Merge Sort Algorithm

Cole Davis
Dept. of Computer Science
Missouri University of Science and Technology

Motivation

This report contains the results and observations of experiments conducted with two common sorting algorithms used in computer science, Merge Sort and Insertion Sort. When mathematically analyzed, Merge Sort and Insertion Sort are found to sort the same data set at different rates, Merge Sort being the more efficient of the two. Merge Sort's time complexity is **O(n*log(n))** while Insertion Sort is **O(n^2)** where n represents the number of elements in the data set to be sorted. The motivation for this experiment was to test whether or not the mathematically derived time complexities accurately predicted the time taken by a computer to execute the algorithms. If the time taken to sort the data sets in the experiment are found to be within a reasonable margin of error of the time predicted by the time complexities, the validity of mathematically derived time complexities will be further cemented. Thus, Merge Sort should be found the be the more efficient sorting algorithm.

Background

Merge Sort and Insertion Sort are two algorithms that accomplish the same goal, sorting data in ascending or descending order. However, Merge Sort is mathematically predicted to accomplish this faster. Merge sort takes advantage of the "divide and conquer" paradigm in order to get a more efficient runtime complexity. On the other hand, Insertion sort has a quadratic runtime complexity. The experiment conducted was testing whether or not Merge Sort truly was the faster of the two algorithms. Because these two algorithms perform the same functionality, they could be implemented interchangeably. However, it is generally true that the more efficient algorithm should be the one that's implemented. A real world example of sorting algorithms being important to the everyday person is the common feature found in music streaming services; sorting by artist. The user of the streaming service would like to see the listing of songs sorted by artist in as little time as possible. Depending on which sorting algorithm the company who produced the music streaming service chose to implement, it is predicted the time it takes for the final listing of songs by artist will vary.

Procedures

Pseudocode for Insertion Sort:

```
A = {set of integers}
index = 1
while( index < length(A))      //Every element will have the inner while loop applied to it
    dec = index
    while( dec > 0 and A[dec-1] > A[dec])         //Work right to left, finding correct
        switch positions of A[dec] and A[dec-1]       //location for A[dec]
        dec = dec – 1
```

Pseudocode for Merge Sort:

```
Arr = {set of integers}
function merge_sort(list Arr)
{
    if (length of Arr < 1)
        return;

    Left = empty list;
    Right = empty list;

    merge_sort(Left)        //Left list recurse
    merge_sort(Right)       //Right list recurse


    //Merge the two lists into one bigger list
    while(Left and Right both still have elements in them)
    {
        if(L’s next element < R’s next element)
            Add L’s next element to Arr;
        else
            Add R’s next element to Arr;
    }

    //Copy remnants of L if R is emptied first
    while(left still has elements in it and right doesn’t)
    {
        Add the rest of L’s elements to Arr
    }

    //Copy the remnants of R if L emptied first
    while(right still has elements in it and left doesn’t)
    {
        Add the rest of R’s elements to Arr
    }
}//End of function merge_sort()
```

Description of Tests:

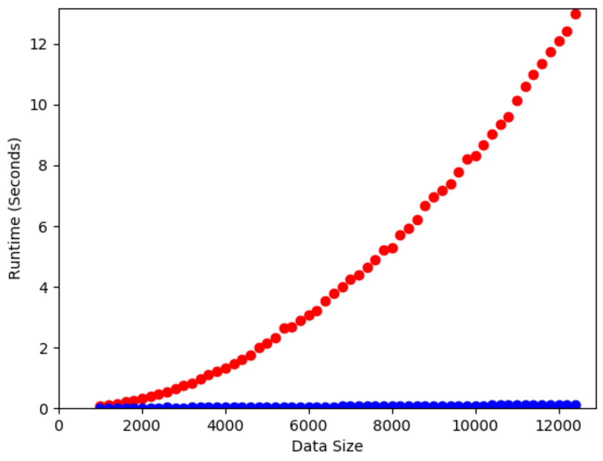
Test #1 and Test #2
Tests 1 and 2 were conducted within the parameters numbered below. The only differences between the tests were the data set sizes and the increment by which the data size increased. As you can see from the graphs below, Insertion Sort appears to be Inferior to Merge Sort. Both graphs exhibit behavior similar to what's predicted by the O(n^2) and O(n*log(n)) expressions. Therefore, it can be concluded that mathematically derived time complexities accurately describe the behavior of algorithms when implemented on a computer.

1) The programming language used in the experiments was Python 3.7.0
2) Array elements were randomly generated integers between 0 and 1000 inclusive
3) Runtime was determined by using Python's "time" library.

