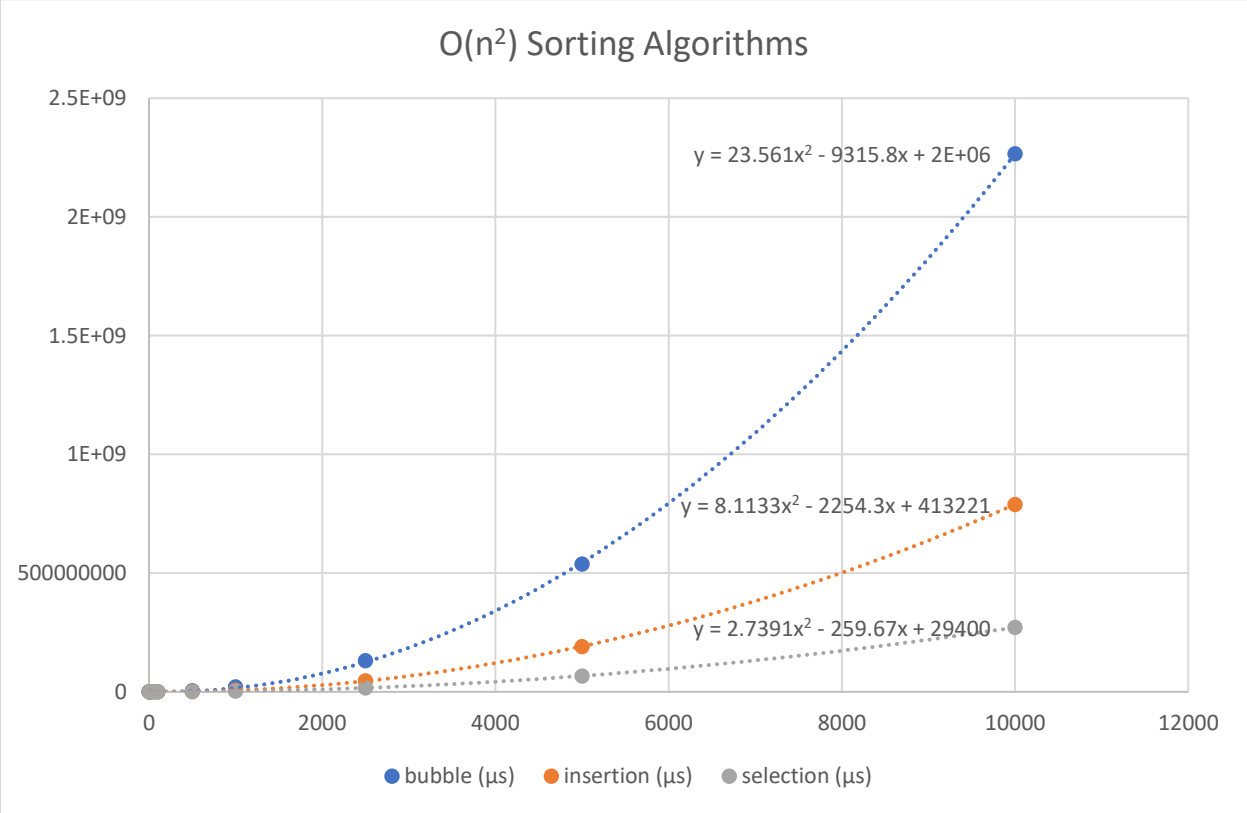
