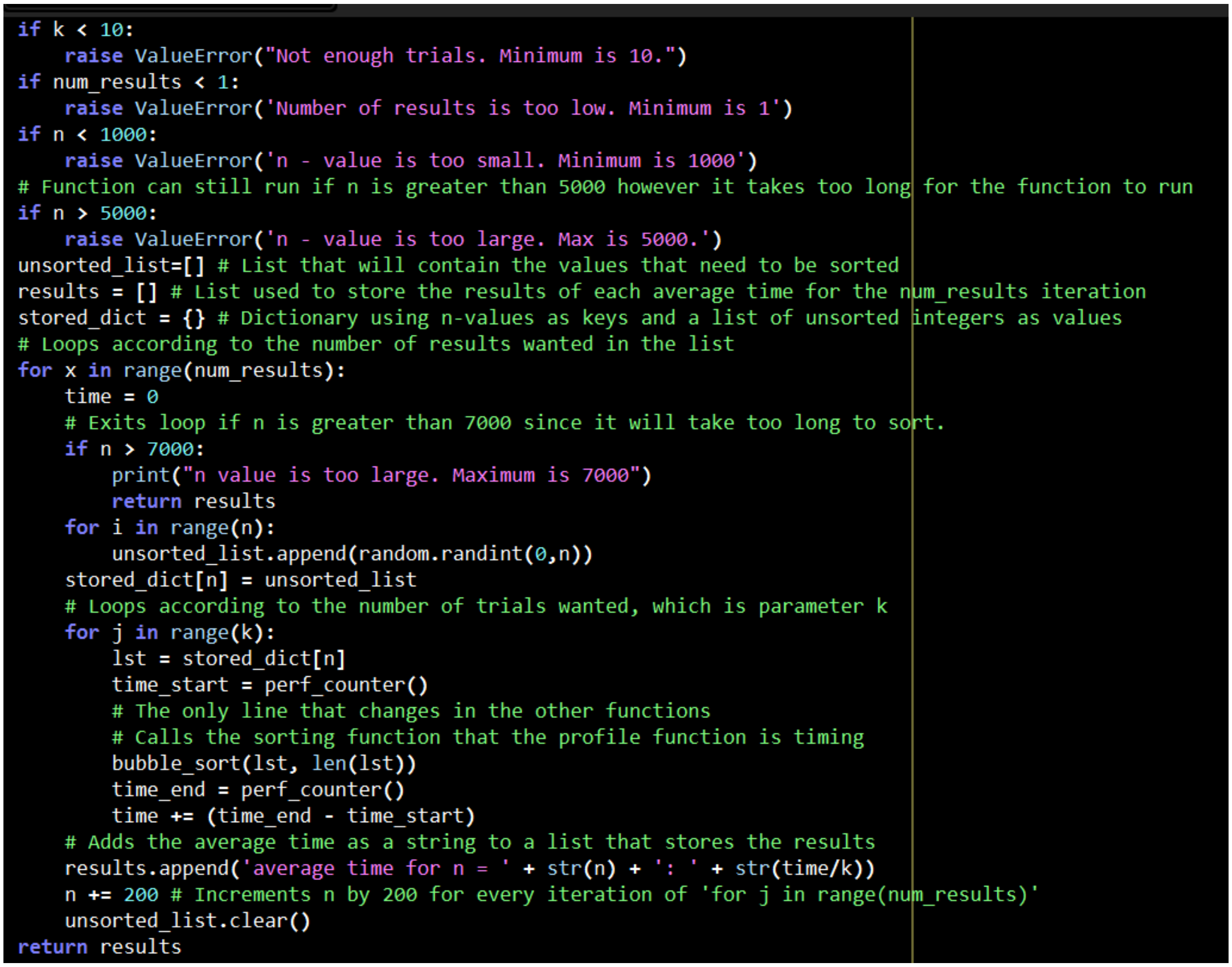
SYSC 2100 Assignment 2 Report

Varrahan Uthayan 101229572
Professor Donald Bailey
SYSC 2100 A
April 12 2023

Introduction

In this assignment, four profile functions were used to time bubble sort, selection sort, insertion sort, and heapsort. The results will then be compared to the theoretical Big-O notation of each sorting function to determine whether the results agree with the theory. These results will then be compared to determine which ones are faster and whether they agree with the theoretical results.

