Comparing How Runtimes Grow for Heap, Merge, and Insertion Sort
Lab #4

By
Michael Harden

CS 303 Algorithms and Data Structures
September 14, 2021

1. Problem Specification

The goal of this lab was to implement the Heap Sort Algorithms and compare its runtime speed to the algorithms implemented in previous labs. This was to be done by sorting multiple files of increasing length, determine how the run time grows as the size of data being sorted grows, then compare and identify when to use Heap Sort algorithm vs a previously explored algorithm.

2. Program Design

This program requires two classes, Sorts and SortsDriver. The Sorts class contains the implementation for Heap Sort and the previous sorting algorithms, Merge and Insertion Sort. The SortsDriver class contains methods to parse the text files containing the integers to be sorted. The files can be parsed using any delimiter. The driver class also calculate the runtime for each algorithm to sorting all required files, and finally test the correctness of each sorting algorithm's implementation.

To implement the Insertion and Merge Sort algorithm, the same process was used as in lab #2 and #3

The Heap Sort algorithm was implemented with the following process.

a) Take the array to sort, and turn it in to a max heap, by starting at the end of the array and heapifying each element and its children until the end or the list is reached
b) If any elements must be swapped as the max heap is being build, then re-heapify for the children of the element which has been swapped.
c) Once a max heap is built, place the max element of the heap at the end of the array and swap it with the last element in the heap.
d) Re-heapify until the heap has a size of 1. At this point the array is sorted

To calculate the run times for each required file, the following procedure was followed.

a) Parse the content of the file into a list of integers, starting with the smallest file and for each iteration move towards the largest file.
b) Create an instance of the Sorts class and pass the list of parsed integers into the Sorts constructor's parameter.
c) Store the current time just before sorting. Once the list has been completely sorted, the difference between the time directly after sorting and the stored time just before sorting is calculated.
d) The time is logged to the console
e) Continue to the next iteration and sort the next required text file.
f) The file sizes range from 100 to 50,000 elements.
g) This process takes place for Merge, Heap, and Insertion Sort

The following constructor and methods are defined within the Sorts class.

a) __init__(data):

   Defines the instance variable, data, and sets its value to the value passed in the data parameter.

b) Heap Sort is split into four methods.

   1. heap_sort():

      First, build_max_heap() is called. Then iterate through each element of the max heap, which is represented as an array, starting with the last index and ending at the second element of the array. For each iteration, due to the properties of a heap, the largest element is at the top of the heap (index 0) the max element is swap with the index of the current iteration, and the heap size is decreased by one. Finally, re-heapify the array, excluding the elements which fall at the current index and after as these have been sorted.

   2. build_max_heap():

      The size of the heap is calculated as the length of the array. Then iterate backward through each element of the heap and calling the max_heapify method on that element. Once the last iteration is complete, the array is in the form of a max heap

   3. max_heapify(i, heapSize):

      Calculates the left and right children of the element at parameter index i. The left child's index is calculated with (2*i) + 1 and the right child's is calculated with (2*i) + 2. The largest element between the indices i, left, and right is taken and if the largest element isn't at index i, the largest element is swapped with that at index i, and the max_heap is called on the element at the previous location of the largest index.

   4. swap(swapA, swapB):

      Swaps the value swapA with the value swapB in the array

c) Merge and Insertion sort are implemented in the same way as covered in the previous lab reports

The following constructor and methods are defined within the SortsDriver class.

a) test_heap(lst):

   This method tests the implementation of Heap Sort implementation by sorting the lists passed through the parameter and sorting it with Sorts.heap_sort(). The sorted list is then returned

b) All other methods are consistent with the descriptions provided in lab #3.

To parse the files, the open() and read() built in methods are used.

3. Testing

To test both sorting algorithms six test cases were used to attempts and break the heap sort algorithm in different ways. The first contains both negative and positive numbers. The second contains multiple duplicates of the same number. The third contains only negative numbers. The forth contains only one single element. The fifth is reverse sorted, and the last is an empty list.

