Tyler Cope

CS325 – Analysis of Algorithms

4/7/2017

Homework 1

Question 1

First, compare the two expressions. $8n^2 = 64n * \log n$ which is $n^2 = 8n * \log n = n = 8 * \log n$. Then you plug it into a calculator (Wolfrom Alpha was suggested by Timothy Reichmann in the discussion group) to see that when n is $1 < n < 44$, insertion sort beats merge sort.

Question 2

a. The limit as n approaches infinity of f(n) / g(n) is 0 so f(n) = O(g(n)).
b. The limit of the functions divided as n approaches infinity is infinity, so f(n) = Ω(g(n)).
c. The limit is 1 and since that's a constant, we know that f(n) = Θ(g(n)).
d. The limit is a positive constant so f(n) = Θ(g(n)).
e. The limit is 0 so f(n) = O(g(n)).
f. The limit is 0 so f(n) = O(g(n)), same as part e.
g. The limit of f(n) / g(n) is a positive constant so f(n) = Θ(g(n)).
h. Limit of the divided functions is zero so f(n) = O(g(n)).
i. Limit of the divided functions is zero so f(n) = O(g(n)).
j. Same as part h and part i. Limit is zero so f(n) =O(g(n)).

Question 3

If the array is odd, set the min and max value equal to the first term in the array. Set a starting index equal to 1 (since arrays are zero-indexed). If it's even, then set the min and max equal to the first two numbers in the array (min is the smaller and max is the larger of the two), and set the starting index to 2. Then as you iterate through the array, compare each number and do the assignment.

Pseudocode:

```
if array.length is even:
        if array element at first index > array element at second index:
                max = array at first element
                min = array at second element
        else:
                max = array at second element
                min = array at first element
        beg_index = 2
else:
        max = array at first element
        min = array at first element
        beg_index = 1
for every element in array from beg_index to end of array:
```

```
        if array at element (array[element]) < array[element + 1]:
            if array[element + 1] > max:
                max = array[element + 1]
            if array[element] < min:
                min = array[element]
        else:
            if array[element] > max:
                max = array[element]
            if array[element + 1] < min:
                min = array[element + 1]
    return max, min
```

This will perform at most 1.5 comparisons because at most you will need 3 * n/2 comparisons which is 1.5n (the first assignment of min and max, finding the max out of all numbers, finding the min out of all numbers). At each step we only need to compare half the numbers to max and min because if an element is smaller than its neighbor, we also know that it can't be the max number in the array; same process for the min. So if we were to pass an array with [9, 3, 5, 10, 1, 7, 12], the max and min would be set to 9 and the starting index would be 1 because there are an odd number of elements. Then we compare 3 and 5. 3 is less than 5 so we compare 3 to the min. The min is currently 9 so since 3 is smaller, it's updated to 3. 5 is less than the current max of 9 so the max is still 9. Now compare 5 and 10. 5 is less than 10 but greater

than the current min of 3, so three is still the min. However, 10 is greater than the current max of 9 so the max is now updated to 10. Continue this process until we reach the end of the array and it will return 12 as the max and 1 as the min.

Question 4

a. $f_1 = O(g(n))$ and $f_2 = O(g(n))$ both describe the upper bound of a function ($f_1(n) = c_1g_1(n)$ and similar for f_2). $\Theta(g(n))$ describes a tighter notation because it bounds the functions between two constants. This means that there could be a completely different lower bound between the two functions. Consider the functions $f_1 = n^3 + n^2$ and $f_2 = n^3 + n$. They have the same $O(n^3)$ because that is the upper bound but their lower order terms are different. Thus f_1 does not equal $\Theta(f_2(n))$.
b. If $f_1 = O(g_1(n))$ and $f_2 = O(g_2(n))$ then there exists constants, $c_1, c_2 > 0$ such that $f_1(n) <= c_1g_1(n)$ and $f_2(n) <= c_2g_2(n)$. Now we can use this information to set up an inequality: $f_1(n) + f_2(n) <= c_1g_1(n) + c_2g_2(n)$. Now we can also create a third constant, c_3 such that c_3 = the max of c_1 and c_2. Thus $f_1(n) + f_2(n) <= c_1g_1(n) + c_2g_2(n) <= c_3g_1(n) + c_3g_2(n)$. Well by definition, since the rightmost part of the inequality is larger than $f_1(n) + f_2(n)$, $f_1(n) + f_2(n)$ cannot be larger than that upper limit. Thus, the conjecture is true.