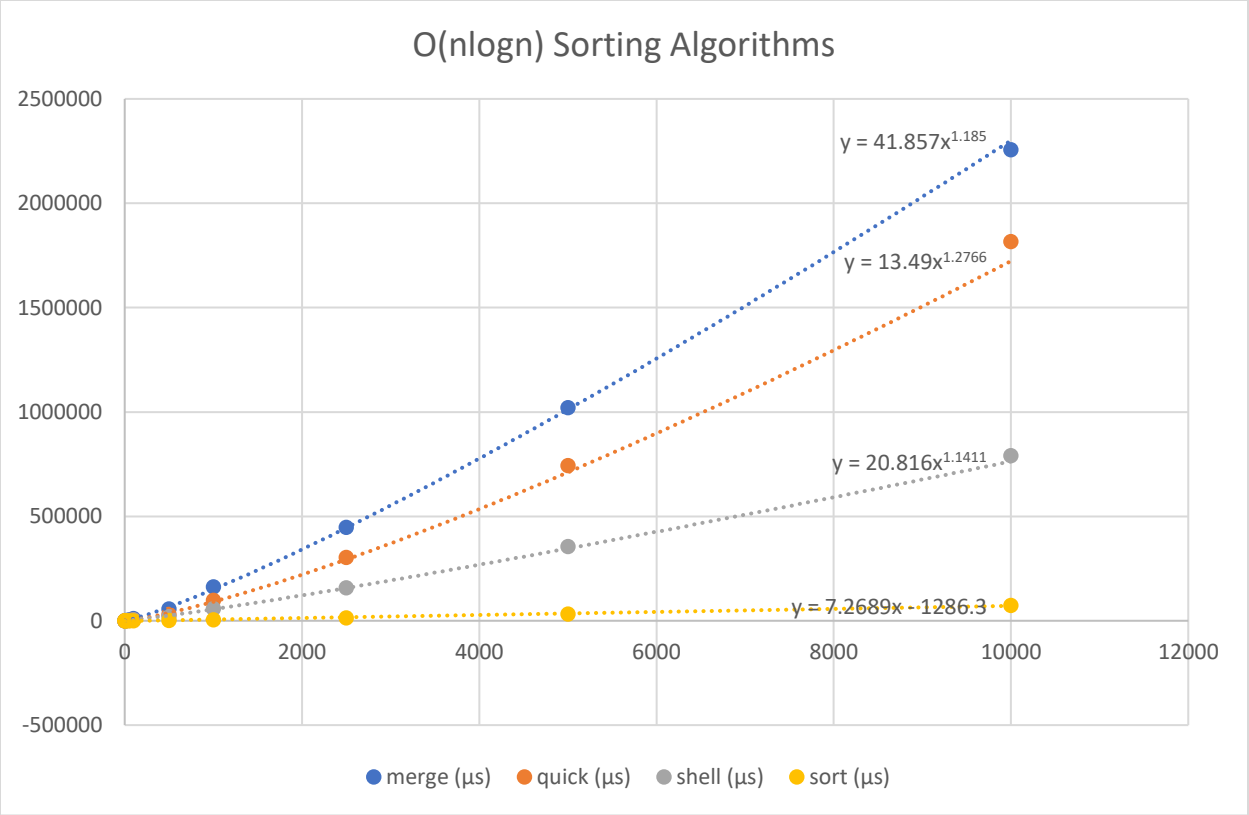
Lab 3

