CS 325 - HW 1 Solutions Examples

1) 3 points total- There are several different correct algorithms

3 points = Algorithm logically correct with $\Theta(n \lg n)$ running time explained, example and pseudocode.

2 points = Algorithm correct but missing the explanation of the running time

1 points = Algorithm is not $\Theta(n \lg n)$-or is not logically correct but an example is given.

Describe a $\Theta(n \lg n)$- time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

Method 1:

The following algorithm solves the problem:

1. Sort the elements in S.
2. Form the set $S' = \{z : z = x - y \text{ for some } y \in S\}$.
3. Sort the elements in S'.
4. Merge the two sorted sets S and S'.
5. There exist two elements in S whose sum is exactly x if and only if the same value appears in consecutive positions in the merged output.

To justify the claim in step 4, first observe that if any value appears twice in the merged output, it must appear in consecutive positions. Thus, we can restate the condition in step 5 as there exist two elements in S whose sum is exactly x if and only if the same value appears twice in the merged output.

Suppose that some value w appears twice. Then w appeared once in S and once in S'. Because w appeared in S', there exists some $y \in S$ such that $w = x - y$, or $x = w + y$. Since $w \in S$, the elements w and y are in S and sum to x.

Conversely, suppose that there are values $w, y \in S$ such that $w + y = x$. Then, since $x - y = w$, the value w appears in S'. Thus, w is in both S and S', and so it will appear twice in the merged output.

Steps 1 and 3 require $\Theta(n \lg n)$ steps. Steps 2, 4, 5, and 6 require $O(n)$ steps. Thus the overall running time is $O(n \lg n)$.

Note: Must also check if x/2 is in S

Method 2:

1. Sort the input list using merge sort, O(n lg n).
2. For each element in the sorted list, calculate the compliment that would need to exist by subtracting x and search the sorted list. Because, in the worst case, this would result in searching the entire list, the loop would execute n times. Calculating the compliment would therefore cost O(n) and the binary search, which is normally O(lg n), when performed n times, would have a cost of O(n lg n).

pseudocode	times executed * cost -> *total cost*
findIfTwoSum(S, x):	
sorted = mergeSort(S)	1 * O(n lg n) -> O(n lg n)
for i = 0 to sorted.length	n * O(1) -> O(n)
compliment = x - sorted[i]	n * O(1) -> O(n)
if (binarySearch(sorted, compliment) != -1)	n * O(lg n) -> O(n lg n)
return True	1 * O(1) -> O(1)
return False	1 * O(1) -> O(1)

There are two steps in the algorithm with cost of O(n lg n), resulting in a total cost of 2 n lg n; however, constant factors are discounted in asymptotic analysis; therefore, the total cost of the algorithm is still O(n lg n).

Method 3:

For an O(n lg$n$), it would seem like a sorted array would be required. Since as part of this weeks learning we discussed Merge Sort, which is O(n lg$n$) - So at a minimum, the data can be sorted in that time. Then we have to find a way to find if the sum exists in NO WORSE than O(n lg$n$) time.

It turns out that the sum, from a sorted array, can be determined in O(n) time in the following manner:

a) Start by taking the sum of the first and last elements in the array, and comparing that value to the sum that we are looking for.
b) If the sum is either smaller than the smallest value, or larger than twice the largest value, then the sum cannot be found in the array, and no further effort is required.
c) If the sum of these is greater than the sum we are interested, then we go to the second largest value in the array from the higher value. If the sum is smaller than this sum, we move from the smallest number to next smallest number.
d) At each increment, the sum is compared, and based on whether the sum is larger or smaller, we take the appropriate action.
e) This continues until either the sum is matched or all available numbers are exhausted.
f) Since each number in the array is visited once, this can be done in O(n) time.

Combining these, we would have a sum of O(n) + O(n lg$n$) - Since as n gets large only the largest term dominates, the O(n) drops out and we have a worst case of O(n lg$n$).

