Computing Merge & Insertion Sort using Python

Homework #3

By

Mario Pendleton

CS 303 [Algorithms and Data Structures](https://uab.instructure.com/courses/1507655)

September 8, 2019

### Problem Specification

Implement a method that will sort a given array using the merge sort algorithm. Write a driver program to test the insertion algorithms implemented. Read the input file “input\_100.txt” for the input numbers and store them in an array. Sort this array using  merge sort. Test the program for the different size input files and compare the performance of insertion sort to that of merge sort. Record the runtime for merge sort on various sized arrays by using the provided files.  Comment on how the execution time of merge sort varies with size of the input array. Use a table or plot to summarize the results and document your observations and explanations in the report.

1. Implement a method to sort a given array using the merge sort algorithm.
2. Write a driver program to test the merge sort algorithm for the arrays of varying lengths provided in Canvas.
3. Compare the execution time of merge sort with insertion sort implemented in Lab-2. Make sure you use the same array to compare the performance. Use a table or plot to summarize the results and document your observations and analysis in the report.
4. Based on the performance modify the merge sort algorithm such that when the array size gets small enough, you would invoke insertion sort instead of invoking merge sort, Instead of hardcoding the array size make it a variable that you can pass as an argument to the merge sort method and test this cutoff value with at least four different values.
5. Test the program for the same array sizes and values. Compare the performance with the original merge sort implementation, plot the execution times, and document the analysis in your lab report.

### Program Design

This program requires an array of data that will be sorted using the merge sort method. The method was designed in python after the pseudo below.

MERGE-SORT (A, temp, p , r) if p < r

q = |\_ (p + r) / 2 \_| MERGE-SORT (A, temp, p , q) MERGE-SORT (A, temp, q + 1, r) MERGE (A, temp, p, q, r)

//////////////////////////////////////////////

MERGE (A, temp, p, q, r) // merge A[p..q] with A[q+1..r] i= p j = q +1

// copy A[p..r] to temp[p..r] for k = p to r

temp[k] = A[k]

//merge back to A[p..r] for k = p to r

if i > q // left half empty, copy from the right A[k] = temp[j]

j = j+ 1 else if j > r // right half empty, copy from the left

A[k] = temp[i]

i = i +1 else if temp[j] < temp[i]

A[k] = temp[j]

j = j+ 1 else

A[k] = temp[i] i = i +1

// copy from the right

// copy from the left

The following steps were required to develop this program:

1. Re-write the recursive merge sort pseudo code to a python methods (mergeSort(x,y,l,r)) merge(x,y,l,m,r) an.
2. Use the a method to read the following txt files and covert them into arrays to be sorted

* input\_100.txt
* input\_1000.txt
* input\_5000.txt
* input\_10000.txt
* input\_50000.txt
* input\_100000.txt
* input\_500000.txt

The following methods were defined within the lab2.py:

1. insertionSort (list)

Method that clones an array then performs an insertion sort on the original array. It then takes the cloned array and performs a python sort (foo.sort()). The method returns True if the insertion sort array is equal to the python sort array. If False, the method will print the python sort results for comparison.

Constructor that creates an instance of an IdealWeight object, setting the feet and inches variables to the user inputs.

1. sampleList (n)

Method to create a random array of n^2 integer characters.

1. load (path): int

Method that opens a txt file and creates an array.

1. mergeSort(x,y,l,r,s)

Recursive method that takes in 2 arrays (x,y) and 2 integers (l,r). This method is responsible for mathematically breaking down each character of array x into separate arrays and defining the mid-point.

1. merge(x,y,l,m,r)

Method that clones array x resulting in array y. The characters of array x are then compared individual and merged back together using the merge sort algorithm.