Figure 1: Profile function used to time bubble sort excluding the function header and docstring. The only change that occurs in all profile functions is on the line that the sorting method is called.



```
if k < 10:
    raise ValueError("Not enough trials. Minimum is 10.")
if num_results < 1:
    raise ValueError('Number of results is too low. Minimum is 1')
if n < 1000:
    raise ValueError('n - value is too small. Minimum is 1000')
# Function can still run if n is greater than 5000 however it takes too long for the function to run
if n > 5000:
    raise ValueError('n - value is too large. Max is 5000.')
unsorted_list=[] # List that will contain the values that need to be sorted
results = [] # List used to store the results of each average time for the num_results iteration
stored_dict = {} # Dictionary using n-values as keys and a list of unsorted integers as values
# Loops according to the number of results wanted in the list
for x in range(num_results):
    time = 0
    # Exits loop if n is greater than 7000 since it will take too long to sort.
    if n > 7000:
        print("n value is too large. Maximum is 7000")
        return results
    for i in range(n):
        unsorted_list.append(random.randint(0,n))
    stored_dict[n] = unsorted_list
    # Loops according to the number of trials wanted, which is parameter k
    for j in range(k):
        lst = stored_dict[n]
        time_start = perf_counter()
        # The only line that changes in the other functions
        # Calls the sorting function that the profile function is timing
        bubble_sort(lst, len(lst))
        time_end = perf_counter()
        time += (time_end - time_start)
    # Adds the average time as a string to a list that stores the results
    results.append('average time for n = ' + str(n) + ': ' + str(time/k))
    n += 200 # Increments n by 200 for every iteration of 'for j in range(num_results)'
    unsorted_list.clear()
return results
```

Function Design:

The profile function that was used can be seen in Figure 1. Figure 1 does not include the function header or the docstring. This function takes three integer parameters, one for the n-value which determines the size of the list, the k-value which determines the number of trials that the sorting algorithm will take for the average time, and num_results, which is the number of n-value iterations the result will have. This function creates two empty lists, one used to be sorted, and another to store the average time of each n-value iteration. The function also creates an empty dictionary to store the unsorted list. The function uses three for-loops. The first nested for-loop ‘for i in range(n)’ builds the unsorted list, which is then stored in the dictionary as a value while its respective n-value is used as the key. The second nested for-loop ‘for j in range(k) copies the dictionary value into a list and sorts the list while timing how long each sort takes. The value stored in the dictionary is copied into the new list inside the loop so that when the block loops, the sequence that needs to be organized does not change, and a more accurate time is found. The outermost for-loop ‘for x in range(num_results)’ is used to add the average time to a list of strings that display the average time of each n-iteration. This for-loop increments the n-value by 200 for each iteration in order to find the average time to sort a larger list with a greater variety of integers to sort for the value of num_iter amount of times. The function returns the list of strings containing the n-value and its respective average time in a string.

Results

Table 1: N values and average times for a sort function with 11 trials and 20 n-value iterations where the n-value represents the length of the list as well as the range of which the elements in the indices are selected.

n-value	**Average Times for Sorting Functions**			
	Bubble	**Selection**	**Insertion**	**Heap**
1000	0.05956514545	0.04902582727	0.005463972727	0.01738963636
1200	0.09103192727	0.06800434545	0.008509936364	0.02080095455
1400	0.1251983091	0.09590581818	0.01120825455	0.02558543636
1600	0.1572333636	0.1237959364	0.01515356364	0.02919785455
1800	0.1996794545	0.1531896273	0.01957689091	0.03503500909
2000	0.2518213091	0.1944219455	0.02355165455	0.03831161818
2200	0.3035630909	0.2371661182	0.02825830909	0.04384393636
2400	0.3617570364	0.2886432182	0.03326973636	0.04813059091
2600	0.4311686273	0.3356675273	0.03807546364	0.05216171818
2800	0.5025081545	0.3903490727	0.04467687273	0.05702058182
3000	0.5850470727	0.4433775364	0.05270688182	0.06073731818
3200	0.6726746818	0.5120258364	0.0607237	0.06574529091
3400	0.7791334	0.5729197818	0.06819364545	0.07052517273
3600	0.8717172636	0.6513989	0.07991483636	0.07656068182
3800	0.9579087455	0.7158769818	0.08932555455	0.07924552727
4000	1.082333	0.7949375	0.09807250909	0.08750855455

4200	1.207755445	0.8882880727	0.1071376273	0.09080536364
4400	1.324639618	0.9715865364	0.1157408909	0.09358559091
4600	1.4289208	1.052085273	0.1300717182	0.09968404545
4800	1.579643464	1.144400318	0.1351520273	0.1049511818

Figure 2: The graphical representation of Table 1. The graph plots the points of each type of sorting function and creates a trendline that goes the points to showcase its growth or decay in time based on the size of n. In this graph, the n-value is 1000, the k-value is 11, and the num_results value is 20.

