Comparing HeapSort with Previous Algorithms and Their Associated Run Times
Lab Assignment #4

by
Justin Anthony Timberlake

CS 303 Algorithms and Data Structures
September 20, 2017

1. **Problem Specification**

The purpose of this lab was to first implement heapSort and then test it with the same values as the previous labs were tested with. This allowed a direct comparison to be made between mergeSort, insertionSort, and heapSort, as documented below. Unlike prior examples, this lab was made with object-oriented design as it involved a specific data structure, the heap which was to be sorted.

2. **Program Design**

The steps taken to develop the program were:

a) Design the “heap” class with the private variables “A” and “size”, which correspond to the array to be sorted as a heap and the size of the associated array respectively.

b) Copy over the part of the prior lab assignments to read in the same values from the files from before.

c) Develop the heapSort algorithm within the heap class, which starts by calling buildMaxHeap from the array provided in the construction of the class itself. Then continue to develop the buildMaxHeap method, which then calls maxHeapify. Finally, develop the maxHeapify method, all from the algorithms provided in class.

d) Develope the printing out statements much like before to show that the list of 100 values and 1,000 values could properly be sorted with my implementation of the heapSort algorithm in an object-oriented program set up.

e) For the homework portion of this lab, I had to change the class name of “heap” to “heapH” so that there was not an error with my workspace in eclipse working with a pre-existing class. Since I upload my lab and lab homework portions separately, I wanted to make sure both java files have their own heap classes included, even though they are exactly the same.

f) Add on the timing statements for ascertaining the number of milliseconds spent for each operation of heapSort.

The following methods were used in the main method:

a) readInNums (Scanner derp, int size) : receives a Scanner object, locally referred to as “derp” as well as a value for the size of the intended read in array to accommodate each input text file.

b) printArray(int [] list) : prints any given array of integers sent to it.

The following methods were used in the heap class:

a) heap() - basic constructor for an empty heap object

b) heap(int[] list, int num) – basic constructor for a heap object, setting the private array A equal to list and the private int size equal to num to initiate the values of the heap.

c) left(int i) – receives an integer "i" and returns 2*i + 1. This is used to return the index of the left child of any non-leaf node in a heap.

d) right(int i) – receives an integer "i" and returns 2*i + 2. This is used to return the index of the right child of any non-leaf node in a heap.

e) heapSort() - first calls buildMaxHeap, which will build a heap out of the array that is part of the heap properties (A). Then, for every value from the length of A minus 1 through zero, a loop decrements and swaps A[i] with A[0] followed by a maxHeapify call on the first element of the array (0) in order to maintain the heap properties.

f) buildMaxHeap() - first declares the size equal to the length of A. Then, a for loop decrements from half the value of size to 0, wherein it calls maxHeapify on all values of "i".

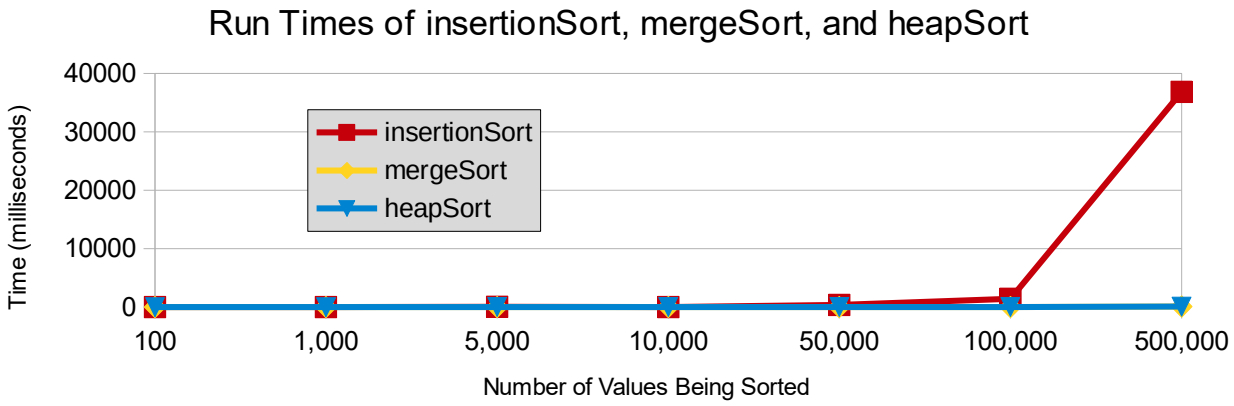
g) maxHeapify(int i) – This recursive method is the bulk of the actual comparisons performed within heapSort. First, the values of the left and right children of the current node index "i" are calculated (l and r). If the left child's index is less than the size of the array and the element at the index of l is greater than the current node, the "largest" value, which corresponds to the largest element within the array, has its index set equal to l. If these conditions are not met, largest is set equal to "i", meaning that the current node's index corresponds to the largest value. Then the same comparison is made for the right child, yielding largest = r if the conditions are met. Finally, if largest is not equal to "i" the value at the index of "i" is swapped with the value at the index of largest within the A array in the heap object before maxHeapify recursively calls itself at the position of "largest" as the new index to look at next.

