CS 325 Spring 17
HW 1 – 25 points

Victor Ness – 4/7/2017

1) (1 pt) Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in 64nlgn steps. For what values of n does insertion sort run faster than merge sort?

   -Insertion sort runs faster for inputs size $2 \leq n \leq 43$, then merge sort is the faster algorithm

Note: lg n is log "base 2" of n or $\log_2 n$. There is a review of logarithm definitions on page 56. For most calculators you would use the change of base theorem to numerically calculate lgn.

2) (5 pts) For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) = Θ(g(n)). Determine which relationship is correct and explain.

   a. $f(n) = n^{0.25}$; $g(n) = n^{0.5}$

      -f(n) is O(g(n)) because it grows much slower as n increases in size and g(n) will act as an upper bound

   b. $f(n) = n$; $g(n) = \log^2 n$

      -f(n) is Ω(g(n)) since it grows much faster as n increases in size and g(n) will be a lower bound

   c. $f(n) = \log n$; $g(n) = \ln n$

      -f(n) is Θ(g(n)) since all logs will grow at roughly the same rate and could be an upper or lower bound if multiplied by a constant

   d. $f(n) = 1000n^2$; $g(n) = 0.0002n^2 – 1000n$

      -f(n) is Θ(g(n)) since they are both degree 2 and will grow at the same rate, g(n) can act as an upper or lower bound if multiplied by a constant

   e. $f(n) = n\log n$; $g(n) = n\sqrt{n}$

      -f(n) is O(g(n)) since the limit function simplifies to logn/sqrtn which will approach 0 as n approaches infinity

f. f(n) = e^n; g(n) = 3^n

-f(n) is O(g(n)) since $(e/3)^n$ will approach 0 as n moves towards infinity

g. f(n) = 2^n; g(n) = 2^{n+1}

-f(n) is Θ(g(n)) since the +1 in the exponent of g(n) is negligible and they will grow at the same rate, g(n) can act as an upper or lower bound if multiplied by a constant

h. f(n) = 2^n; g(n) = 2^{2n}

-f(n) is O(g(n)) since using the limit method (1/2^n) will approach 0 as n moves towards infinity

i. f(n) = 2^n; g(n) = n!

-f(n) is O(g(n)) since 2^n will grow slower than n! as n increases in size, g(n) will act as an upper bound to f(n)

j. f(n) = lgn; g(n) = $\sqrt{n}$

-f(n) is O(g(n)) since logs will grow slower than sqrt as n increases in size, g(n) will act as an upper bound to f(n)

3) (5 pts)

a. Describe in words and give pseudocode for an efficient algorithm that determines the maximum and minimum values in a list of n numbers.

-My initial instinct and the naïve solution would be to loop over the array while keeping tracking of high and low variables. However, it was clear that this method would not be under 1.5n comparisons for all inputs.

Instead of comparing each array element to the current high/low, we can compare 2 array elements at a time to each other and form 2 new lists with highs and lows. The high list will be guaranteed to have the highest number, while the low list will be guaranteed to have the lowest number. Then, we can again compare elements against each other for each of the high and low lists to find the highest/lowest number in the whole set.

See next page

Pseudocode

```
Int input[];
Int highNums[];
Int lowNums[];

Int high = input[0];
Int low = input[0];
//separate input list into highs and lows
For (int I = 0; I < array.size; i+=2)
      If I == input.size-1 //account for odd numbered input
            If input[i] > low
                  highNums.add(input[i]
      num1 = input[i];  //caching to reduce array accesses
      num2 = input[i+1];

      if num1 < num2
            lowNums.add(num1)
            highNums.add(num2)
      else
            lowNums.add(num2)
            highNums.add(num1)

for (I = 0; I < array.size/2; I+=2)
      if highNums[i] > highNums[i+1]
            high = highNums[i]
      if lowNums[i] < lowNums[i+1]
            low = lowNums[i]
```

b. Show that your algorithm performs at most 1.5n comparisons.

