Brian Loughran
Programming Assignment #2 Analysis

README:

Python version: 3.6.2

Input File: inputVals.txt

Input File Format: Each of the sorting algorithms included in this package run off the same input file. The input file contains the values to be sorted. Each value is on its own line. Any line that cannot be trivially converted to an integer will be ignored. One example of a correct input for 5 values to be sorted would be the following:

5
2
4
1
3

Input File Alternate Format: There is an alternate format for the input file which creates a random series of values which is quite useful for testing purposes. The format for this file requires that the first line be the string “Random Values” (not case sensitive). The second line of the file indicates the number of values as an integer. The third line of the file represents the maximum number value of the values. One example of a correct input for 5 values with a maximum value of 10 would be the following:

Random Values
5
10

Input File Alternate Format (2): There is a second alternate file format for the input file which creates a series of values which will be in reverse-sorted order, again useful for testing purposes like getting the worst-case runtime for quicksort. The format of this file requires that the first line be the string “Backwards Values” (not case sensitive). The second line of the file indicates the number of vales to be created as an integer. One example of a correct input for 5 backwards values would be the following:

Backwards Values
5

A file is included in the submitted package with the designated formatting for convenience. As stated above, the input file is the same for all of the sorting algorithms included in the package.

Execution: There are seven .py files in the directory. Each .py file corresponds to a different sorting algorithm (except for the CommonUtils.py file, which has some common functions that each of the sorting algorithms use, and the Counter.py file, which is a counter) and each sorting .py file has its own __main__() function, thus can be run to perform the corresponding sorting algorithm on the given input file. A table is included below to specify the sorting algorithm implemented by each .py file.

Brian Loughran
Programming Assignment #2 Analysis

.py file	Sorting Algorithm
HeapSort.py	Standard Heap Sort
QuickSort.py	Quick Sort (Uses last element in subarray as partition)
IntroSort.py	Intro Sort (Uses QuickSort.py and HeapSort.py in implementation)
QuickSortMOT.py	Quick Sort (Uses median of three for partition)
IntroSortMOT.py	Intro Sort (Uses QuickSortMOT.py and HeapSort.py in implementation)
CommonUtils.py	Does not sort, just for some common functions
Counter.py	Does not sort, just a counter

Each program (CommonUtils.py and Counter.py notwithstanding) has its own __main__() function, so you can run each separately from the command line or in your favorite python IDE. The code will read the values in the input file and write the result to an output file corresponding to the file ran. If the output is less than 50 lines it will output to the console as well.

Output: After running one of the given sorting algorithms, results will appear in the output file designated by the table below (depending on which .py file was ran). The output here will first echo the points to the output. Next there will be some debugging level messages to show each step in the algorithm (ex. for quick sort it will show the partition value and result of the partition, for heap sort it will show each value popped from the top of the heap and the heapify step). Finally it will output the sorted values. Example output files are included in the submitted package for convenience. See the table below to see which output file corresponds to which .py file:

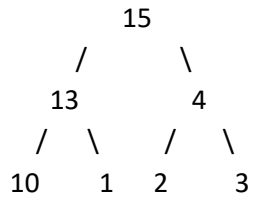
.py file	Output file
HeapSort.py	HeapSortOutput.txt
QuickSort.py	QuickSortOutput.txt
IntroSort.py	IntroSortOutput.txt
QuickSortMOT.py	QuickSortMOTOutput.txt
IntroSortMOT.py	IntroSortMOTOutput.txt
CommonUtils.py	-
Counter.py	-

ANALYSIS: Heap Sort

Heaps (Background):

To understand heap sort we must first understand how heaps work (in this implementation, we use a max heap, so we will discuss how max heaps work). Max heaps are a type of tree structure where each parent node can have up to two children. Each of the children in a max heap must have value less than the value of their parents and the tree must be complete. One valid heap may look something like this (note this is a valid max heap since each of the children have a value less than their parent):

Brian Loughran
Programming Assignment #2 Analysis



While heaps can be represented as trees like in the above diagram, in some instances it is advantageous to represent a max heap as an array. Denoting the parent index as i, this can be done by having the root node at index 0, and the right child of any parent node at index 2i + 1 and the left child of any parent node at index 2i + 2. The above max heap tree structure then can be represented as the following array:

[15, 13, 4, 10, 1, 2, 3]

Now that we know the properties of a max heap, we can begin to define some operations that can be done. To get the maximum value of a max heap, we can simply reference index 0 of the max heap. We know this is the maximum value of the heap because it is a value greater than each of its children, and those children are the maximum values of their children, etc. Thus we can get the maximum value of the heap in O(1) time at index 0 of the max heap.