CS 325 - HW 1 Solutions Examples

2. **3 points - 0.5 point deduction for each one missed.**

a. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^{0.25}}{n^{0.5}} = 0$, so **f(n) is O(g(n)).**

b. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log n^2}{\ln n} = \frac{2\log n}{\ln n} = 2$, so **f(n) is Θ (g(n)).**

c. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{nlogn}{n\sqrt{n}} = \frac{logn}{\sqrt{n}} = 0$, b/c g(n) grows at much faster rate. **f(n) is O(g(n)).**

d. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4^n}{3^n} = \lim_{n\to\infty} \left(\frac{4}{3}\right)^n = \infty$, so **f(n) is Ω(g(n)).**

e. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^n}{2^{n+1}} = \frac{1}{2}$ so **f(n) is Θ (g(n)).**

f. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{2^n}{n!} = 0$, so **f(n) is O(g(n)).**

3) a. **2 points: 0.5 for true (prove), 1.5 for correct proof.**

If $f_1(n) = O(g(n))$ and $f_2(n) = O(g(n))$ then $f_1(n) + f_2(n) = O(g(n))$.

Proof: By definition there exists a $c_1, c_2, n_1, n_2 > 0$ such that

$f_1(n) \le c_1 g(n))$ for $n \ge n_1$ and $f_2(n) \le c_2 g(n)$ for $n \ge n_2$. Since the functions are asymptotically positive

$$f_1(n) + f_2(n) \le c_1 g(n) + c_2 g(n)$$
$$\le (c_1 + c_2) g(n)$$

Let $k = (c_1 + c_2)$ and $n_0 = \max(n_1, n_2)$ then

$f_1(n) + f_2(n) \le k\, g(n)$ for $n \ge n_0$; $k, n_0 > 0$ and by definition

Therefore, $f_1(n) + f_2(n) = O(g(n))$

b. **2 points: 0.5 for false (disprove), 1.5 for any counter example.**

If $f(n) = O(g_1(n))$ and $f(n) = O(g_2(n))$ then $g_1(n) = \Theta(g_2(n))$.

Counterexample: Let $g_1(n) = n$, $g_2(n) = n^2$ *and* $f(n) = n$ but $g_1(n) \neq \Theta(g_2(n))$.

CS 325 - HW 1 Solutions Examples

4) **10 points total**

README file – **2 point**
Fully commented code Code **(2 points)**

Run code on TEACH with the file

data.txt containing the values below
10 10 9 8 7 6 5 4 3 2 1
3 1 1 1
5 9 8 2 3 3

merge.out and insert.out each should contain
1 2 3 4 5 6 7 8 9 10
1 1 1
2 3 3 8 9

3 points for the correct execution of insertion sort and 3 points for execution of merge sort.

5) **10 points total- Solutions may vary**
a) **2 points** - Insertion Sort and Merge Sort code with timing added (you do not have to run)

b) **2 points** for data at **least 5 values for each algorithm that are non-zero**

