Josh Huff, huffj@oregonstate.edu
Professor Schutfort
CS 325 – Winter 2018
January 12, 2018

Homework 1

1. Describe a Θ(n lgn) 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. Explain why the running time is Θ(n lgn).

Assumptions: the set has been previously sorted using an algorithm of Θ(n lg n) or better, and we have access to a binary search function that returns a boolean value. Sorting the set separately will result in a runtime of 2 * (n lg n), but run-time complexity ignores multiplicative constants.

Python-like Pseudocode
```
for i in range(0, len(S) - 1):
    target = x – S[i]
    if target == S[i]: continue
    if binarySearch(target): return True
return False
```

This algorithm subtracts each set element from x to get a target value. If that value exists elsewhere in the set, then there are two elements in the set that sum to x. If the entire set is considered without finding such a target that evaluates to true when passed to the binary search, then it is not the case that two elements in the set sum to x.

By definition, a set will have no duplicate values, so if the target value equals the current element we're considering, that means there cannot be two elements to sum to x, so we can move on to the next element without running a binary search.

Iterating through the members of S (i.e. "for i in S:") is a linear-time operation. A binary search runs in log n time. Because the occurrence of binary searches is directly proportional to the number of elements in the set, the time complexity will equal Θ(n lg n).

2. 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) = O(g(n))

The limit of f(n)/g(n) as n approaches infinity is 0.

b. f(n) = Ω(g(n))

The limit of f(n)/g(n) as n approaches infinity is ∞.

c. f(n) = O(g(n))

The limit of f(n)/g(n) as n approaches infinity is 0.

d. f(n) = O(g(n))

The limit of f(n)/g(n) as n approaches infinity is 0.

e. f(n) = Θ(g(n))

The limit of f(n)/g(n) as n approaches infinity is a constant.

f. f(n) = O(g(n))

The limit of f(n)/g(n) as n approaches infinity is 0.

3. Let f1 and f2 be asymptotically positive non-decreasing functions. Prove or disprove each of the following conjectures. To disprove give a counter example.

a. If f1(n) = O(g(n)) and f2(n) = O(g(n)) then f1(n)+ f2(n) = O(g(n))

Assuming f1(n) = O(g(n)) and f2(n) = O(g(n)),
Then f1(n) = n and f2(n) = n,
Then, f1(n) + f2(n) = n + n = 2n
But, O(2n) = O(n)
Therefore, f1(n) + f1(n) = O(g(n)).

b. If f(n) = O(g1(n)) and f(n) = O(g2(n)) then g1(n) = Θ(g2(n))

Counter example:
O(g1(n)) = n^4 and O(g2(n)) = n!

Technically, f(n) is bounded above by both, but the g functions are not necessarily of the same order.

5a.

Insertion Sort – Timed

```
import random
import sys
from timeit import Timer

def insertsort(in_array, length):
    """Adapted from pseudocode found in the Wikipedia article for Insertion Sort"""
    # Comparison ranges from the second array item to the last item.
    for i in range(1, length):
        j = i
        while j > 0 and in_array[j - 1] > in_array[j]:
            # Swap operation
            in_array[j], in_array[j - 1] = in_array[j - 1], in_array[j]
            j -= 1
        i += 1

if __name__ == "__main__":

    # Get the array size as a command line argument
    n = int(sys.argv[1])

    # Populate a local array with n integers from the range of 0 to 10,000
    array = [0] * n
    for i in range(0, n):
        array[i] = random.randint(0, 10000)

    num_of_elements = len(array)

    # Using timeit module, as suggested by HW Group 18's Nathaniel Breneman
    # Call Timer, passing insertsort function and global variables,
    # running the sort ten times to get an average run-time for this n
    timer = Timer('insertsort(array, num_of_elements)', globals=globals())
    average = timer.timeit(10)

    # Exporting runtimes for later use
    # Writing in append mode
    with open("insert_runs.txt", "a") as f2:
        f2.write("Average run-time for " + str(n) + " inputs: " + str(average))
        f2.write("\n")
```