Another procedure we can do to a heap is called heapify(). heapify() is a recursive procedure which checks the left and right child of any node in the max heap and swaps the value of the child and parent if one of the children has a value greater than the parent. Once that step is complete, it will heapify() the effected subtree. This method builds the max heap in a top-down manner and will complete in O(lgn) time, where n is the number of nodes in the subtree. This is a useful procedure if you know/suspect one of the parent nodes is not conforming to the properties of the max heap and want to place it at the right index among its children. We will see where this can be very useful in both creating the heap and later in the HeapSort() algorithm.

To create a max heap from an array of random data, we can simply call heapify() on each index of the array in descending order (start at index n, move up to index 0). Since heapify has been shown to take O(lgn) time and we call it O(n) times, creating the max heap will take no more than O(nlgn) time.

Algorithm:

Now that we know how max heaps work, we can define a sorting algorithm using max heaps called HeapSort(). A valid sorting algorithm will look something like this:

1. Build max heap (O(nlgn))
2. Swap first and last values in the heap, this will result in the maximum value in the heap being at the last index O(1)

3. Reduce the size of the heap by 1, now the previous highest value in the heap is outside of the heap and in its proper position in the result array. O(1)
4. Run heapify() on the root node of the heap, this will result in a valid heap. O(lgn)
5. Repeat until the size of the heap is 1

This algorithm will result in a sorted array. This is a valid algorithm because for each iteration of steps 2-4, the greatest remaining value in the heap will be sent to the right most index of the heap. Doing this repeatedly will result in the array being sorted in descending order from left to right (a.k.a. ascending order).

Psuedo-Code (Heap Sort): Heap Sort Implementation

```
// run heapify() as described in Heaps (background)
heapify(heap, size, index):
  largest_index = index            // initialize the largest value to node value
  left_index = index*2+1           // get the index of the left child
  right_index = index*2+2          // get the index of the right child

  if (left_index < size && heap[left_index] > heap[index]):        // check if left child greater than parent
    largest_index = left_index                                     // set largest index to left child

  // check if right child greater than largest
  if (right_index < size && heap[right_index > heap[largest_index]):
    largest_index = right_index                                    // set largest index to right child

  if (largest_index != index):                                     // check if the largest is not the root
    swap(heap[index], heap[largest_index])                         // swap largest and root
    heapify(heap, size, largest_index)                             // recursively call heapify for valid heap

// sort an array of values using the heap sort algorithm
heap_sort(values):
  size = len(values)               // get the number of values to sort
  for i in range(len(array)-1, -1, -1):        // iterate over values starting at last index
    heapify(values, size, i)       // call heapify on each value in values to create a max heap

  for i in range(len(array)-1, 0, -1):         // iterate over values starting at last index
    swap(values[0], values[i])     // swap first and last values of heap
    heapify(values, i-1, 0)        // build valid max heap on the reduced heap

  return values                    // values is a sorted array!

__main__()
```

```
values = get_values()          // get the values as an array
return heap_sort(values)       // heap sort the values and return the sorted array
```

Runtime (HeapSort): For a problem with n values, HeapSort will take O(nlgn) to sort. We have already discussed how creating a max heap takes O(nlgn) time, which is step 1 of our algorithm. Steps 2-4 will take O(lgn) time, and steps 2-4 are repeated O(n) times since they are done to each value in the array. Thus the total runtime is O(nlgn) in the worst case. This implementation will do O(nlgn) comparisons.

ANALYSIS: Quick Sort

Algorithm:

Quick sort is a divide and conquer algorithm, somewhat similar to merge sort. Quick sort works by selecting an element in a subarray of values to be the partition element. Values less than the partition element are sent to the left of the partition element, and values greater than the partition element are sent to the right of the partition element. Thus the partition element is in its proper place in the resultant array, and you can partition the left and right subarrays. Doing this recursively until the size of each subarray is 1 or less will result in a sorted list. These steps are enumerated below:

1. Partition the array of values
2. Partition the subarray to the left of the partition
3. Partition the subarray to the right of the partition
4. Repeat until each subarray has a size of 1 or less

Note that we have not specified the value of the partition. This algorithm will actually work using any value as the partition, however for simplicity sake in this implementation we will select the last value of the subarray to be the partition.