Using the above algorithm, splitting the input array into high and low arrays will take n/2 comparisons. Finding the high number in the highNums array will be another n/2 comparisons, and finding the low number in the lowNums array will be n/2 comparisons. Therefore the total comparisons made will be 1.5n.

c. Demonstrate the execution of the algorithm with the input L = [9, 3, 5, 10, 1, 7, 12].

L = [9, 3, 5, 10, 1, 7, 12]
highNums=[], lowNums=[]
high = 9, low = 9
… compare and split arrays …
highNums= [9, 10, 7, 12], lowNums = [3, 5, 1]
high = 10, low = 3
high = 12, low = 1

4) (4 pts) Let f_1 and f_2 be asymptotically positive non-decreasing functions. Prove or disprove each of the following conjectures. To disprove give a counter example.

a. If $f_1(n) = O(g(n))$ and $f_2(n) = O(g(n))$ then $f_1(n) = \Theta(f_2(n))$.
-Suppose $f_1 = n$, $f_2 = n^2$, and $g(n) = n^2$. It would then be true that $f1(n) = O(g(n))$ and $f2(n) = O(g(n))$. However, n cannot be $\Theta(n^2)$ so the conjecture is false.

b. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$

$By\ definition$:

$0 \leq f_1(n) \leq c_1 g_1(n) for\ all\ n \geq n_0$

$0 \leq f_2(n) \leq c_2 g_2(n) for\ all\ n \geq n_1$

$0 \leq f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) for\ all\ n \geq \max(n_0, n_1)$

$0 \leq f_1(n) + f_2(n) \leq c_1(\max(g_1(n), g_2(n))) + c_2(\max(g_1(n), g_2(n)))$

$0 \leq f_1(n) + f_2(n) \leq (c_1 + c_2)(\max(g_1(n), g_2(n)))$

Therefore by definition, $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$

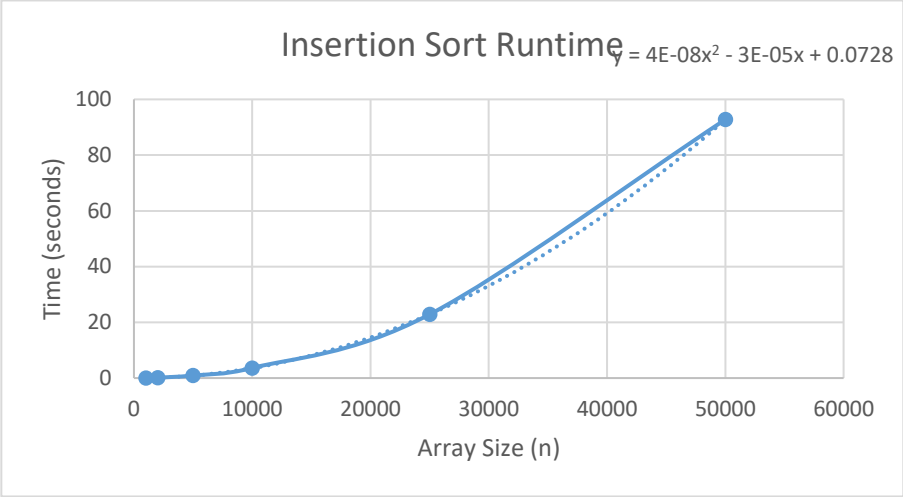
5) (10 pts) **Merge Sort vs Insertion Sort**

The goal of this problem is to compare the experimental running times of two sorting algorithms. These are very common algorithms and you may modify existing code if you reference it.

a) Implement merge sort and insertion sort to sort an array/vector of integers. To test the algorithms you will need to generate arrays of random integers. You may implement the algorithms in the language of your choice. Provide a copy of your code with your HW. Since we will not be executing the code for this assignment you are not required to use the flip server.