Merge Sort – Timed

```
#!/usr/bin/env python3
"""

	mergesort_timed.py

	---------------
	Josh Huff
	huffj@oregonstate.edu
	CS325 -- Winter 2018
	HW1 -- Problem 4 -- Merge Sort - Timed
	---------------

"""
import math # Math imported to take advantage of infinity constant
import random
import sys
from timeit import Timer

def merge(in_array, beginning, middle, end):
	"""Adapted from pseudocode found in CLRS"""

	# n1 = q - p + 1
	len_left_subarray = middle - beginning + 1

	# n2 = r - q
	len_right_subarray = end - middle

	# let L[1..n1 + 1] and R[1..n2 + 1]
	left_subarray = [0] * (len_left_subarray + 1)
	right_subarray = [0] * (len_right_subarray + 1)

	# for i = 1 to n1
	for i in range(0, len_left_subarray):
		left_subarray[i] = in_array[beginning + i]

	# for j = 1 to n2
	for j in range(0, len_right_subarray):
		right_subarray[j] = in_array[middle + j + 1]

	# L[n1 + 1] = infinity
	left_subarray[len_left_subarray] = math.inf

	# R[n2 + 1] = infinity
	right_subarray[len_right_subarray] = math.inf

	i = 0
	j = 0
	#for k = p to r
	for k in range(beginning, end + 1):
		if left_subarray[i] <= right_subarray[j]:
			in_array[k] = left_subarray[i]
			i += 1
		else:
			in_array[k] = right_subarray[j]
			j += 1

def mergesort(in_array, beginning, end):
	"""Adapted from pseudocode found in CLRS"""

	if beginning < end:
		middle = (beginning + end)//2
		mergesort(in_array, beginning, middle)
		mergesort(in_array, middle + 1, end)
		merge(in_array, beginning, middle, end)

if __name__ == "__main__":

	# Get the array size as a command line argument
	n = int(sys.argv[1])

	# Populate a local array with n integers from the range of 0 to 10,000
	array = [0] * n
	for i in range(0, n):
		array[i] = random.randint(0, 10000)

	# Using timeit module, as suggested by HW Group 18's Nathaniel Breneman
	# Calling Timer, passing mergesort function and global varaibles,
	# running the sort ten times to get an average run-time for this n
	timer = Timer('mergesort(array, 0, len(array) - 1)', globals=globals())
	average = timer.timeit(10)

	# Exporting runtimes for later use
	# Writing in append mode
	with open("merge_runs.txt", "a") as f2:
		f2.write("Average run-time for " + str(n) + " inputs: " + str(average))
		f2.write("\n")
```

5b.

Both algorithms were run with each *n* 10 times; the times below are averages of these trials.