This algorithm must result in a sorted array since each partition step will place the value of the partition in its proper place in the given subarray. Since this partition step is repeated until the size of each subarray is 1 or less, each value will be placed in its proper place in a sorted array. Since each value is in the correct place, the whole array must be sorted.

Algorithm using Median of Three:

As mentioned in the previous section, it does not matter which value we select as the partition. However, in the previous implementation we chose the last value of the subarray to be the partition. In this implementation, for a given subarray to sort, we will select the median of the first value, the last value, and the middle value. This is commonly referred to the median of three partition. Other than selecting the median of three element for the partition, the rest of the algorithm is the same. Thus the steps and proof of correctness are the same as in the Quick Sort Analysis section, and will not be repeated here.

Brian Loughran
Programming Assignment #2 Analysis

Psuedo-Code (Quick Sort): Quick Sort Implementation

```
// create the partition as described in the previous section
partition(array, low_index, high_index):
  swap_index = low_index                        // initialize the value of the swap index to the low index
  pivot = array[high_index]                     // get the value of the pivot as rightmost value

  for index in range(low_index, high_index):    // iterate over each index in subarray
    if (array[index] < pivot):                  // check if value is less than pivot
      swap(array[index], array[swap_index])     // place value to left of eventual pivot location
      swap_index += 1                           // increment swap index

  swap(array[high_index], array[swap_index]     // place pivot
  return swap_index                             // return the index of the pivot

// run quick sort using the partition() function above recursively
quick_sort(values, low_index, high_index):
  if low_index < high_index:                    // ensure that we are working with valid subarray
    // perform partition on subarray and get the index of the partition
    partition_index = partition(values, low_index, high_index)

    quick_sort(values, low_index, partition_index-1)      // quicksort left subarray
    quick_sort(values, partition_index+1, high_index)     // quicksort right subarray

__main__()
  values = get_values()                         // get the values as an array
  quick_sort(values, 0, len(values)-1)          // run quicksort on values
  return values                                 // return the sorted array
```

Psuedo-Code (Quick Sort using Median of Three): Quick Sort using Median of Three Implementation

The quick_sort() and __main__() functions will be the same as discussed in the previous section. The pseudocode below displays how to use the median of three in the partition() function

```
// create the partition as described in the previous section
partition(array, low_index, high_index):
  swap_index = low_index                        // initialize the value of the swap index to the low index
  // get the value of the MOT pivot and its index
  pivot, pivot_index = get_pivot_median(array, low_index, high_index

  swap(array, high_index, pivot_index)          // place the median value at the end of the array

  for index in range(low_index, high_index):    // iterate over each index in subarray
```

```
   if (array[index] < pivot):                  // check if value is less than pivot
     swap(array[index], array[swap_index])     // place value to left of eventual pivot location
     swap_index += 1                           // increment swap index

  swap(array[high_index], array[swap_index]    // place pivot
  return swap_index                            // return the index of the pivot

get_pivot_median(array, low_index, high_index):
  if high_index - low_index < 2:               // check that there are at least 3 values for median
    return array[high_index], high_index       // return the value and the index of the last element
  else:
    mid_index = (low_index + high_index) // 2  // get the index in the middle
    low_val = array[low_index]                 // set the value of the low item
    mid_val = array[mid_index]                 // set the value of the middle item
    high_val = array[high_index]               // set the value of the high item
     if low_val > mid_val:
       if low_val < high_val:
        median = low_val                       // the low value is the median
       elif mid_val > high_val:
        median = mid_val                       // the middle value is the median
       else:
        median = high_val                      // the high value is the median
    else:
     if low_val > high_val:
      median = low_val                         // the low value is the median
     elif mid_val < high_val:
       median = mid_val                        // the middle value is the median
     else:
      median = high_val                        // the high value is the median
     if median == low_val:
      median_index = low_index                 // the low index is the median index
     elif median == mid_val:
      median_index = mid_index                 // the middle index is the median index
  else:
    median_index = high_index                  // the high index is the median index
   return median, median_index                 // return the median and its index
```