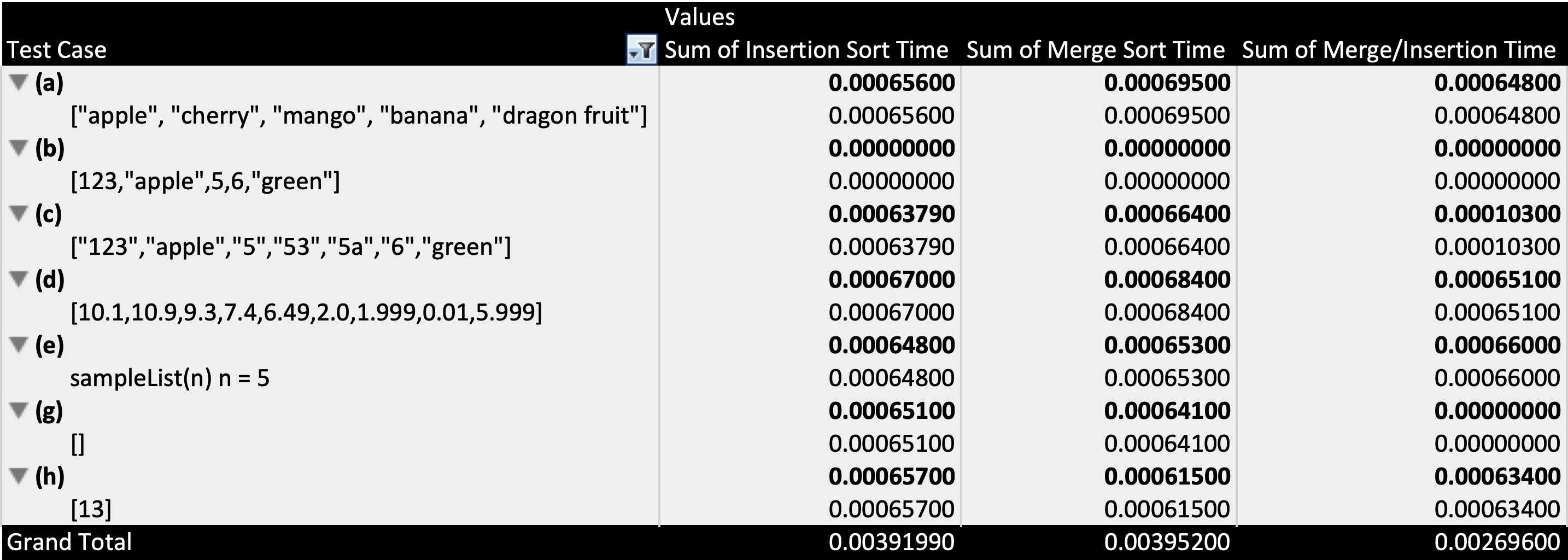
### Testing Plan

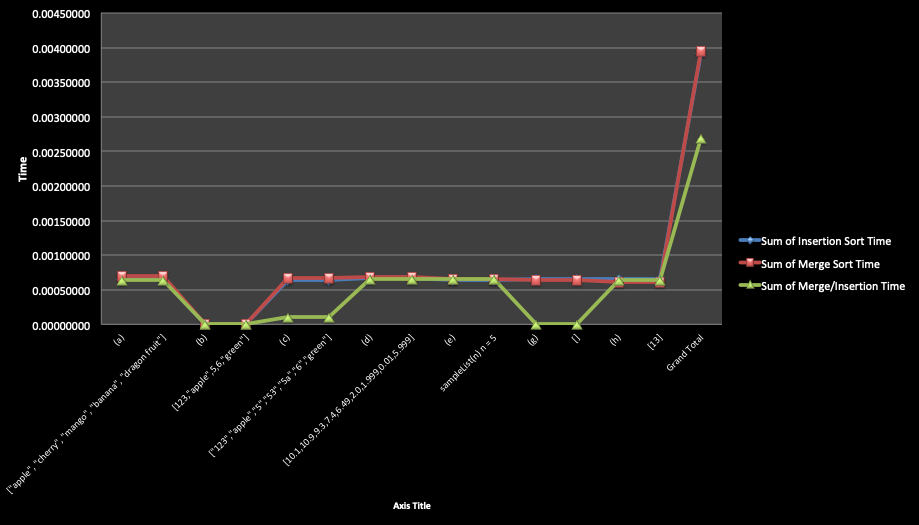
Sample string inputs were selected to see the program could sort (a) none integer values, (b) string and integer values, (c) string integers with strings, (d) floats, (e) random array of integers of 2^n in size, (f) values loaded from the txt files, (g) empty array, (h) single integer, (i) single string, and (j) list of positive and negative integers.