Test Number	Input	Expected Output
#1	[10, 4, 6, 3, 2, 9, 16, 0, 3, -1]	[-1, 0, 2, 3, 3, 4, 6, 9, 10, 16]
#2	[4, 3, 3, 3, 2, -2]	[-2, 2, 3, 3, 3, 4]
#3	[-3, -103, - 5, -2, -10, -44, -31]	[-103, -44, -31, -10, -5, -3, -2]
#4	[10]	[10]
#5	[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
#6	[]	[]

The file provided “input_100.txt” was also used to test the Heap Sort implementation and the input was sorted correctly in increasing order

4. Timing

The times to sort the provided files are as follows:

File name	**Insertion Sort Time** (seconds)	**Merge Sort Time** (seconds)	**Heap Sort Time** (seconds)
input_100.txt	0.0064421	0.000217915	0.000689983
input_1000.txt	0.084224425	0.003399849	0.010570049
input_5000.txt	1.81907511	0.019459009	0.062674999
input_10000.txt	6.46251702	0.041306019	0.122979879
input_50000.txt	182.156554	0.24689889	0.77559495

5. Analysis and Conclusions

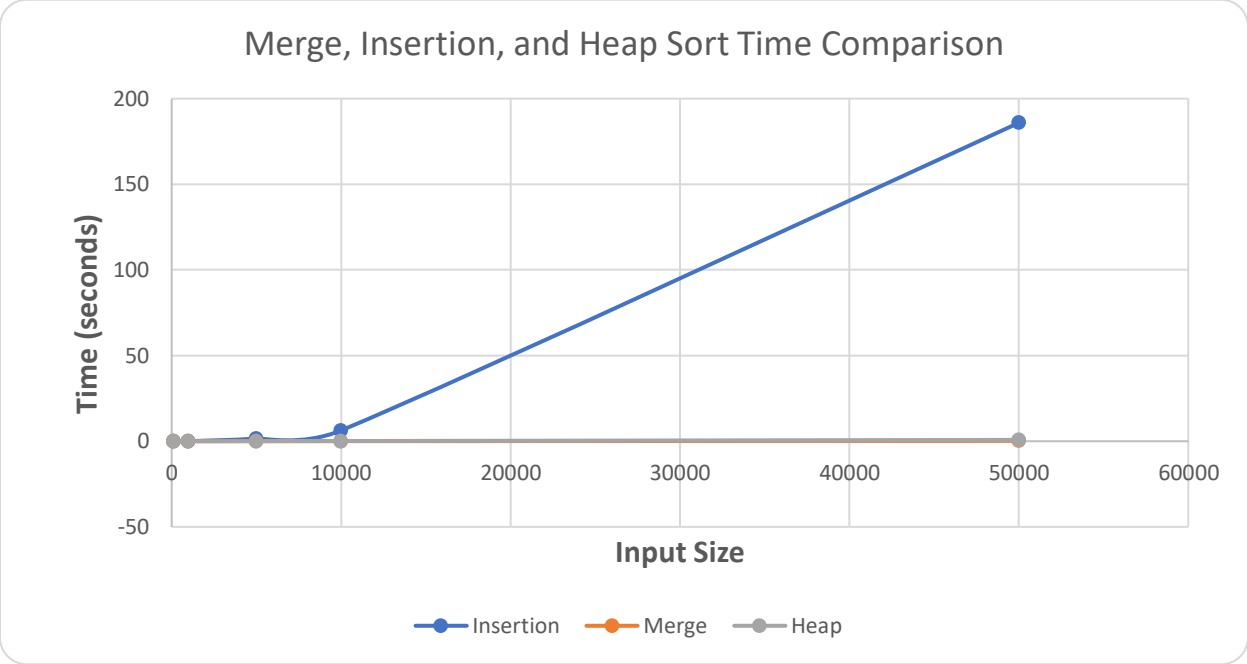
In conclusion, the times show how heap sort is very comparable to merger sort as far as how the time grows. Heap sort is a far superior sorting algorithm to Insertion Sort. As