Runtime (QuickSort): For Quicksort, it is helpful to understand the runtime of the partition() function and use that to determine the total runtime. The partition() function checks the value of each element in the subarray and places it either to the left or right of the partition. Since it does this once for each element in the subarray, the partition function takes O(n) time where n is the length of the subarray

Brian Loughran
Programming Assignment #2 Analysis

QuickSort has different runtimes for worst-case and average-case. First we will discuss worst case runtime. The worst case scenario for QuickSort is receiving an array sorted in descending order. In this case, QuickSort will select the highest value in the subarray and partition() around that. This will result in the left subarray containing all the values except the highest value, and the right subarray containing no values. Removing one value from the subarray at a time will cause QuickSort to run partition() O(n) times, and since partition takes O(n) time, QuickSort has a worst case runtime of O(n^2).

However, if we assume that the values given to QuickSort are random, we can arrive at a different result. Assuming that the values are random, the value selected for partition() will fall at some random index in the subarray, however the average location will be the middle of the subarray. Since on average, partition() will cut the subarray in half, each subsequent recursion will only have to act on half of the subarray, resulting in O(lgn) recursions. O(lgn) recursions of time O(n) will take an average of O(nlgn) time (assuming that the input data is random).

Thus QuickSort has worst case O(n^2) runtime, but averages O(nlgn) runtime.

Runtime (QuickSort using Median of Three): The algorithm for QuickSort using Median of Three is widely similar to the previous implementation of QuickSort. The first difference of note is the get_pivot_median() function. This function takes O(1) time since it is performing a finite set of computations on just O(3) items. Thus this has no effect on the runtime of QuickSort.

Another difference is an added swap() call within the partition() call. This is done to send the pivot to the end of the array, which allows us to reuse as much of the previous partition() function as possible. Since we have already discussed the runtime of partition() as O(n) worst case, adding an extra O(1) call with swap() will not have any effect on runtime.

Previously discussed was the worst case runtime of QuickSort when sorting an array in descending order. Using the median of three method will reduce the number of times that we have to call partition. However, in the worst case, the median of three will select the second highest value in the subarray and partition() around that. This will cause QuickSort to call partition() O(n/2) times, unfortunately still resulting in O(n^2) runtime in the worst case. However, it will likely perform better (O(nlgn)) on values sorted in descending order, since the median of three will partition() the array into equal halves in this case.

For the same reasons discussed in the QuickSort Runtime section, for random values we still have O(nlgn) runtime. Thus the worst case runtime of O(n^2) and average runtime of O(nlgn) remain the same for Quicksort using Median of Three.

ANALYSIS: Intro Sort

Algorithm:

Brian Loughran
Programming Assignment #2 Analysis

IntroSort is a hybrid sorting algorithm, meaning that it uses the best aspects of multiple sorting algorithms to create a faster, more versatile sorting algorithm to work on a wide variety of inputs. IntroSort works by using QuickSort and HeapSort (of which the process, pseudocode and runtime are discussed in detail in previous sections). For IntroSort, the base algorithm used is QuickSort, and HeapSort is used if the recursion depth of QuickSort gets too high. So the algorithm for IntroSort is shown below (note that the algorithm remains the same for standard QuickSort or the Median of Three method of selecting the pivot):

1. Use QuickSort to partition the array into subarrays
2. Use HeapSort if QuickSort passes the recursion depth limit

One element which has not yet been discussed is how to determine the recursion depth limit. This value is mostly arbitrary (however we would not want to select a value less than 1 or more than the length of the list of values). One suggestion that I found doing research on other IntroSort implementation used a recursion depth of 2*lg(n), where n is the length of values to be sorted. This has worked pretty well in my own tinkering, so we will use that as our recursion depth limit.

Since both QuickSort and HeapSort are valid sorting algorithms as described in previous sections, IntroSort must also be a valid sorting algorithm.