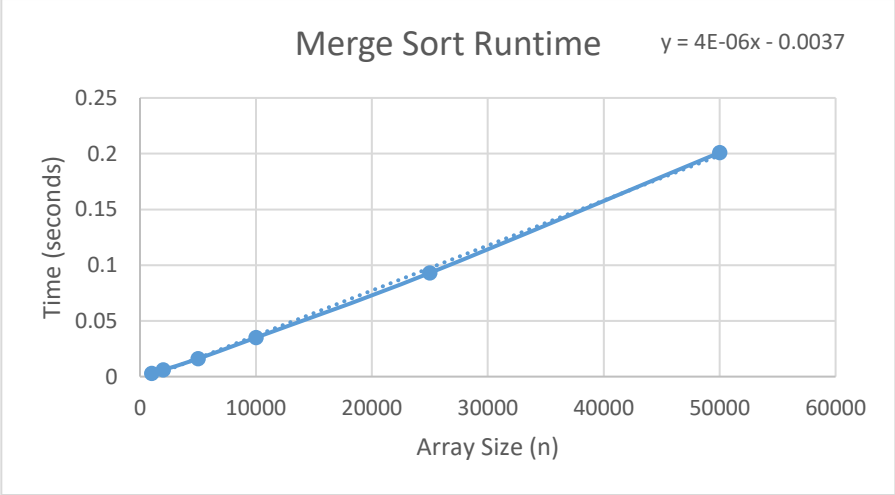
b) Use the system clock to record the running times of each algorithm for n = 1000, 2000, 5000, 10,000, …. You may need to modify the values of n if an algorithm runs too fast or too slow to collect the running time data. If you program in C your algorithm will run faster than if you use python. You will need at least seven values of t (time) greater than 0. If there is variability in the times between runs of the same algorithm you may want to take the average time of several runs for each value of n.

c) For each algorithm plot the running time data you collected on an individual graph with n on the x-axis and time on the y-axis. You may use Excel, Matlab, R or any other software. Also plot the data from both algorithms together on a combined graph. Which graphs represent the data best?



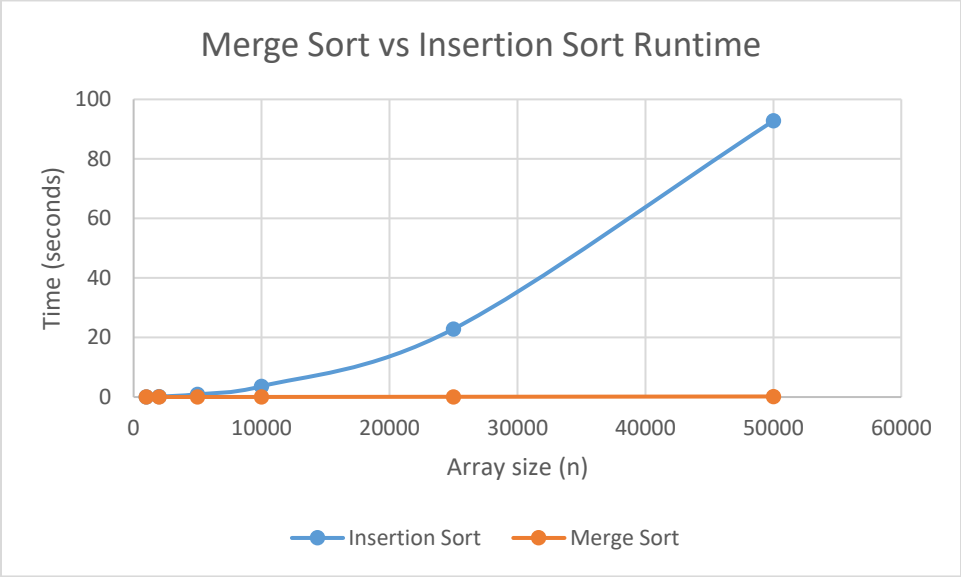
Insertion Sort

Array Size (n)	1000	2000	5000	10000	25000	50000
Time (sec)	0.034	0.143	0.938	3.619	22.858	92.77