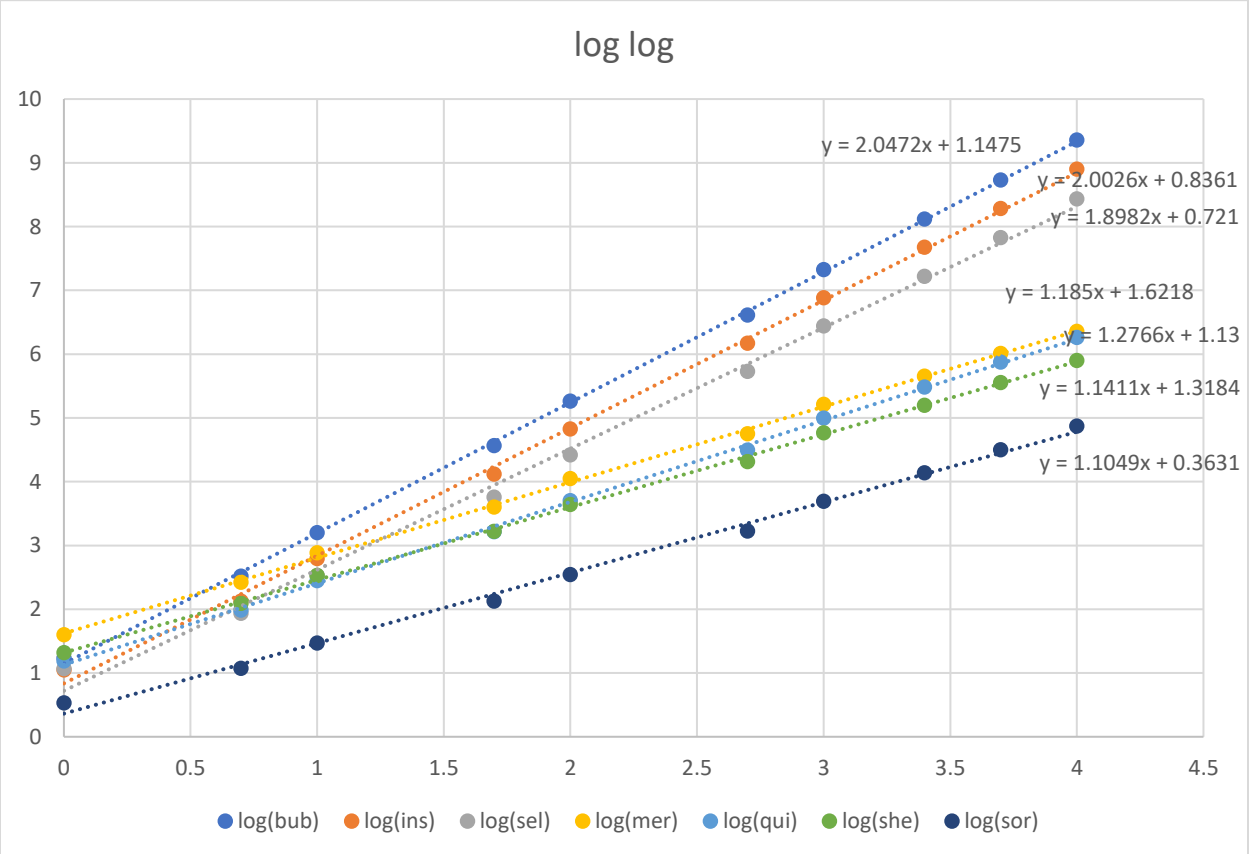
Drake Song

Data

n (tens)	bubble (µs)	insertion (µs)	selection (µs)	merge (µs)	quick (µs)	shell (µs)	sort (µs)
1	17.00613137	11.11635699	11.59006539	39.39674815	15.37973254	20.68526656	3.379119881
5	328.6273036	133.8068312	85.86754166	262.1975964	96.35228746	122.8799576	11.82691958
10	1577.638437	619.5947898	336.6487658	767.0128422	279.4563722	334.7539322	29.18043711
50	36472.94099	13093.78953	5664.541821	3999.298912	1638.352063	1648.789438	132.8278338
100	182731.9553	66810.31611	26092.04786	11143.26363	4994.370765	4378.660384	349.2967799
500	4080396.02	1474370.623	535816.5075	55853.09355	31203.37723	20431.72204	1677.985666
1000	21145017.41	7638297.697	2744864.401	162316.7972	98655.64716	58097.66604	4882.512423
2500	131389791.9	47034585.2	16438656.26	447732.1526	304007.952	157538.2322	13673.79808
5000	537798011.3	190307313.4	67177869.99	1020069.919	744010.3995	355198.1601	31660.53741
10000	2265807938	789506955.7	271347174.9	2255668.268	1816787.815	790560.6339	74031.95321




O(nlogn) Sorting Algorithms
2500000
2000000
1500000
1000000
500000
0
-500000
0
2000
4000
6000
8000
10000
12000
y = 41.857x^1.185
y = 13.49x^1.2766
y = 20.816x^1.1411
y = 7.2689x - 1286.3
merge (µs)
quick (µs)
shell (µs)
sort (µs)


log log
10
9
8
7
6
5
4
3
2
1
0
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
y = 2.0472x + 1.1475
y = 2.0026x + 0.8361
y = 1.8982x + 0.721
y = 1.185x + 1.6218
y = 1.2766x + 1.13
y = 1.1411x + 1.3184
y = 1.1049x + 0.3631
log(bub)
log(ins)
log(sel)
log(mer)
log(qui)
log(she)
log(sor)