🔴 = **Insertion Sort**

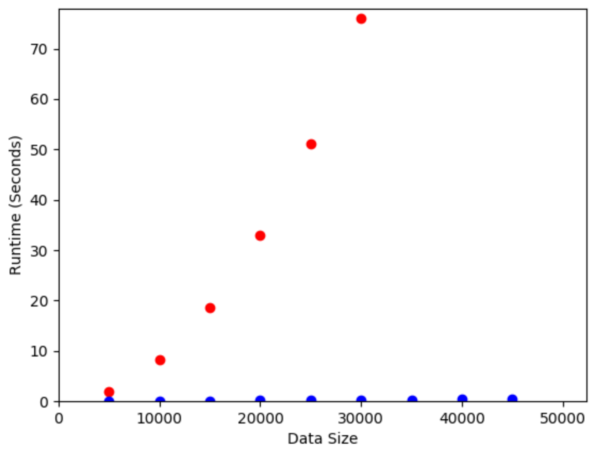
🔵 = **Merge Sort**

Test #1 Results (Random Values)



Increment = 200 Range = 1,000-12,600

Test #2 Results (Random Values)



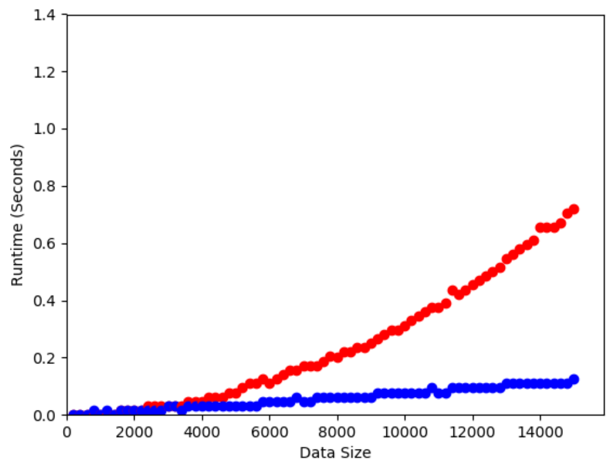
Increment = 5,000 Range = 5,000-45,000

Tests #3 and #4
Tests 3 and 4 were conducted with nearly sorted input. The data set was generated by simply incrementing through a for loop and appending the value of the incrementer to the list. The mod operator was used to select every 50 append operations and place a value of 0 in that spot instead of the for loop's incrementer value.

🔴 = Insertion Sort

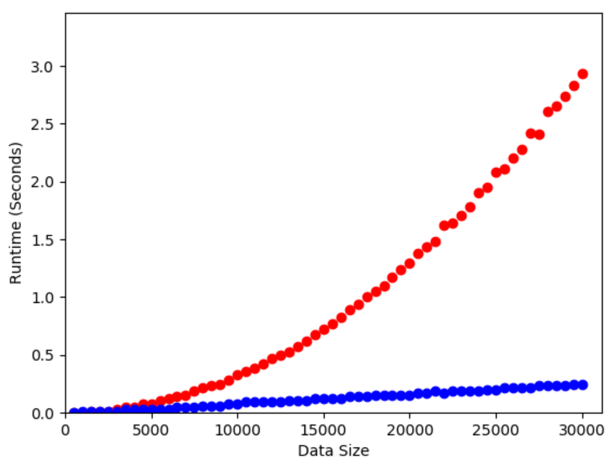
🔵 = Merge Sort

Test #3 Results (98% Sorted)



Increment = 200   Range = 200-15,000

Test #4 Results (98% Sorted)



Increment = 500   Range = 500-30,000

Problems Encountered:
- Technical problems getting Python's graphing library to install correctly (Matplotlib)
- Learning how to use Matplotlib
- Implementing the invariants correctly and making sure they threw an error when needed.

Conclusions:

This paper briefly described the differences between Merge Sort and Insertion Sort. The theoretical analysis of the algorithms showed to be accurate with the actual behavior of the algorithms when implemented on a computer. Below are two statements found to be true within the context of this experiment.

Merge Sort’s runtime ranged from 9 to 250 times more efficient with random data sets containing 1000 to 45,000 elements.

Merge Sort’s runtime ranged from .015 to 5 times more efficent with nearly sorted data sets containing 200 to 15,000 elements respectively.

Therefore, it can be concluded from these experiments that Merge Sort executes faster than Insertion Sort when the data set contains at least 200 elements that are no more than 98% sorted. This does not necessarily mean that Merge Sort continues to outperform Insertion Sort for smaller input sizes that are more than 98% sorted though. Experiments on data sets smaller than 200 elements were not performed. Additionally, it could be argued that Insertion Sort is a practical option for sorting small data sets that are nearly sorted, especially if the programmer’s language does not support recursion.