Question 5

Insertion sort code:

```
def insertSort(a):

for index in range(1, len(a)):
```

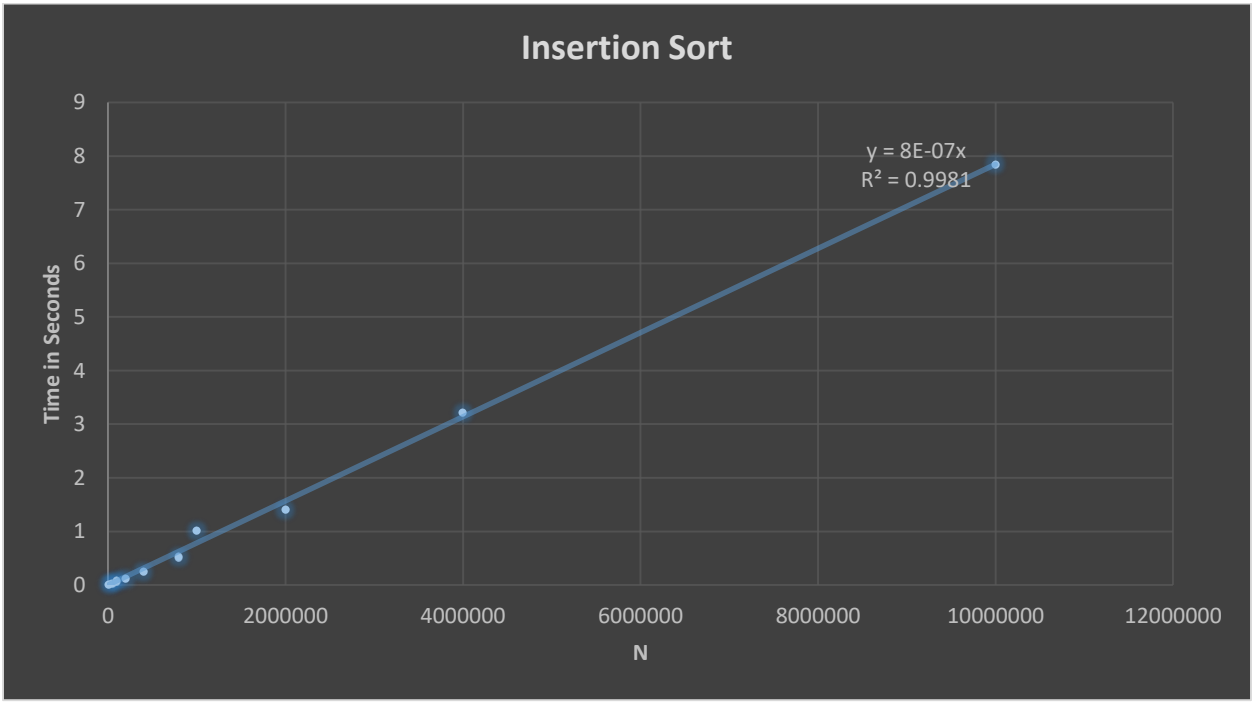
```
        cur_val = a[index]
        i = index - 1
        while i >= 0:
            if cur_val < a[i]:
                a[i + 1] = a[i]
                a[i] = cur_val
                i -= 1
            else:
                break
```

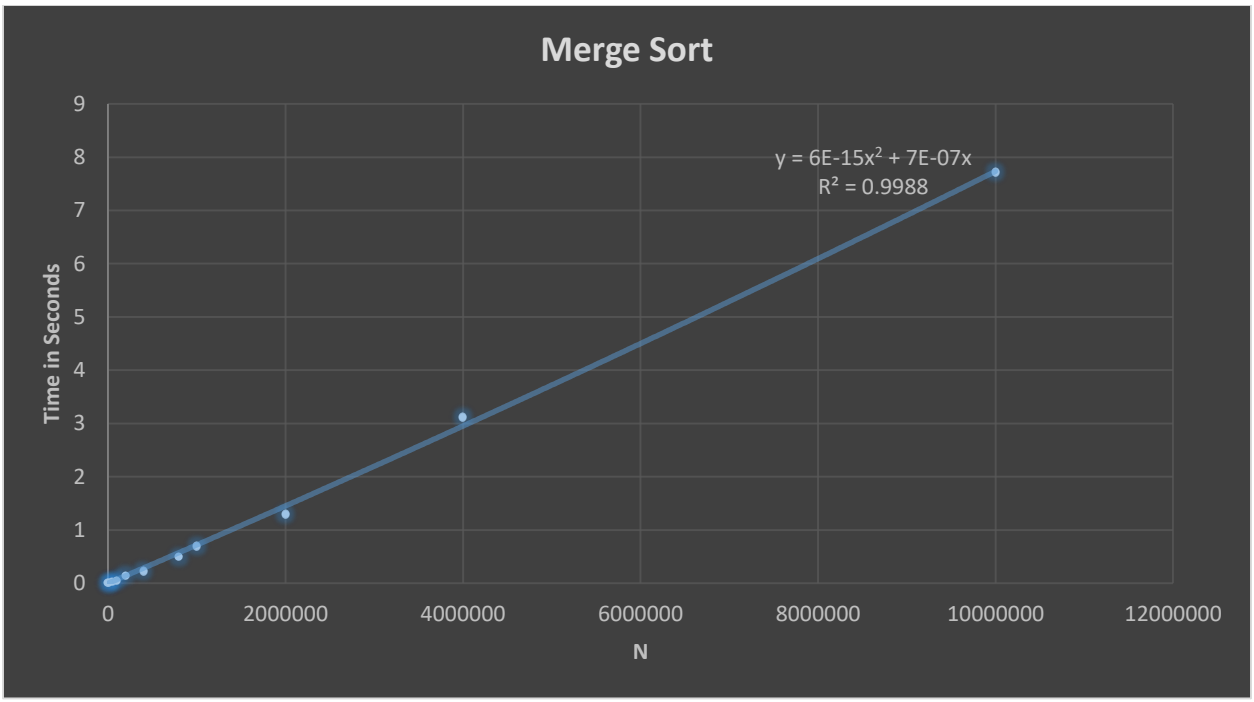
Merge Sort code:

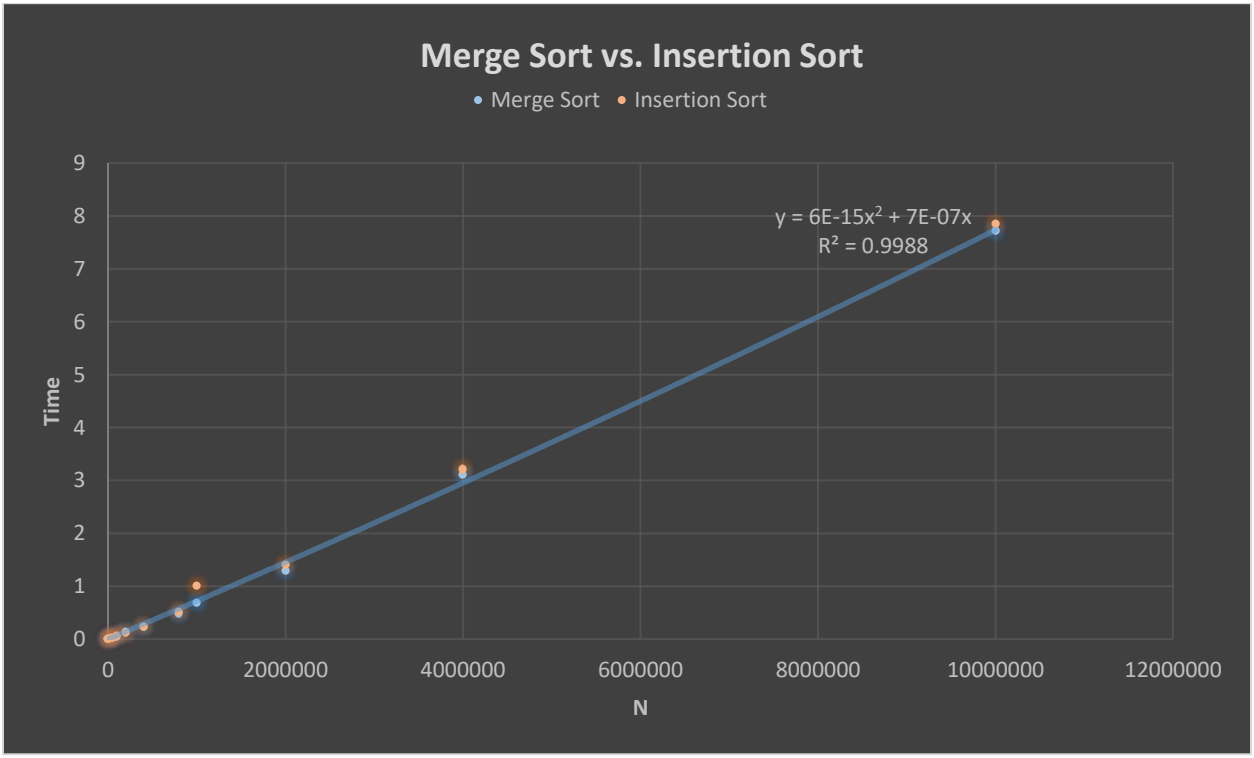
```
def mergeSort(a):
    if len(a) > 1:
        mid = len(a) // 2
        left = a[:mid]
        right = a[mid:]

        mergeSort(left)
        mergeSort(right)
```

```
i=0

j=0

k=0


while i < len(left) and j < len(right):
    if left[i] < right[j]:
        a[k]=left[i]
        i=i+1
    else:
        a[k]=right[j]
        j=j+1
    k=k+1


while i < len(left):
    a[k]=left[i]
    i=i+1
    k=k+1
```

```
while j < len(right):
    a[k]=right[j]
    j=j+1
    k=k+1
```







The merge sort graph represents data the best. The R^2 value is higher. A polynomial curve best fits merge sort and a linear curve best fits insertion sort. This is clear by their regression

equations. The experimental runtimes were similar to what I would expect from the theoretical runtimes. In most cases, doubling the input size of n coincided with double the runtime of the program. For larger input sizes, merge sort performed better than insertion sort, which was expected.

Extra Credit

The best and worst case scenarios are the same for both insertion sort and merge sort, as the goal of both is to sort lists of items. So I used this site: https://www.random.org/integer-sets/ to generate my lists for the best case and worst case (I just changed the algorithms to sort in descending order for the worst case) , which is a completely sorted list. This made both algorithms run much faster as the input size increased. There wasn't much of a difference for smaller inputs but the max allowed, 10000 integers, created a difference of about .2 second for merge sort and about 5(!) second for insertion sort. Obviously the worst case scenarios made the algorithms run much slower. The time to execute almost doubled as n increased.