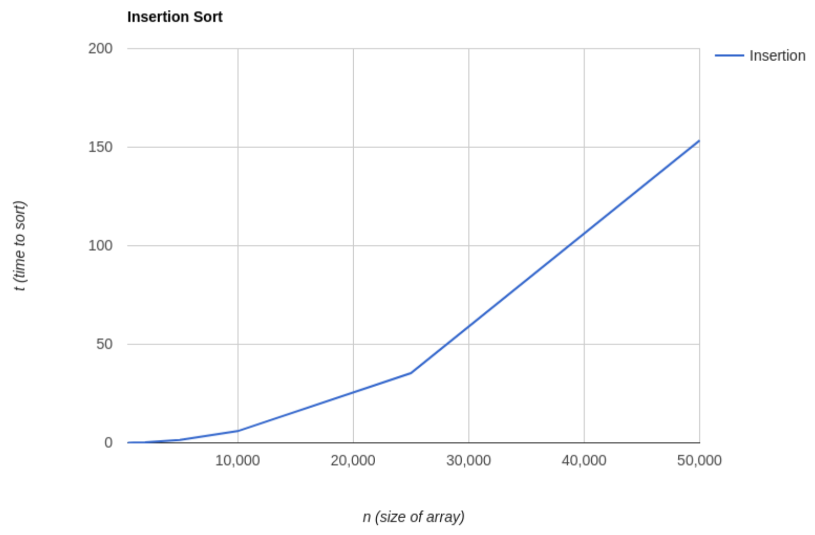
Insertion Sort

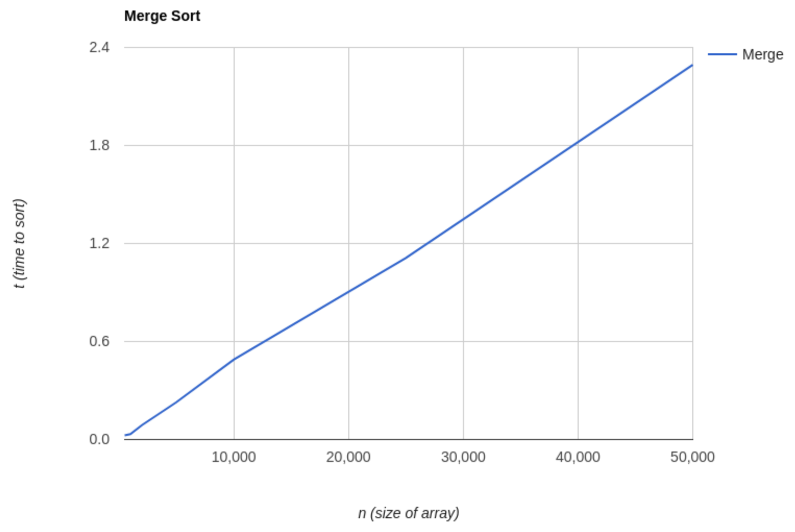
Average run-time for 500 inputs:	0.012684002984315157
Average run-time for 1000 inputs:	0.09592146199429408
Average run-time for 2000 inputs:	0.2261220820073504
Average run-time for 5000 inputs:	1.4549467710021418
Average run-time for 10000 inputs:	5.938299213012215
Average run-time for 25000 inputs:	35.37073920399416
Average run-time for 50000 inputs:	153.40190192501177

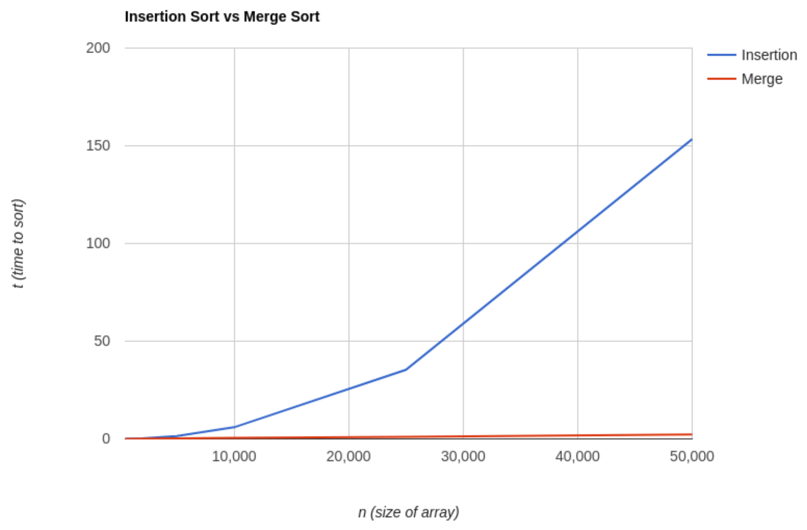
Merge Sort

Average run-time for 500 inputs:	0.026230368996039033
Average run-time for 1000 inputs:	0.03318440102157183
Average run-time for 2000 inputs:	0.08804758402402513
Average run-time for 5000 inputs:	0.22859938399051316
Average run-time for 10000 inputs:	0.489571302983677
Average run-time for 25000 inputs:	1.1121499389992096
Average run-time for 50000 inputs:	2.2948985380062368

5c.