Appendix A – Source Code

```
'''////////////////////////////////////////////////////////////////////
/// Project1.py
/// Cole Davis, CS2500 Algorithms. Section 1A
/// This file contains the class Sort() as well as the drivers for it.
////////////////////////////////////////////////////////////////////
'''

import time
import math
import random

import matplotlib.pyplot as plt
plt.xlim([0, 50000000])
plt.ylim([0, 10000])
plt.ylabel('Runtime (Seconds)')
plt.xlabel('Data Size')




'''
////////////////////////////////////////////////////////////////////
/// Sort class
/// This class contains the code for merge and insertion sort.
/// It is constructed with a list of data to be sorted. With that
/// Data, you can choose to perform one of the two sorting operations
////////////////////////////////////////////////////////////////////
'''
class Sort():
    #Paramaterized Constructor
    def __init__(self, Array):
        #Here's the input set we're given
        self.Array = Array
        self.data_size = len(Array)

        self.merge_sort_time = 0
        self.insertion_sort_time = 0




    def do_merge_sort(self):
        #Make a copy of the data set for merge_sort to use
        merge_sort_arr = list(self.Array)

        #Time merge sort
        start = time.time()
        self.merge_sort(merge_sort_arr)
        end = time.time()

        #Plot the time taken and the data size
        self.merge_sort_time = (end-start)
        print("Merge sort time: ",str(self.merge_sort_time))
        plt.scatter(len(merge_sort_arr),self.merge_sort_time, color = "b")



    def do_insertion_sort(self):

        #Make a copy of the data set for insertion_sort to use
        insertion_sort_arr = list(self.Array)

        #Perform Insertion Sort
        start2 = time.time()
        self.insertion_sort(insertion_sort_arr)
        end2 = time.time()

        #Plot the time taken and the data size
        self.insertion_sort_time = (end2-start2)
        print("Insertion sort time: ",str(self.insertion_sort_time))
        plt.scatter(len(insertion_sort_arr),self.insertion_sort_time, color = "r")




    def insertion_sort(self, Arr):
        for i in range(1,len(Arr)):
            dec = i-1
            elem = Arr[i]
            while(dec >= 0 and Arr[dec] > elem):
                Arr[dec+1] = Arr[dec]
                dec = dec-1
            Arr[dec+1] = elem



    def merge_sort(self, Arr):
        #Base case
        if(len(Arr) > 1):
            middle = int(len(Arr)/2) #Casting to an int() because splice operator below requires ints, and
interpreter is typing it to a float for some reason

            #Splice to get our right and left sub arrays
            L = Arr[:middle]
            R = Arr[middle:]
            #Recursive Calls
            self.merge_sort(L)
            self.merge_sort(R)


            #Below this line is where the comparisons happen and the lists are sorted in ascending order
            left_ctr = 0
            right_ctr = 0
            total_ctr = 0

            #While there's still elements left in both lists
            while(left_ctr < len(L) and right_ctr < len(R)):
                if(L[left_ctr] <= R[right_ctr]):
                    Arr[total_ctr] = L[left_ctr]
                    left_ctr = left_ctr+1



                else:
                    Arr[total_ctr] = R[right_ctr]
                    right_ctr = right_ctr+1
                total_ctr = total_ctr+1 #Increment every time



            #The two while loops copy the rest of the elements in if one sub array was emptied before the
other. Only one of these while loops will execute.
            while(left_ctr < len(L)):
                Arr[total_ctr] = L[left_ctr]
                total_ctr=total_ctr+1
                left_ctr = left_ctr+1

            while(right_ctr < len(R)):
                Arr[total_ctr] = R[right_ctr]
                total_ctr = total_ctr+1
                right_ctr = right_ctr+1





#Appends an array into the Testing list
def Rand(iterations, lower_bound, upper_bound):
    arr = []
    for j in range(iterations):
        arr.append(random.randint(lower_bound, upper_bound))
    return arr

#Appends an array into the Testing list
def nearly_sorted(iterations):
    arr = []
    for j in range(iterations):
        if(j%50 == 0):
            arr.append(0)
        else:
            arr.append(j*2)
    print("array is ",len(arr))
    return arr





'''
Driver Below
'''
#Random Element Values
INCREMENTER1 = 5000
LOWER_BOUND1 = 0
UPPER_BOUND1 = 45000
for i in range(LOWER_BOUND1, UPPER_BOUND1, INCREMENTER1):
    Arr = Rand(i, 0, 1000)
    SortObject = Sort(Arr)
    print (i)
    print()
    print()

    #Call the sorts
    SortObject.do_insertion_sort()
    SortObject.do_merge_sort()


#Nearly Sorted Element Values
INCREMENTER1 = 500
LOWER_BOUND1 = 500
UPPER_BOUND1 = 30500
for i in range(LOWER_BOUND1, UPPER_BOUND1, INCREMENTER1):
    Arr = nearly_sorted(i)
    SortObject = Sort(Arr)
    print (i)
    print()
    print()


    #Call the sorts
    SortObject.do_insertion_sort()
    SortObject.do_merge_sort()


plt.show()
```