertion Sort

When looking at decreasing input, insertion sort is slowest of the 4 algorithms. This is because insertion sort's worst case running time occurs when the input is sorted in decreasing order since it has to swap every value. When comparing the other algorithms time complexity in decreasing input it should be noted that insertion sort has a quadratic running time in worse case while the other three have a polynomial run time.

When looking at the table it is very clear that the increasing order input is the fastest run time for insertion sort even in the largest input of 65536. These results make sense since when the input is already in increasing order insertion sort has a linear running time. Insertion sort is one of the fastest algorithms for sorting small arrays but in the random input and decreasing input where the input size is 65536 it is clear that insert sort has trouble handling large arrays when compared to quicksort and heap sort.

HeapSort

Also mentioned in class the fact that the worst and best case for HeapSort is O(n log (n)). When you examine the HeapSort column it is clear that this algorithm is fairly consistant regardless of input given. This consistency is because HeapSort's time complexity in both worst and best case is O(n log (n)). The time complexity remains O(n log (n)) regardless of input, this is clear when you look at Decreasing Input, Increasing Input and Random input, the times are all very similar because each has O(n log(n)) runtime. I found it interesting that HeapSort consistently runs faster than QuickSort. At first I did not understand this since both have a polynomial running time but then when you take into account the fact that QuickSort uses recursion while HeapSort does not it makes more sense that QuickSort would be slower.

CPSC 331 Lecture 1, Tutorial T04
Written by Bruce Laird 10158951
Assignment 2

QuickSort

In QuickSort the pivot value is critical to how fast the program will run. If the pivot is one of the smallest values or one of the largest values then the array will be partitioned into 2 arrays in which one will be very large and the other will be very small - which causes worst case partitioning and slows down the runtime. This is exactly what happens when the arrays are in increasing and decreasing order, which is why the runtime is so large. It was also interesting to see that when looking at smaller inputs the recursion in quicksort slows it down causing it to be slower than insertion sort.

The best runtimes for QuickSort were found using Random Input. This makes sense because the pivot value is a large factor in determining how efficient the algorithm will be. I believe the reason random input was so efficient in the larger inputs (16384 and 65536) is the value picked for the pivot (in this case length n -1) is found close to the middle of the array when it is sorted. Therefore it causes partition to create 2 similar sized arrays to sort and makes the runtime much shorter when compared to the increasing and decreasing input. I actually checked this theory on one of the inputs; I located the median and determined the pivot was very close to it.

Improved QuickSort

This algorithm was clearly the best when looking at the table. Insertion sort worked best with smaller arrays and QuickSort was best for larger sub-arrays. Improved QuickSort uses Insertion sort when the array size was partitioned to a small enough array and when it was too large it would use quicksort. The problem the original QuickSort encountered was in choosing a pivot value which was not close to the middle of the sorted array, Improved Quicksort solves this problem by taking the first, last and middle value in the unsorted array and determines the median value and then uses that median value as the pivot.