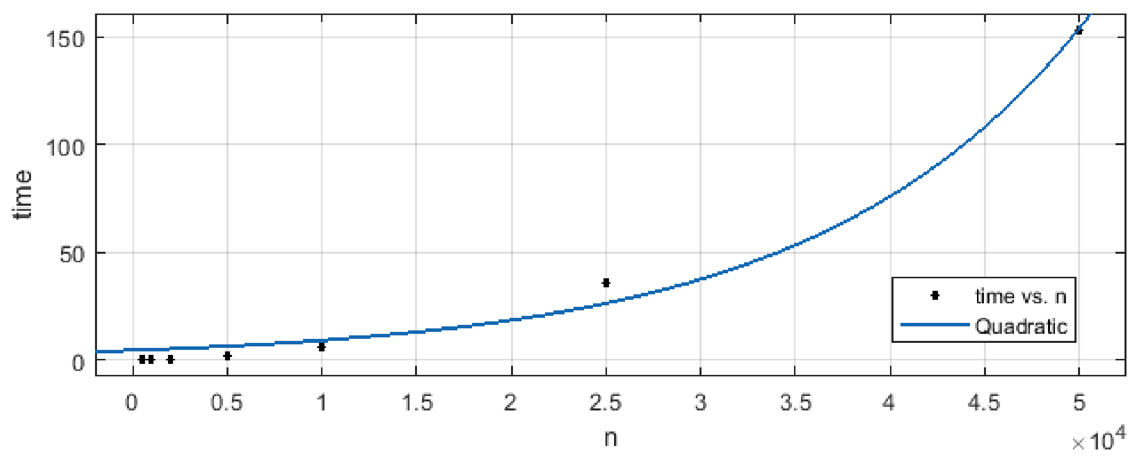

Insertion Sort
Insertion
t (time to sort)
200
150
100
50
0
10,000
20,000
30,000
40,000
50,000
n (size of array)


Merge Sort
Merge
t (time to sort)
2.4
1.8
1.2
0.6
0.0
10,000
20,000
30,000
40,000
50,000
n (size of array)

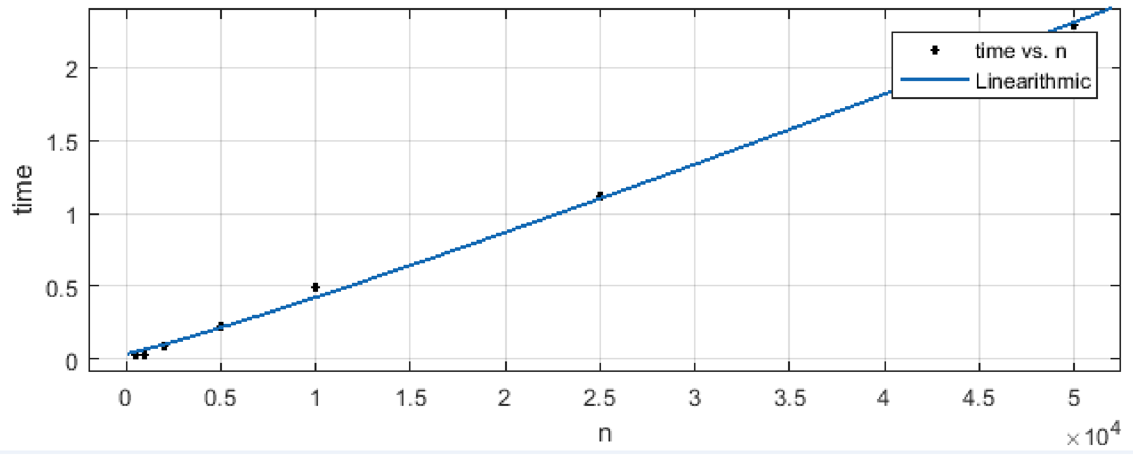

Insertion Sort vs Merge Sort
Insertion
Merge
t (time to sort)
200
150
100
50
0
10,000
20,000
30,000
40,000
50,000
n (size of array)

5d.

Insertion Sort Curve Fit – f(x) = a * exp(b * x) (confidence >99%)



Merge Sort Curve Fit – f(x) = a * x * log(x) + b (confidence >99%)



5e.

For low values of *n*, insertion sort experiences some variance from the expected growth, but as *n* grows larger, the values line up more faithfully with the theoretical. For example, the growth from a 5,000-length array to a 10,000-length array should represent a quadrupled running time, as the input has doubled. The run-time for insertion sort on an *n* of 5,000 was ~1.45, so the theoretical expectation is a run-time of ~5.8. In practice, the value was ~5.9.

Likewise, for low values of *n*, merge sort experiences some run-times that don't conform to expectations. However, after *n* is sufficiently large, the run-time stays in proportion to the input, with "spare change" coming from the (log n) factor.

Extra Credit

Insertion Sort Best Case Analysis:

Because the array is sorted in increasing order, the insertion sort has to do *n* traversals over the array and zero swaps. The result is a linear-time performance that, in fact, leaves merge sort in the dust after a surprisingly low breaking point. This makes it clear that, while insertion is inappropriate for most cases, it can be very useful (and quicker and easier to implement) than asymptotically better sorts if the data is suited for it.