Algorithm	Approximate O(n)	Actual O(n)
Bubble	n^2	n^2
Insertion	n^2	n^2
Selection	n^2	n^2
Merge	nlogn	nlogn
Quick	nlogn	nlogn
Shell	nlogn	≈ nlogn
Sort	nlogn	nlogn

Analysis

Out of the three algorithms that have an average runtime of $O(n^2)$, selection sort is the fastest (coefficient of 1.898 in log-log), then insertion sort (2.003), and bubble sort is the slowest (2.047). Bubble sort is known to perform horribly when n is large; however, there is one scenario when bubble sort is useful: to check if the list is sorted or not when n is small. Comparing the runtimes when n = 1 (10 numbers), it is clear to see that the runtime of bubble sort is on par with other algorithms; in fact, bubble sort is quicker than some of the O(nlogn) algorithms. However, as n increases, the runtime of bubble sort increases dramatically. Looking at insertion and selection sorts, the algorithm themselves are very similar in the way that after *i* iterations, the first *i* numbers are sorted in order. However, the difference between the two algorithms is that while insertion sort has to traverse through the entire list every iteration, selection sort only has to traverse the necessary amount of numbers in the list; meaning, selection sort goes through less numbers in general. Due to this fact, selection sort performs better than insertion sort.

The rest of the algorithms have average runtime of O(nlogn). Out of these algorithms, quick sort is the slowest (coefficient of 1.277 in log-log), then merge sort (1.185), then shell sort (1.141), and then Python's sort method (Timsort) with coefficient of 1.105 being the fastest. Quick sort has a problem where if the pivots are poorly chosen, the algorithm tends to perform badly. In addition, because there's no real efficient way of choosing the pivots within the algorithm for this lab, quick sort performs the worst.

Although merge sort has smaller coefficient, when just looking at the actual data and not the log-log, it is clear to see that merge sort actually takes the longest out of all of the nlogn sorting algorithms. This is due to the fact that for merge sort, all three runtimes (best, average, worst) are O(nlogn); in fact, merge sort continues to divide the list to the smallest units and then compare each unit and merge them together no matter how big the list is. This means that even though merge sort might be taking longer than quick sort, as *n* continues to grow, quick sort will eventually take longer than merge sort as the coefficient for quick sort in log-log plot is greater than that of merge sort.

For shell sort, it seems like choosing the gap as the half of each list seems to be working fine as both the runtime and then coefficient of the log-log plot seems to be smaller than quick sort and merge sort.