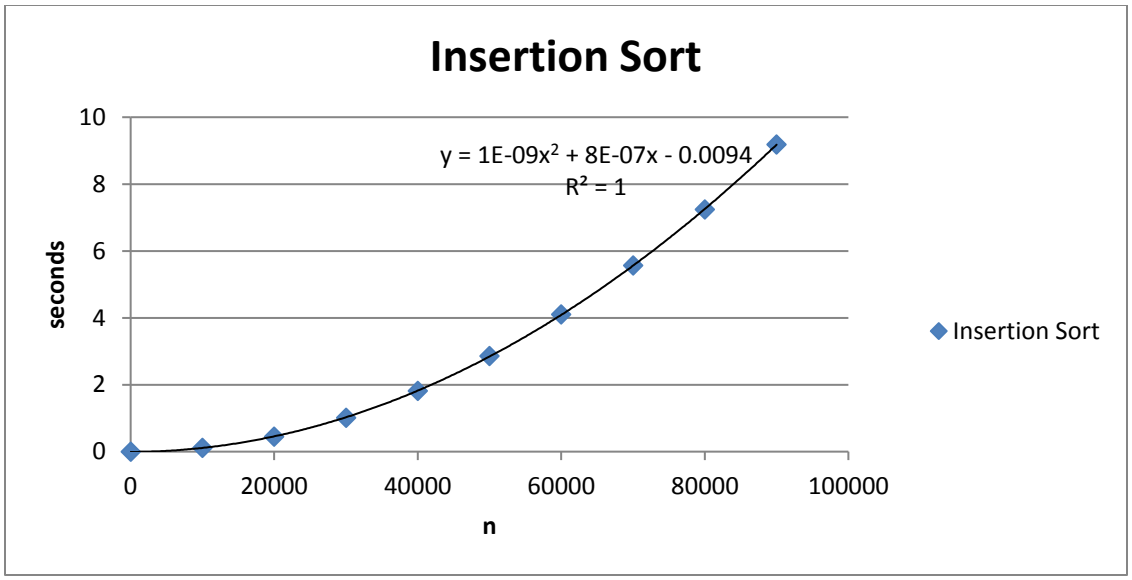
Insertion Sort

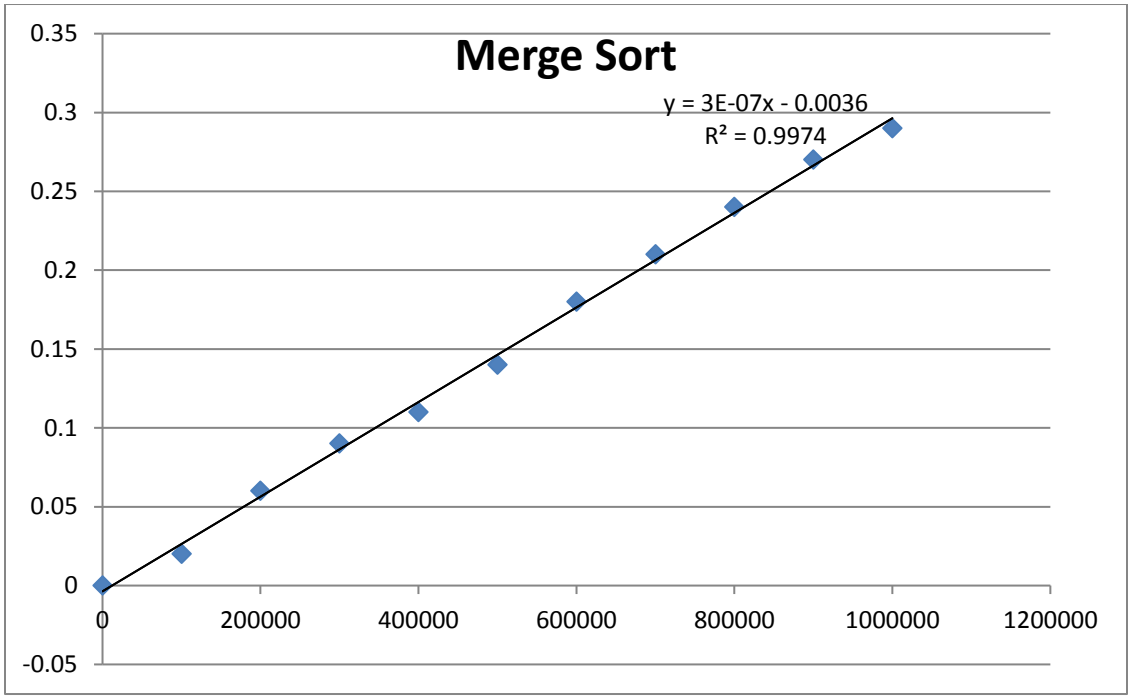
n	seconds
0	0
10000	0.11
20000	0.45
30000	1.01
40000	1.82
50000	2.86
60000	4.1
70000	5.57
80000	7.24
90000	9.18

Merge Sort

n	Seconds
0	0
100000	0.02
200000	0.06
300000	0.09
400000	0.11
500000	0.14
600000	0.18
700000	0.21
800000	0.24
900000	0.27
1000000	0.29

c) **2 points** - Plot the running time data you collected on graphs with n on the x-axis and time on the y-axis. **1 point for combined graph**





d) 2 points

Insertion sort has a quadratic fitted curve is displayed on the graph,

$y = 1E\text{-}09x^2 + 8E\text{-}07x - 0.0094$
$R^2 = 1$

This is an almost perfect fit. (results may vary)

Merge sort looks linear (or nlogn) fitted line is on the graph. Full credit for fitting an nlgn curve

y = 3E-07x - 0.0036
R^2 = 0.9974

This is also a very good fit.

e) 2 point

Insertion sort – average experimental running time is $\Theta(n^2)$ which matches the theoretical value

Merge sort – average experimental running time is $\Theta(n)$ which differs from the theoretical value of $\Theta(nlgn)$. Answers may vary.