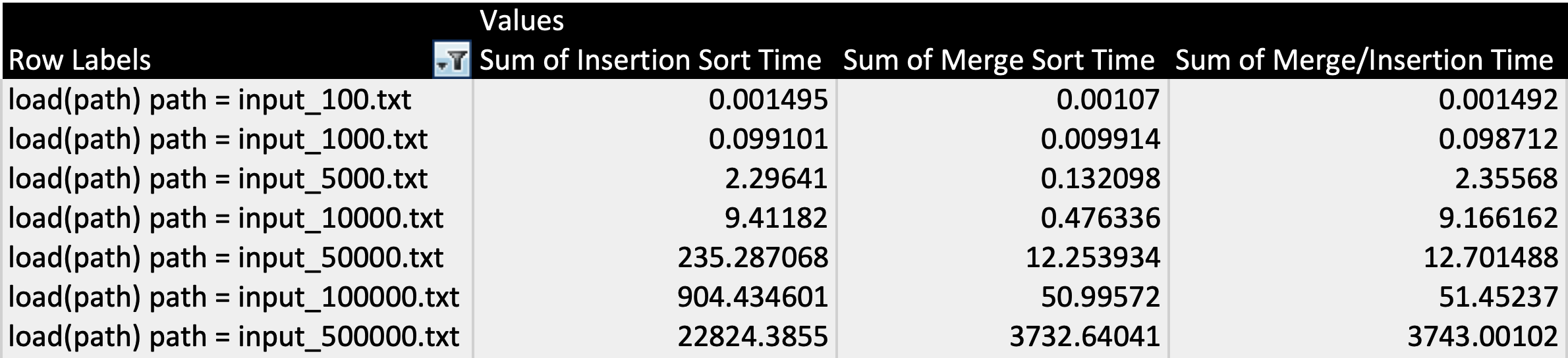
### Test Cases

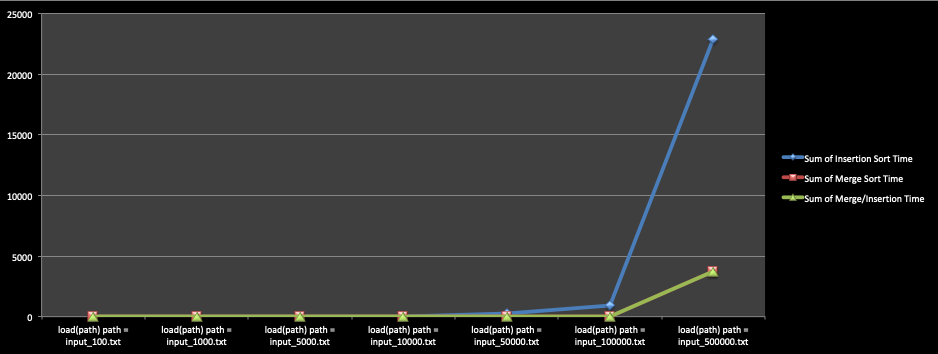
The test cases are shown in the table below using a MacBook Pro 16GB, 8 core 2.3 GHz Intel Core i9:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Case | Input Values | Expected Output | Actual Output | Insertion Sort Time | Merge Sort Time | Merge/Insertion Time |
| (a) | ["apple", "cherry", "mango", "banana", "dragon fruit"] | ['apple', 'banana', 'cherry', 'dragon fruit', 'mango'] | ['apple', 'banana', 'cherry', 'dragon fruit', 'mango'] | 0.000656 | 0.000695 | 0.000648 |
| (b) | [123,"apple",5,6,"green"] | not supported between instances of 'int' and 'str' | not supported between instances of 'int' and 'str' | N/A | N/A | N/A |
| (c) | ["123","apple","5","53","5a","6","green"] | ['123', '5', '53', '5a', '6', 'apple', 'green'] | ['123', '5', '53', '5a', '6', 'apple', 'green'] | 0.0006379 | 0.000664 | 0.000103 |
| (d) | [10.1,10.9,9.3,7.4,6.49,2.0,1.999,0.01,5.999] | [0.01, 1.999, 2.0, 5.999, 6.49, 7.4, 9.3, 10.1, 10.9] | [0.01, 1.999, 2.0, 5.999, 6.49, 7.4, 9.3, 10.1, 10.9] | 0.000670 | 0.000684 | 0.000651 |
| (e) | sampleList(n) n = 5 | Random array of 8 integers  [8, 7, 3, 5, 8, 3, 2, 5] | Random array of 8 integers sorted  [2, 3, 3, 5, 5, 7, 8, 8] | 0.000648 | 0.000653 | 0.000660 |
| (f) | load(path) path = input\_100.txt  [4, 50, 34, 40, 22, 54, 94, 3, 94, 38, 8, 95, 0, 36, 54, 54, 81, 30, 24, 98, 12, 25, 43, 0, 52, 52, 88, 22, 83, 70, 96, 57, 89, 53, 13, 64, 74, 18, 37, 86, 73, 76, 15, 1, 93, 69, 77, 81, 29, 78, 14, 45, 67, 1, 0, 41, 60, 63, 74, 16, 75, 75, 36, 49, 68, 5, 67, 29, 15, 84, 47, 77, 40, 80, 24, 61, 25, 7, 85, 83, 81, 47, 10, 39, 22, 72, 87, 64, 92, 27, 50, 69, 12, 54, 23, 85, 38, 75, 73, 94] | [0, 0, 0, 1, 1, 3, 4, 5, 7, 8, 10, 12, 12, 13, 14, 15, 15, 16, 18, 22, 22, 22, 23, 24, 24, 25, 25, 27, 29, 29, 30, 34, 36, 36, 37, 38, 38, 39, 40, 40, 41, 43, 45, 47, 47, 49, 50, 50, 52, 52, 53, 54, 54, 54, 54, 57, 60, 61, 63, 64, 64, 67, 67, 68, 69, 69, 70, 72, 73, 73, 74, 74, 75, 75, 75, 76, 77, 77, 78, 80, 81, 81, 81, 83, 83, 84, 85, 85, 86, 87, 88, 89, 92, 93, 94, 94, 94, 95, 96, 98] | [0, 0, 0, 1, 1, 3, 4, 5, 7, 8, 10, 12, 12, 13, 14, 15, 15, 16, 18, 22, 22, 22, 23, 24, 24, 25, 25, 27, 29, 29, 30, 34, 36, 36, 37, 38, 38, 39, 40, 40, 41, 43, 45, 47, 47, 49, 50, 50, 52, 52, 53, 54, 54, 54, 54, 57, 60, 61, 63, 64, 64, 67, 67, 68, 69, 69, 70, 72, 73, 73, 74, 74, 75, 75, 75, 76, 77, 77, 78, 80, 81, 81, 81, 83, 83, 84, 85, 85, 86, 87, 88, 89, 92, 93, 94, 94, 94, 95, 96, 98] | 0.001495 | 0.001070 | 0.001492 |