Insertion Sort Worst Case Analysis:

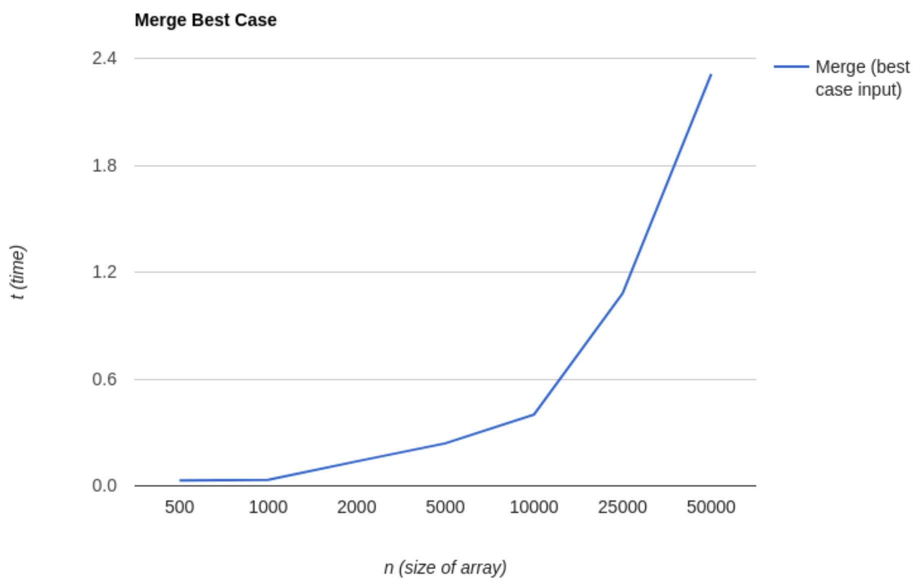
Because the array is sorted in decreasing order, the insertion sort has to do *n* traversals over the array and *n* swaps. The result is a guaranteed quadratic performance that does not differ much from the average case (though knowing it is performing at its worst somehow makes it feel longer). It's interesting to see how closely the trial plot conforms to the curve plot.

Merge Sort Best/Worst Case Analysis:

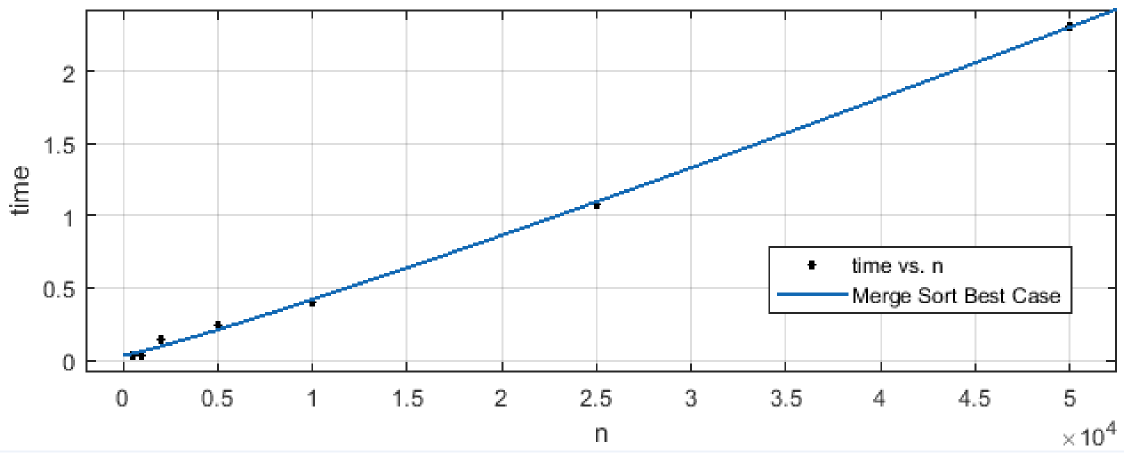
Despite the array being sorted in increasing order, the merge sort has to do the same number of recursive calls and make the same number of comparisons,. This explains why maximizing (or minimizing) the number of swaps makes no difference to the overall expected performance. The "steady" numbers make merge sort the obvious choice for larger inputs that are not necessarily sorted.