Psuedo-Code (Intro Sort): Intro Sort Implementation

```
// run introsort
intro_sort(values, left_index, right_index, depth_limit):
 if depth_limit < 1:                                  // check if we have reached the recursion depth limit
  // run heap sort on subarray
  values[left_index:right_index] = heap_sort(values[left_index:right_index])
 else:
  // perform partition on subarray and get the index of the partition
  partition_index = partition(values, left_index, right_index)

  intro_sort(values, left_index, partition_index-1, depth_limit-1)        // quicksort left subarray
  intro_sort(values, partition_index+1, high_index, depth_limit-1)        // quicksort right subarray

__main__():
 values = get_values()                                                    // get the values as an array
 recursion_depth = 2 * lg(len(values))                                    // get the recursion depth
 intro_sort(values, 0, len(values)-1, recursion_depth)                    // run quicksort on values
 return values                                                            // return the sorted array
```

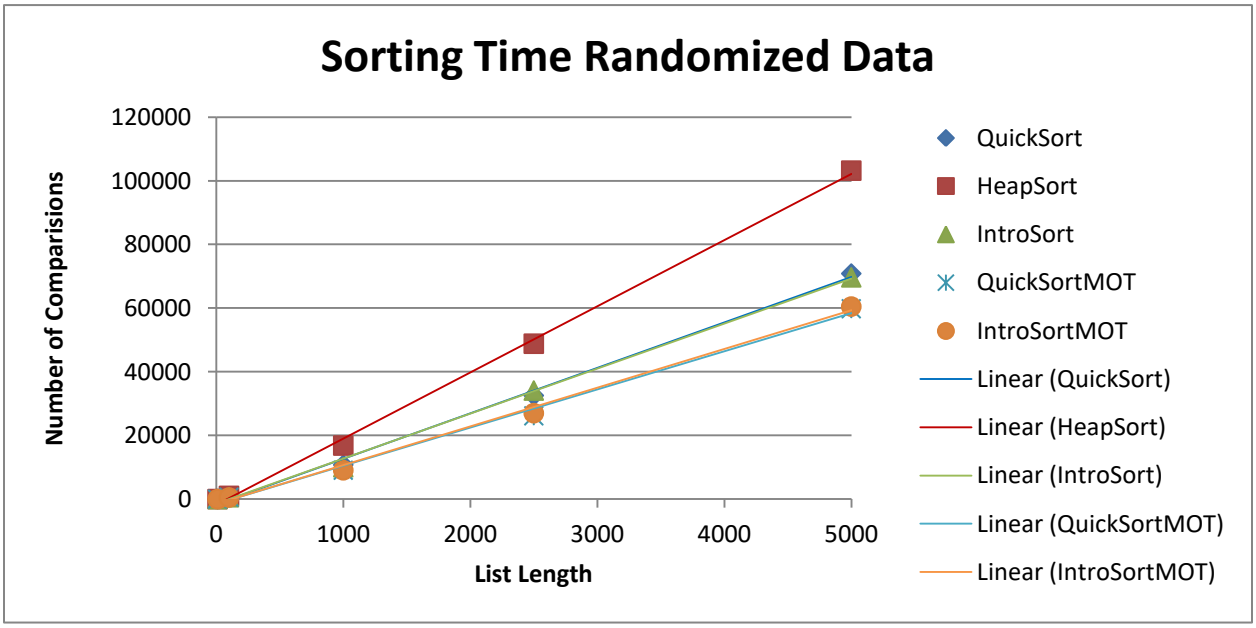
Runtime (IntroSort): We recall from previous sections that HeapSort had a runtime of O(nlgn), while QuickSort had a worst case runtime of O(n^2) and an average runtime of O(nlgn) regardless of which

value is used as the pivot. We also recall that QuickSort only had a runtime of O(n^2) when the recursion depth was O(n). Since we specify a recursion depth of 2*lgn, the recursion depth will not surpass O(lgn). This is an example of a hybrid sorting algorithm taking the best components of two or more sorting algorithms and producing a worst case runtime for IntroSort as O(nlgn).

Empirical Measurements:

Empirical measurements in this section are taken not by measuring the runtime of each algorithm, as that is prone to error, but by measuring the number of comparisons made by each sorting algorithm. A "comparison" is made whenever two values in the list are compared against each other to logically determine whether either of the two elements need to be moved. For example, this is done in QuickSort when checking a value against the pivot to see if we should put the value to the right or left of the pivot. This is also done in HeapSort during heapify() to check if a child node has a value greater than its parent.