Extra Credit

Must include verbal explanation for full credit

+2 if all analysis and graphs are done for both algorithms with best case data.

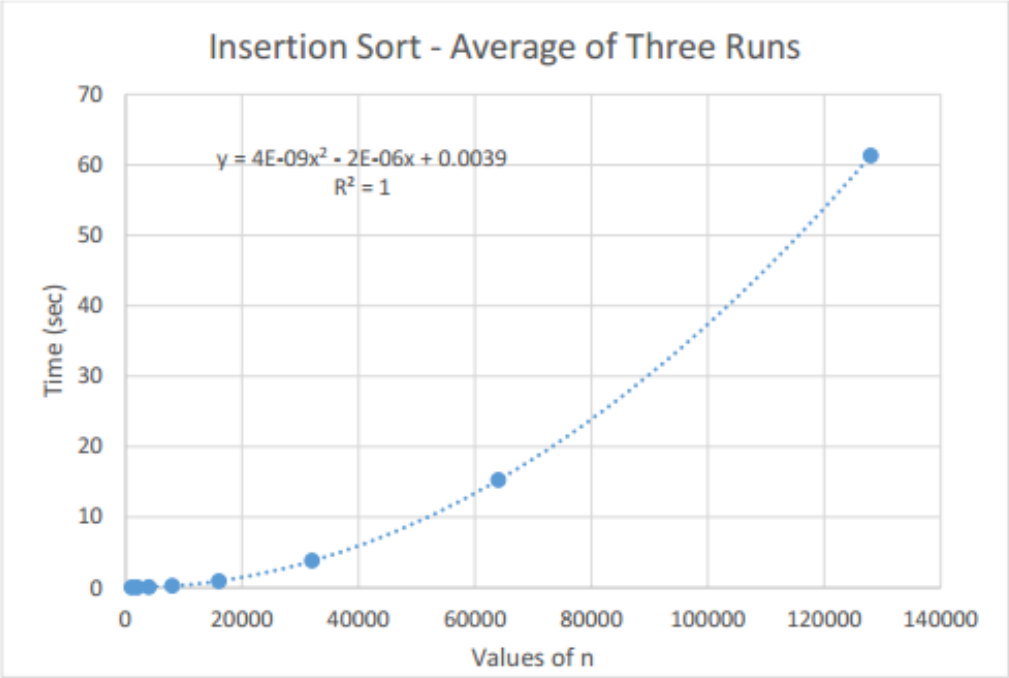
Best case the input array is already sorted. Insertion Sort will be linear and will run faster than Merge Sort which should have results similar to the average case.

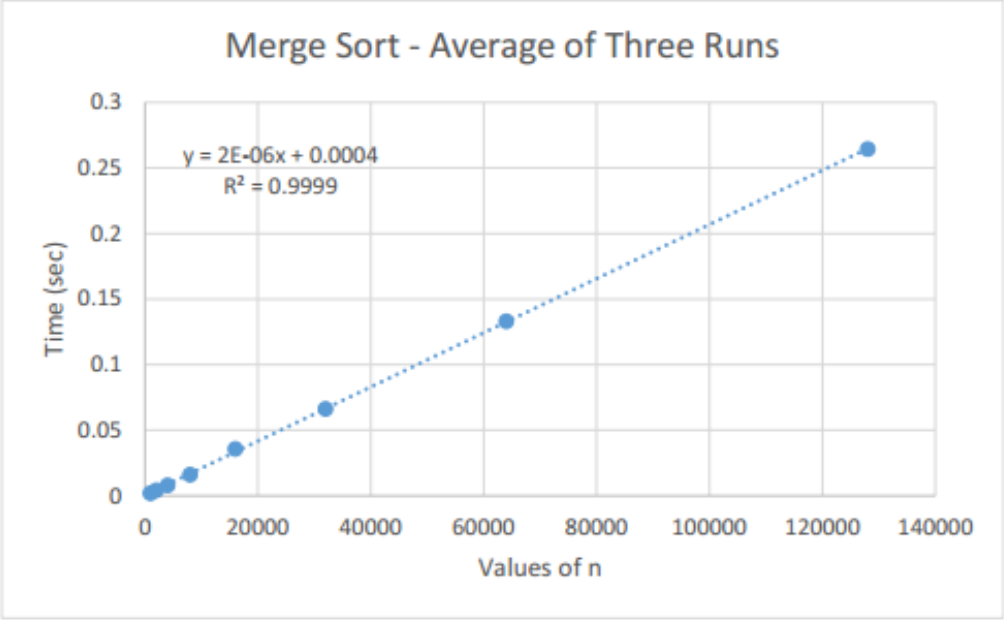
+2 if all analysis and graphs are done for both algorithms with worst case data.