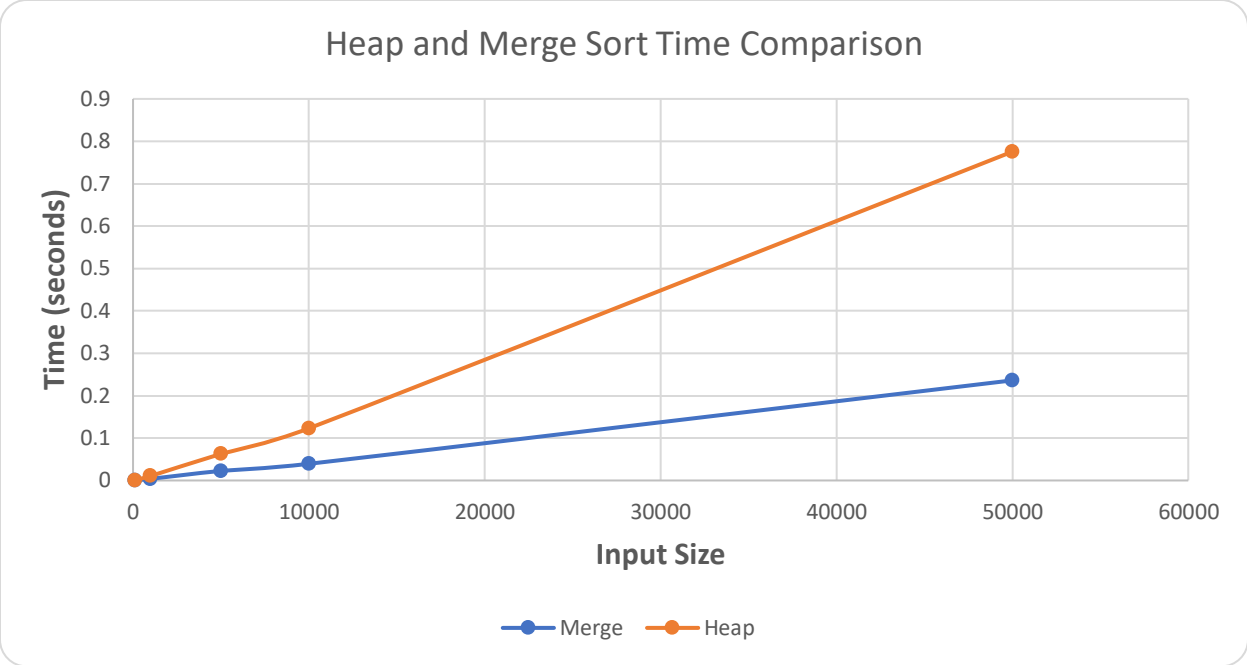
Heap Sorts Big-O time complexity is O(n log n) and insertion sort has a worst case time complexity of O(n^2). This holds particularly true when the number of elements reaches into the tens of thousands.

Although Heap Sort has the same Big-O time complexity of Merge Sort, Merge Sort is still considerably faster for large data sets, as it requires less swaps and comparisons.

The place where Heap Sort is superior to Merge, is its space complexity, which is O(1). Because it is an in place sorting algorithm and needs no auxiliary arrays.

The following graphs displays how the sorting time grows in relation to a linearly growing input size. The first chart compares the time complexity of Heap, Merge and Insertion Sort. The second chart only compares Heap and Merge sort, as their differences are not great enough to show up when comparing to Insertion Sort.
(data collected with provided text files):





6. **Credits**
   - For the Heap Sort algorithm, the pseudo code in the lab4 document was used.
   - Parts of my lab report 3 were also used in this report.
   - For the Report the sample report provided on canvas was used as a guide.