Merge Sort

Array Size (n)	1000	2000	5000	10000	25000	50000
Time (sec)	0.003	0.006	0.016	0.035	0.093	0.201



Array Size (n)	1000	2000	5000	10000	25000	50000
Insertion (sec)	0.034	0.143	0.938	3.619	22.858	92.77
Merge (sec)	0.003	0.006	0.016	0.035	0.093	0.201

d) What type of curve best fits each data set? Again you can use Excel, Matlab, any software or a graphing calculator to calculate a regression equation. Give the equation of the curve that best "fits" the data and draw that curve on the graphs of created in part c).

-That data created with insertion sort best fit a quadratic curve, while the data created by merge sort best fits a linear trendline. (trendline equations are on the graphs for insertion and merge sort)

How do your experimental running times compare to the theoretical running times of the algorithms? Remember, the experimental running times were "average case" since the input arrays contained random integers.

-For insertion sort, there was some variation since the algorithm would not have to iterate through all array elements for every index. For example, when doubling the array elements from 1000 to 2000, you would expect a 4x increase in runtime, however only a 2x increase was seen. Other times, such as the doubling of 25000 to 50000, a result pretty close to the 4x increase was seen.

For merge sort, the algorithm is always going to increase linearly in runtime as n increases. Since the array is always going to be broken down all the way and all the comparisons will be made, the data is spot on for this expectation.

EXTRA CREDIT: *It was the best of times it was the worst of times…*

Generate best case and worst case input for both algorithms and repeat the analysis in parts b) to d) above. Discuss your results.

Best Case: Already Sorted

Best Case	1000	2000	5000	10000	25000	50000
Merge Sort	0.0027	0.0056	0.0146	0.0311	0.084	0.1821
Insertion Sort	0.0001	0.0002	0.0006	0.0012	0.003	0.0059

Worst Case: Reverse Sorted

Worst Case	1000	2000	5000	10000	25000	50000
Merge Sort	0.0022	0.0049	0.0132	0.0286	0.0912	0.163
Insertion Sort	0.0643	0.2816	1.8336	7.3747	46.6758	188.2563

-We can see from these results that merge sort will upload its O(nlogn) complexity regardless of what ordering our input is in. This is because every merge sort run will go through the process of breaking down the input all the way down into size 1 arrays, then do the same number of comparisons no matter what. This means there is very little variability in time when you run a merge sort on an input array.

This is contrasted by the insertion sort. Insertion sort will run in linear time if the array is already sorted since it does not need to scan through all previous elements in order to find the correct space. However, its $O(n^2)$ worst case complexity is extremely evident when you have a reverse sorted array. In this case, it will need to scan down through the entire array to find the correct space for every single element in the array. The time ramps up extremely quickly, roughly increasing by 4x for each doubling of array size.

Code Used

```
# CS325 Homework 1
# Victor Ness
# 4/7/2018
import random
import time


def insertion_sort(array):
    """insertion_sort accepts a list of integers and sorts them in ascending order"""

    if len(array) == 0 or len(array) == 1:
        return array

    for i in range(1, len(array)):
        current = array[i]
        pos = i

        while pos > 0 and current < array[pos-1]:
            array[pos] = array[pos-1]
            pos -= 1

        array[pos] = current


def merge_sort(array):
    """merge_sort accepts a list of integers and sorts them in ascending order"""

    if len(array) > 1:
        middle = len(array) // 2
        left = array[:middle]
        right = array[middle:]

        merge_sort(left)
        merge_sort(right)

        i = 0  # left index
        j = 0  # right index
        k = 0  # new array index

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

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

        while j < len(right):
```

```
            array[k] = right[j]
            j += 1
            k += 1

#testing
n = 50000
randoms =[random.randrange(0,100,1) for _ in range(n)]
start = time.clock()
merge_sort(randoms)
end = time.clock()
time = end-start
print("For %d terms, running time is: #d seconds", n, time)
```