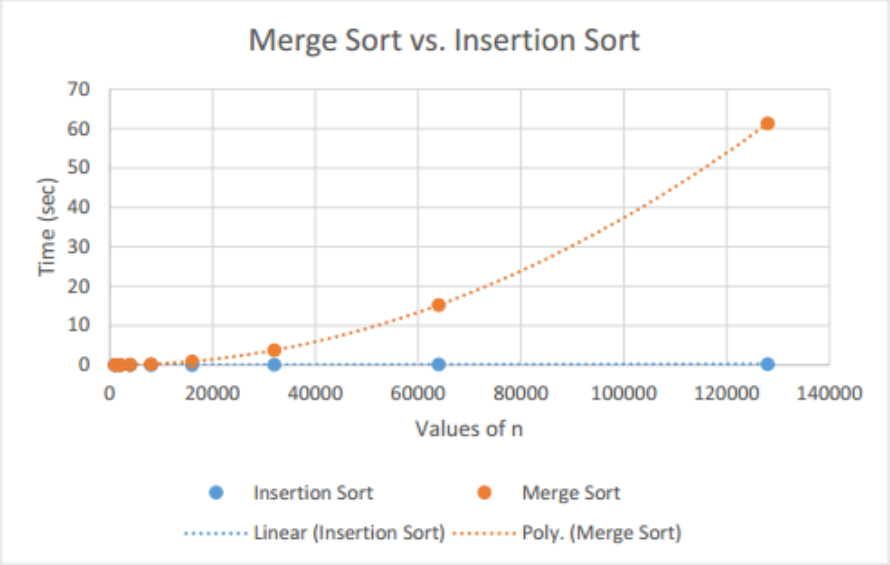
Worst case the input array in reverse order. Insertion Sort will be quadratic and will probably slower than in the average analysis. Merge Sort which should have results similar to the average case.