| (f) | load(path) path = input\_1000.txt | (Data) Insertion sort was successful. No errors found.  True | (Data) Insertion sort was successful. No errors found.  True | 0.099101 | 0.009914 | 0.098712 |
| (f) | load(path) path = input\_5000.txt | (Data) Insertion sort was successful. No errors found.  True | (Data) Insertion sort was successful. No errors found.  True | 2.29641 | 0.132098 | 2.35568 |
| (f) | load(path) path = input\_10000.txt | (Data) Insertion sort was successful. No errors found.  True | (Data) Insertion sort was successful. No errors found.  True | 9.41182 | 0.476336 | 9.166162 |
| (f) | load(path) path = input\_50000.txt | (Data) Insertion sort was successful. No errors found.  True | (Data) Insertion sort was successful. No errors found.  True | 235.287068 | 12.253934 | 12.701488 |
| (f) | load(path) path = input\_100000.txt | (Data) Insertion sort was successful. No errors found.  True | (Data) Insertion sort was successful. No errors found.  True | 904.434601 | 50.99572 | 51.45237 |
| (f) | load(path) path = input\_500000.txt | (Data) Insertion sort was successful. No errors found.  True | (Data) Insertion sort was successful. No errors found.  True | 22824.3855 | 3732.64041 | 3743.00102 |
| (g) | [] | [] | [] | 0.000651 | 0.000641 | 0/000637 |
| (h) | [13] | [13] | [13] | 0.000657 | 0.000615 | 0.000634 |
| (i) | [“red”] | [‘red’] | [‘red’] | 0.000670 | 0.000628 | 0.000646 |
| (j) | [-245,245,-1,1,0,-45,45,2,-2,3,-3] | [-245, -45, -3, -2, -1, 0, 1, 2, 3, 45, 245] | [-245, -45, -3, -2, -1, 0, 1, 2, 3, 45, 245] | 0.000681 | 0.000662 | 0.000665 |









### Analysis and Conclusions

Merge sort was successfully computed for all listed test cases. Test case (b) failed as expected. Testing confirmed that the algorithm was able to properly sort the known test cases. Combining merge sort with insertions improves the quality of merge sort by reducing memory consumption. Defining a point where the algorithm swaps over to merge sort should be based on the need of the client. In cases where memory is in abundance merge sort is favorable. In case were clients rely on core speed insertion sort is useful but has its limitations. The speed of merge sort did not prove significantly faster than insertion sort until I got to file size 10000 clocking at a speed of 9.166162. At this point I chose to convert my algorithm to merger sort to improve speed. Merge sort was able run file size 50000 at a speed of 12.701 which is the nearly the same speed in which insertion sort handled 10000.

### References

Textbook, python.org, and examples provided in the assignment.

**Screen Shot**