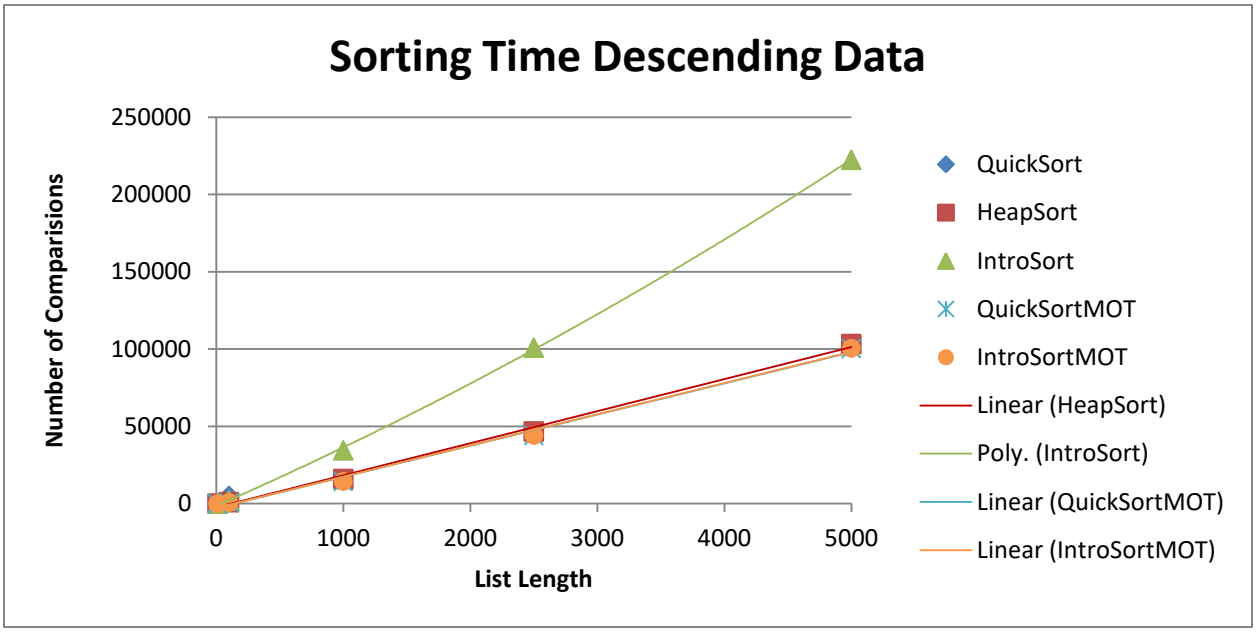
To get a full picture of the runtime for each of the sorting algorithms, we run the sorting algorithms with randomized values and descending values. This will expose algorithms which are not as good dealing sorting descending values such as QuickSort and give us an idea of how each algorithm does in a wider range of scenarios. First we will look at the data for randomized values