CS 325 - HW 1 Solutions Examples

Student example submissions





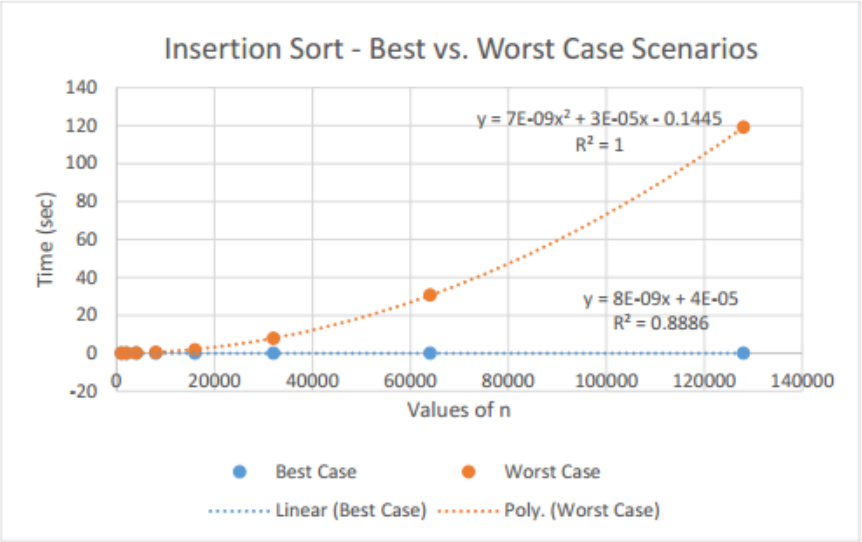


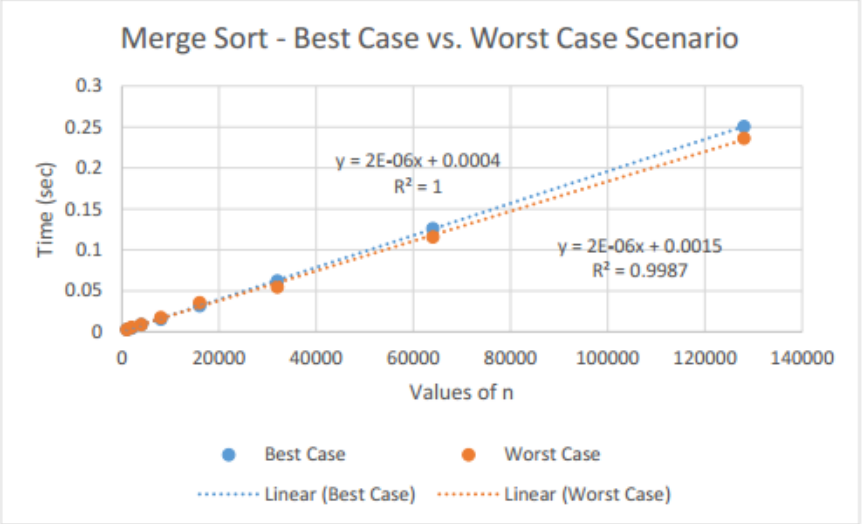
The graph that represents the data is best is the combination graph displaying the results of both sorting methods, as it clearly demonstrates the difference in running times as the values of n become large. This ratio is not as obvious when examining the individual graphs.

d. The type of curve that best fits each data set:
   - Both data sets adhere closely to their predicted curves.
   - Insertion sort behaves a 2nd degree polynomial with $y = 4e\text{-}09x^2 - 2e\text{-}06x + 0.0039$
   - The R^2 value of the insertion sort data is 1, indicating a perfect trendline, most likely a result of finding the average of three runs
   - The merge sort behaves like a linear function, as n dominates logn, with $y = 2e\text{-}06x + 0.0004$
   - The R^2 value of the merge sort data was also very close at 0.9999, again most likely as result of finding the average of three runs.

e. How do the experimental running times compare to the theoretical running times of the data?
   - The experimental running times of data are exactly on par with the theoretical running times for both methods. This is evidenced by the R^2 value of 1 for the insertion sort data and the almost perfect R^2 value of 0.9999 for the merge sort data. In both cases, the data behaved exactly as predicted with insertion sort adhering to $O(n^2)$ and merge sort adhering to O(nlogn). It is interesting to note that, although merge sort has a O(nlogn) it behaves more linear than nlogn on the graph. This is due to n being dominate over the values of logn, but those values of logn could be what is preventing the R^2 value from achieving that perfect value of 1.

CS 325 - HW 1 Solutions Examples





Running each algorithm under best case and worst case scenarios lead to some interesting results. In examining the results for Insertion Sort, there is a huge variance between best case (linear) and worst case scenario (2nd degree polynomial). The running times for the best scenario were essentially at zero, but the worst case scenario, while a 2nd degree polynomial like the average insertion sort scenario, took considerably more time than the average case. This is in stark contrast to the merge sort algorithm, which remained steady regardless of the scenario. This leads me to believe that some algorithms are more volatile than others, in that there are significant differences depending on the input scenario, while others like merge sort are more stable and return predictable results regardless of the input. This provides evidence of how important it is to consider the type of input your algorithms will be receiving. If you have an array that you know will be mostly sorted, insertion sort is the way to go; however, if that won't always be the case, or if there will be large values of n, an algorithm like merge sort that provides a more consistent running time would be the better option.