And then there's Python's sort function which utilizes Timsort. Timsort, according to the internet, is a hybrid stable sorting algorithm that is derived from merge sort and insertion sort. The algorithm goes through the list and finds subsequences that are already in order. If the list is already sorted or is almost sorted, the algorithm only needs to go through the list; hence the best-case runtime of O(n). The average runtime, however, is O(nlogn). Although this might be the case, from looking at the data and the coefficient of the log-log plot, it is clear to see that Timsort is definitely a lot faster than all of the other algorithms. When comparing the time it took for shell sort (the fastest after Timsort) with that of Timsort, Timsort ran about 9.3% faster than shell sort for n = 10000 (100,000 numbers in the list). This is due to the fact that unlike other sorting algorithms, Timsort utilizes subsequences that are already sorted to save time.

```
# CS 317 Algorithm Anaylsis Lab 3
# Drake Song
# Python 3.6
# All of the following sorting algorithms have been copied from CodeCodex with
# some adjustments to make sure the program runs smoothly as a whole.

import numpy as np
import timeit
import csv

def bubblesort(input_list):
    lst = input_list[:]
    swapped = True
    while swapped:
        swapped = False
        for i in range(len(lst)-1):
            if lst[i] < lst[i+1]:
                lst[i], lst[i+1] = lst[i+1], lst[i]
                swapped = True
    return lst

def insertsort(input_list):
    lst = input_list[:]
    for removed_index in range(1, len(lst)):
        removed_value = lst[removed_index]
        insert_index = removed_index
        while insert_index > 0 and lst[insert_index - 1] > removed_value:
            lst[insert_index] = lst[insert_index - 1]
            insert_index -= 1
        lst[insert_index] = removed_value
    return lst

def selectionsort(input_list):
    lst = input_list[:]
    l=lst[:]
    srt_lst=[]
    while len(l):
        lowest=l[0]
        for x in l:
            if x<lowest:
                lowest=x
        srt_lst.append(lowest)
        l.remove(lowest)
    return srt_lst

def mergesort(input_list):
    lst = input_list[:]
    if len(lst) == 1:
```

```
        return lst

    m = round(len(lst) / 2)
    l = mergesort(lst[:m])
    r = mergesort(lst[m:])

    if not len(l) or not len(r):
        return l or r

    result = []
    i = j = 0
    while (len(result) < len(r)+len(l)):
        if l[i] < r[j]:
            result.append(l[i])
            i += 1
        else:
            result.append(r[j])
            j += 1
        if i == len(l) or j == len(r):
            result.extend(l[i:] or r[j:])
            break

    return result

def quicksort(input_list):
    lst = input_list[:]
    if len(lst)<=1:
        return lst
    pivot=lst[0]
    less=    [x for x in lst if x<pivot]
    equal=   [x for x in lst if x==pivot]
    greater= [x for x in lst if x>pivot]
    lst = quicksort(less)+equal+quicksort(greater)
    return lst


def shellsort(input_list):
    lst = input_list[:]
    inc = int(round(len(lst) / 2))
    while inc:
        for i in range(len(lst)):
            j = i
            temp = lst[i]
            while j >= inc and lst[j-inc] > temp:
                lst[j] = lst[j-inc]
                j -= inc
            lst[j] = temp
        inc = int(inc/2) if inc/2 else (0 if inc==1 else 1)
```

```
    return lst



time_bub = []
time_ins = []
time_sel = []
time_mer = []
time_she = []
time_qui = []
time_sor = []
n = [1, 5, 10, 50, 100, 500, 1000, 2500, 5000, 10000]

with open('data.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    for j in n:
        print("starting n = {}".format(j))
        for i in range(25):
            x = np.random.randint(-1000000, 1000000, size=(j*10)).tolist()

            tic = timeit.default_timer()
            bubblesort(x)
            toc = timeit.default_timer()
            time_bub.append((toc-tic)*1000000)
            print("bubblesort is done")

            tic = timeit.default_timer()
            insertsort(x)
            toc = timeit.default_timer()
            time_ins.append((toc-tic)*1000000)
            print("insertsort is done")

            tic = timeit.default_timer()
            selectionsort(x)
            toc = timeit.default_timer()
            time_sel.append((toc-tic)*1000000)
            print("selectionsort is done")

            tic = timeit.default_timer()
            mergesort(x)
            toc = timeit.default_timer()
            time_mer.append((toc-tic)*1000000)
            print("mergesort is done")

            tic = timeit.default_timer()
            shellsort(x)
            toc = timeit.default_timer()
```

```
    time_she.append((toc-tic)*1000000)
    print("shellsort is done")

    tic = timeit.default_timer()
    quicksort(x)
    toc = timeit.default_timer()
    time_qui.append((toc-tic)*1000000)
    print("quicksort is done")

    tic = timeit.default_timer()
    sorted(x)
    toc = timeit.default_timer()
    time_sor.append((toc-tic)*1000000)
    print("sorted is done")


bub = sum(time_bub)/25
ins = sum(time_ins)/25
sel = sum(time_sel)/25
mer = sum(time_mer)/25
she = sum(time_she)/25
qui = sum(time_qui)/25
sor = sum(time_sor)/25

writer.writerow((j, bub, ins, sel, mer, she, qui, sor))
print("{} is done\n".format(j))
```