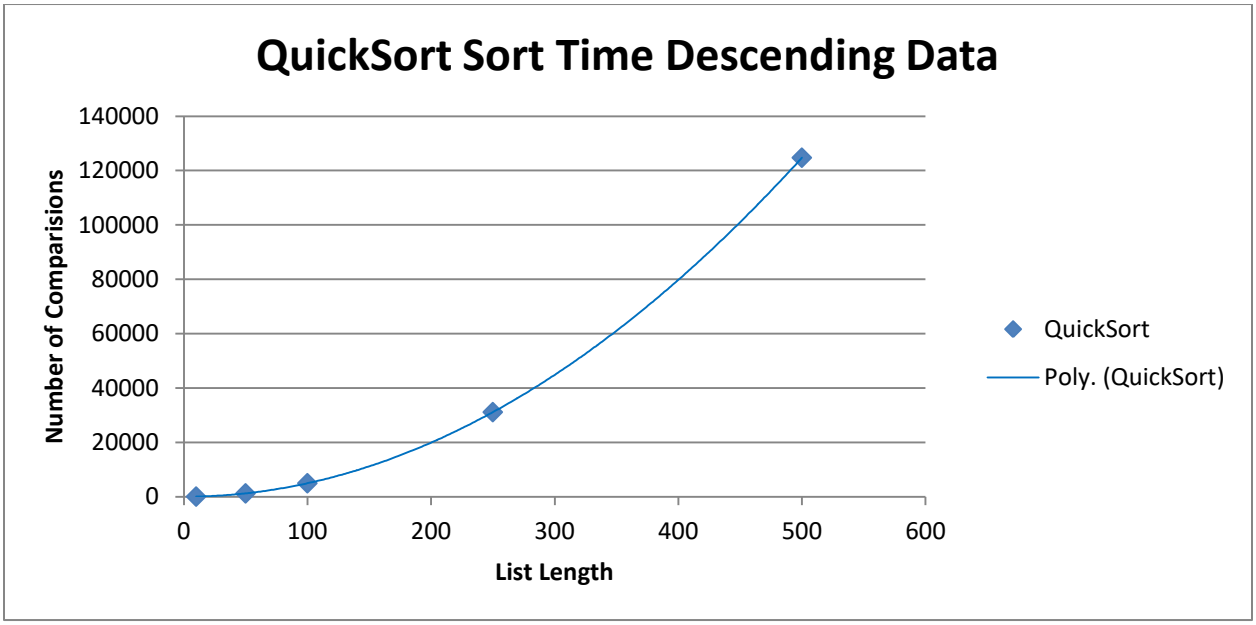


We first note that for each of the algorithms analyzed, we recall that the runtime was theoretically O(nlgn) As the O(lgn) term is taken over by the O(n) term as the list size grows, we expect to see a linear characteristic for each of the sorting algorithms. As suspected, each of the algorithms analyzed have a pretty good linear fit for randomized data.

Also of interest for randomized data, HeapSort is the slowest of the algorithms. This makes some sense, as it is known that QuickSort typically performs better than HeapSort given an array of random data, and the other four sorting algorithms are some deviation of QuickSort. It is also interesting to see that the sorting algorithms using the median of three approach perform better than the algorithms that do not utilize median of three. This is also expected, since choosing the median of three will likely get you a better pivot, thus reducing the number of comparisons that you have to make against the pivot.

Next we look at the number of comparisions for descending data:





Brian Loughran
Programming Assignment #2 Analysis

For the algorithms analyzed, we should expect QuickSort and IntroSort to have O(n^2) runtime for descending data, while we should expect all the others to have O(nlgn) runtime. While the theoretical worst case runtime for QuickSortMOT and IntroSortMOT is O(n^2), we note that descending data is not the worst case for these two algorithms. The worst case runtime for these algorithms would have the highest three values (or lowest three) located at the first index, median index, and high index for each recursive step. For descending data these two algorithms will pick the middle element as the pivot, which will split the array in half, resulting in O(nlgn) runtime for descending data. Thus, as the O(lgn) term is taken over by the O(n) term, we see a nice linear characteristic for HeapSort, QuickSortMOT, and IntroSortMOT. For IntroSort and QuickSort we see a nice O(n^2) characteristic that we would expect for the worst case runtime for these algorithms.

One thing to note is that QuickSort has no data for list lengths greater than 500. This is because if the list length exceeds 500 the recursion depth limit is reached, and the program fails. We note that IntroSort will help to prevent that a bit by switching to HeapSort when the recursion depth gets too large, but as the list size grows further eventually the IntroSort recursion depth will be exceeded as well, since the recursion depth increases as the number of elements in the list increases. However, that is not the case for a list with less than 5000 elements, clearly.

Thus our empirical data matches with our theoretical runtimes, giving us a decent indication that the theoretical runtimes are correct. You can see the raw data for each of the charts in the appendix for reference, or view in EmpericalData.exe.

Brian Loughran
Programming Assignment #2 Analysis

Appendix (Raw Runtime Data):

	QuickSort		HeapSort		IntroSort		QuickSortMOT		IntroSortMOT	
List Length	Rand	Desc	Rand	Desc	Rand	Desc	Rand	Desc	Rand	Desc
10	25	45	36	35	21	45	21	19	19	19
100	626	4950	1028	944	614	2010	623	686	519	686
1000	10772	-	16848	15965	10034	34478	9091	14378	9086	14380
2500	32570	-	48847	46694	34135	100897	26193	44086	27023	44114
5000	70888	-	103227	103227	69703	222458	59840	100346	60515	100530

List Length	QuickSort
10	45
50	1225
100	4950
250	31125
500	124750