Merge Sort Best – (predictably and strikingly similar to average case)

Average run-time for 500 inputs:	0.029334275983273983
Average run-time for 1000 inputs:	0.03389139799401164
Average run-time for 2000 inputs:	0.13771004299633205
Average run-time for 5000 inputs:	0.23867426899960265
Average run-time for 10000 inputs:	0.40015805000439286
Average run-time for 25000 inputs:	1.0803264830028638
Average run-time for 50000 inputs:	2.311929385003168

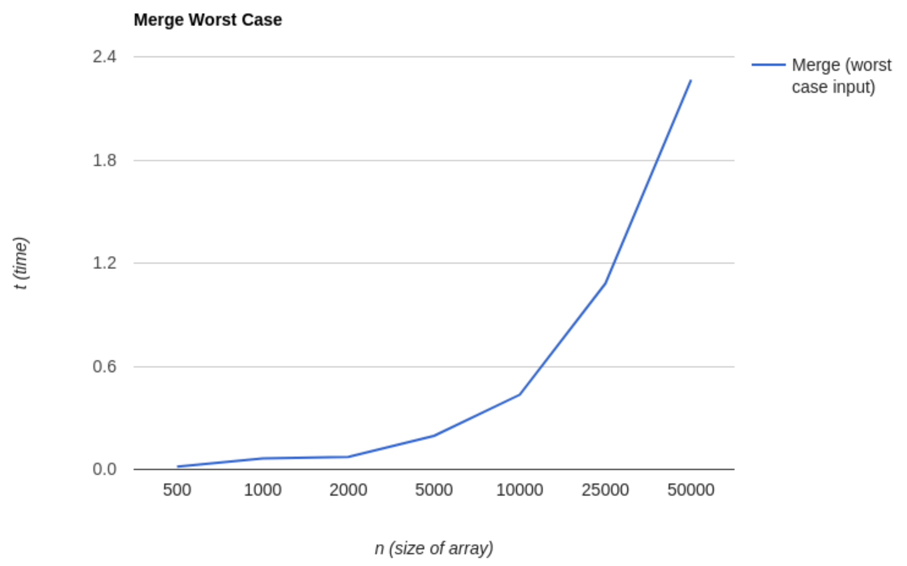


Merge Sort Best Curve – f(x) = a * x * log(x) + b (confidence >99%)

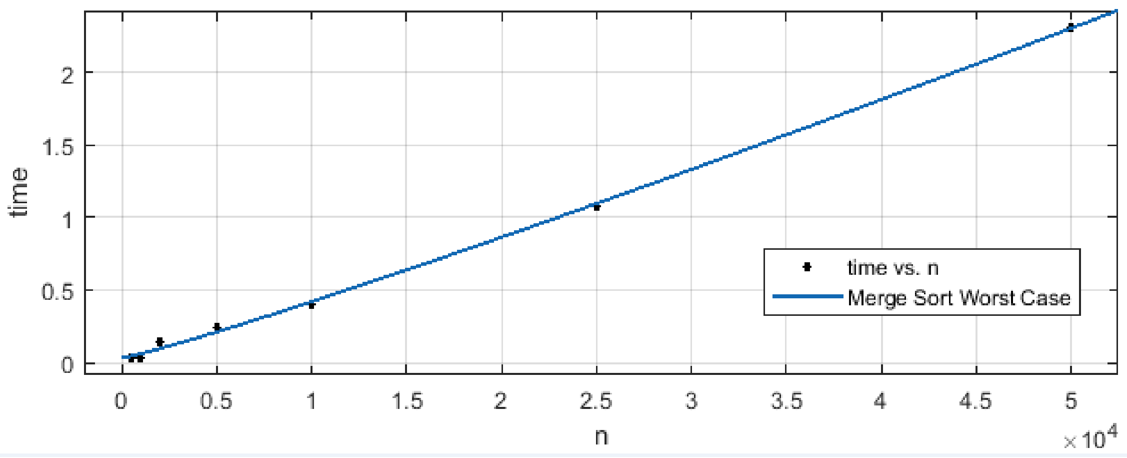


Merge Sort – Worst (predictably and strikingly similar to best and average cases)