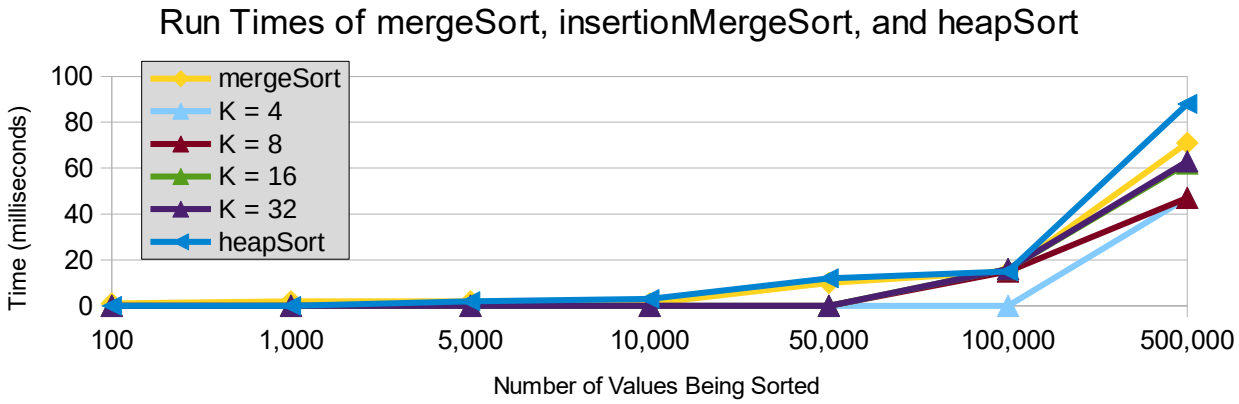
3. **Testing Plan**

The only testing needed for this lab was to print out the corresponding values of the associated run times with each value N of the size of the arrays, which have been sorted using different algorithms, which were described and performed in previous lab assignments to include insertionSort, insertMergeSort, and mergeSort. The time reported for the run time of each of these was already reported in milliseconds and can now be compared in a direct quantifiable means with the latest tests in run times with the same lists for heapSort. The graph and chart comparing the run times is documented below.

4. **Test Cases**

Insertion/MergeSort Run Times and Variations							
Input Size (number of values)	Time (milliseconds)						
	insertionSort	insertMergeSort, cutoff at values K				mergeSort	heapSort (max heap)
		K = 4	K = 8	K = 16	K = 32		
100	0	0	0	0	0	1	0
1,000	4	0	0	0	0	2	0
5,000	33	0	0	0	0	2	2
10,000	24	0	0	0	0	1	3
50,000	371	0	0	0	0	10	12
100,000	1,412	0	15	16	15	15	15
500,000	36,819	47	47	62	63	71	88





5. **Analysis and Conclusions**

From these graphs and tables, we can conclude that insertionSort very clearly works better on smaller values and quickly expands in run time in stark contrast to the other sorting algorithms. Along with the comparisons with previous examples, at much larger values, heapSort is still a much better choice than insertionSort. While the apparent runtime is very slightly worse off than all the insertMergeSort and mergeSort trials, it is known that they are functionally the same runtime in all cases (O (n lg n)). However, heapSort is actually in-place and therefore uses a much smaller amount of space complexity (not observable in this specific report, but worth noting) for relatively the same runtime.

6. **References**

All code in this assignment was either presented in prior labs and assignments or provided through means of pseudocode within the in-class portion of the lab.

7. **Recurrences**

1. T(n) = T(n-1) + a*n, if T(1) = 1 and a is a constant

T(n-1) = T (T(n -1) - 1) + a(n-1)

T(n-1) = T(n-2) + a(n-1)

T(n) = T(n-2) + a(n-1)

T(n-2) = T(T(n-2) – 1) + a(n-2)

T(n-2) = T(n-3) + a(n-2) + a(n-1)

T(n-3) = T(T(n-3) – 1) + a(n-3)

...

T(n) = T(n-3) + a(n-2) + a(n-1) + a(n)

T(n) = T(1) + a(2) + … + a(n-2) + a(n-1) + a(n)

T(n) = 1 + a(1 + 2 +...+ (n-1) + (n))

T(n) = 1 + a [n(n+1)/2]

$T(n) = an^2/2 + a*n/2 + 1$

The biggest term is n^2, so $\theta(T(n)) = n^2$.

2. T(n) = T(n-1) + a, if T(1) = 1 and a is a constant

$T(n-1) = T(T(n-1) - 1) + a$

$T(n-1) = T(n-2) + a + a$

$T(n) = T(n-2) + 2a$

$T(n-2) = T(T(n-2) -1) + 2a + a$

$T(n-2) = T(n-3) + 3a$

$T(n) = T(n - k) + ka$

if $k = n-1$

$T(n) = T(n - (n-1)) + (n-1)a$

$= T(1) + (n-1)a$

$= 1 + (n-1)a$

The biggest term is n, so so $\theta(T(n)) = n$.