Average run-time for 500 inputs:	0.015735205000964925
Average run-time for 1000 inputs:	0.06471202801913023
Average run-time for 2000 inputs:	0.07238150999182835
Average run-time for 5000 inputs:	0.19571801400161348
Average run-time for 10000 inputs:	0.43455194600392133
Average run-time for 25000 inputs:	1.081239443999948
Average run-time for 50000 inputs:	2.265992735978216

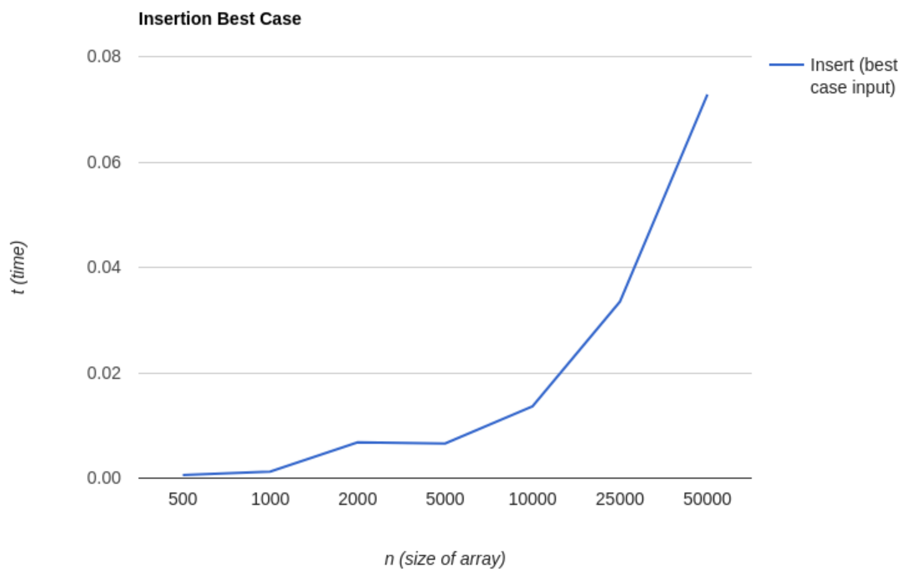


Merge Sort Best Curve – f(x) = a * x * log(x) + b (confidence >99%)

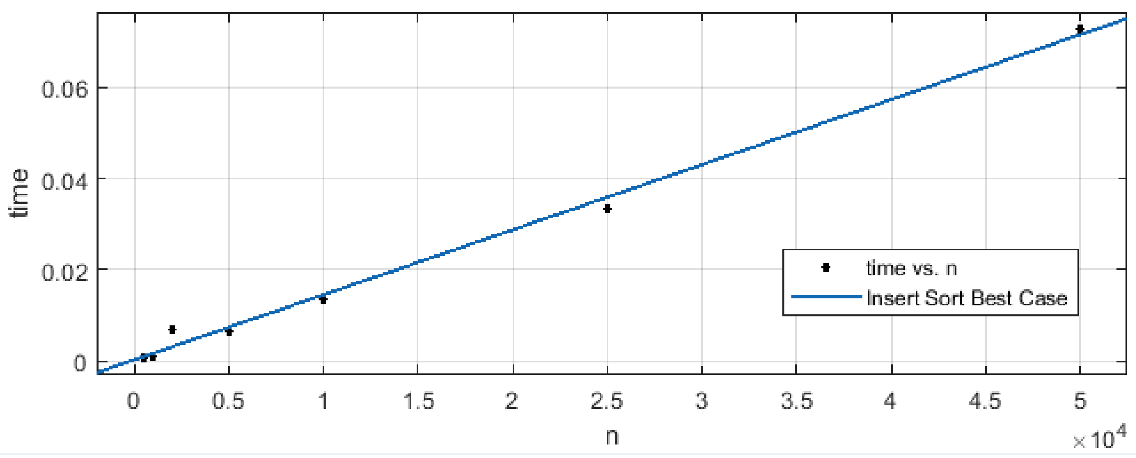


Insertion Sort – Best (ridiculously fast compared to average case)

Average run-time for 500 inputs:	0.00060149602235916078
Average run-time for 1000 inputs:	0.0012322590046096593
Average run-time for 2000 inputs:	0.006809302023611963
Average run-time for 5000 inputs:	0.0065800430020317435
Average run-time for 10000 inputs:	0.013607171014882624
Average run-time for 25000 inputs:	0.03351201399345882
Average run-time for 50000 inputs:	0.07280500698834658

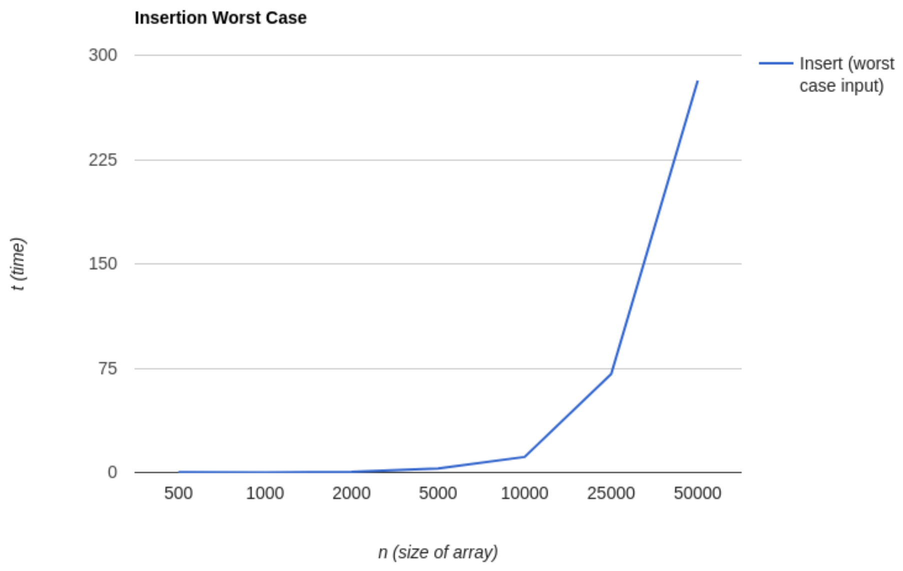


Insertion Sort Best Curve – f(x) = a * x + b (confidence >99%)

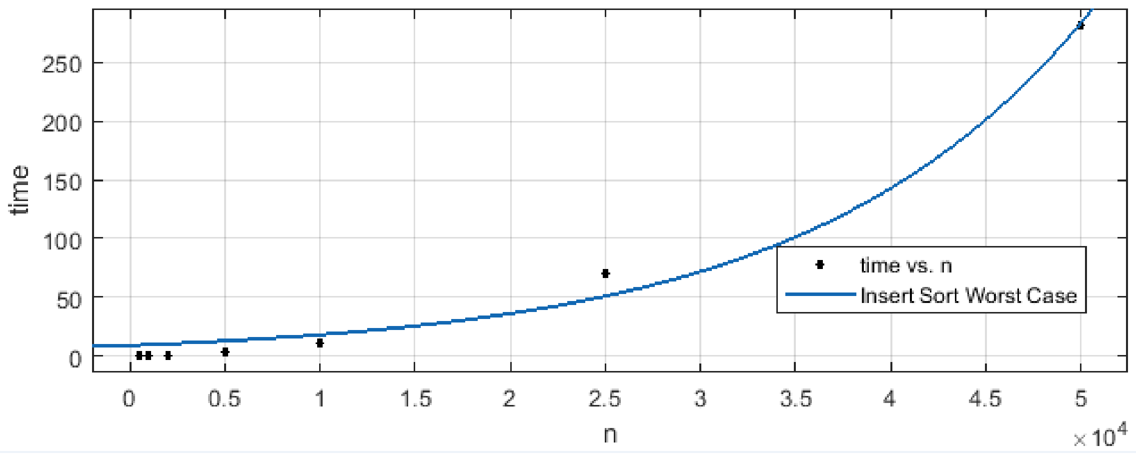


Insertion Sort – Worst (ridiculously slow compared to best and average cases)

Average run-time for 500 inputs:	0.1746587709931191
Average run-time for 1000 inputs:	0.10444826999446377
Average run-time for 2000 inputs:	0.43059751801774837
Average run-time for 5000 inputs:	2.8631126070104074
Average run-time for 10000 inputs:	11.178219530003844
Average run-time for 25000 inputs:	70.82835735101253
Average run-time for 50000 inputs:	281.85211014098604



Insertion Sort Worst Curve – f(x) = a * exp(b * x) (confidence >98%)


time
n
×10^4
time vs. n
Insert Sort Worst Case