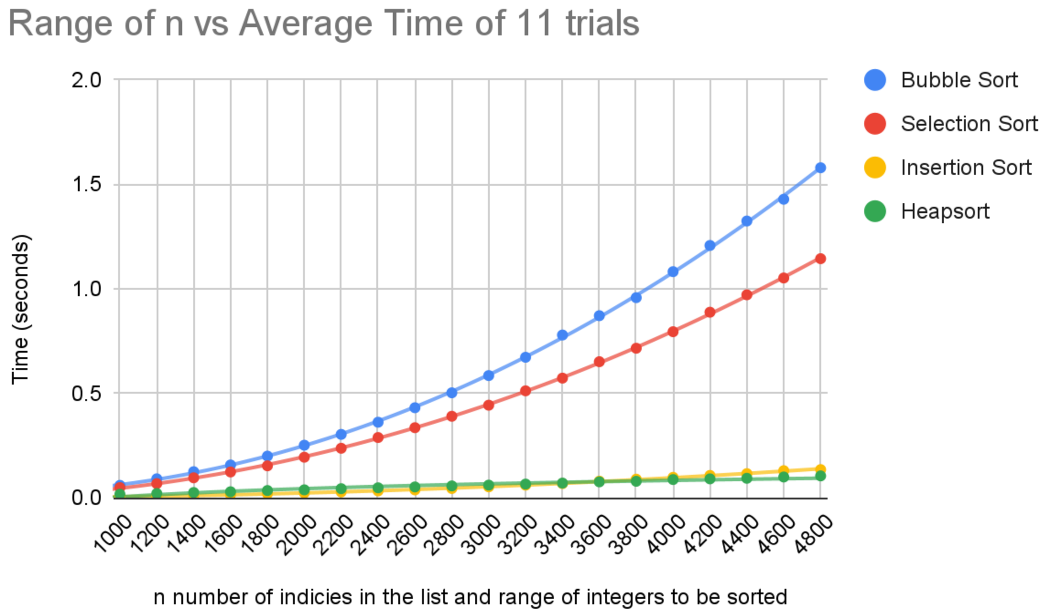
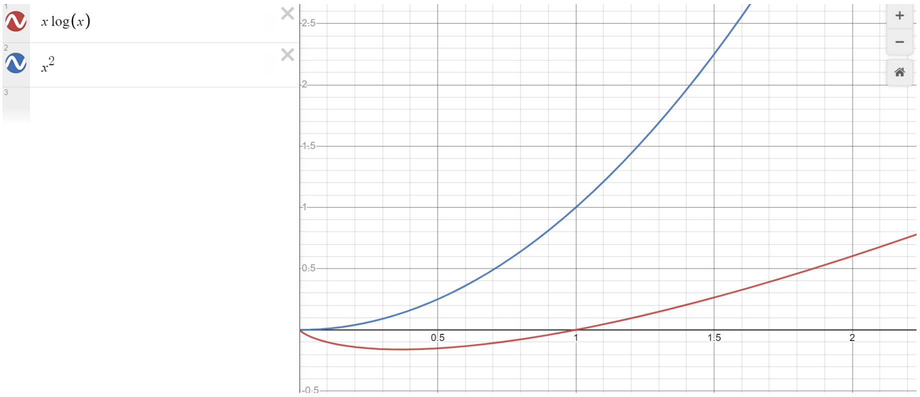


Figure 3: Graphs of the base parabola and xlog(x) function. The functions in Figure 2 will be compared to the overall shapes of these functions in order to determine whether or they fit the Big-O notation.



Analysis

The theoretical Big-O notation of the bubble sort is $O(n^2)$. The bubble sort function in Figure 2 resembles the parabola in Figure 3, as with both cases, the farther along the object is on the x-axis, the steeper the slope of the function is, and the change in slope is discernable on both graphs. Another reason why the result produces $O(n^2)$ as the bubble sort function itself has an inner for-loop that compares and swaps values and an outer for-loop that iterates through each element. Since both for-loops are O(n) and one is nested in another, we use O(n) * O(n) which results in $O(n^2)$.

The theoretical Big-O notation of the selection sort is $O(n^2)$. The selection sort function in Figure 2 resembles the parabola in Figure 3, as with both cases, the farther along the object is on the x-axis, the steeper the slope of the function is, and the change in slope is discernable on both graphs. Another reason why the result produces $O(n^2)$ as the selection sort function itself has an inner for-loop that moves the smallest value to the starting index and an outer for-loop that iterates through each element and sets the starting indices. Since both for-loops are O(n) and one is nested in another, we use O(n) * O(n) which results in $O(n^2)$.

The theoretical Big-O notation of the insertion sort is $O(n^2)$. The insertion sort function in Figure 2 almost looks like a linear function, however, it deviates off the path when compared to a linear function, indicating that the function is actually a parabola and is consistent with the theory. The insertion sort method itself contains an inner for-loop that inserts the element and moves the array to the right and an outer for-loop that iterates through each element of the list, so we also know that the output yields O(n2). We use O(n) * O(n), which yields O(n2) because the for-loop is nested inside of the for-loop and both are O(n).

The theoretical Big-O notation of the heapsort is O(nlogn). The heapsort function in Figure 2 is similar to xlog(x) in Figure 3, in that in both cases, the slope of the function increases as an object moves further along the x-axis, but the change is imperceptible, and the graph seems to be practically linear. As the heapsort function loops trickle-down, there is another explanation for why the result generates O(nlogn), which is O(logn). Due to the trickledown, this generates O(n) from the loop multiplied by O(logn), which is O(nlogn).

The selection sort is faster than the bubble sort, and the insertion sort is faster than the selection sort, but the heapsort is slower than the insertion sort until n = 3600. One reason that this may have occurred is that the sequences that were sorted by the insertion sort may have been very close to their original, as the closer the original sequence is to the desired sequence, the faster the insertion sort will be. This coupled with the notion that the heapsort must first construct the heap in order to sort through it would allow for the insertion sort to be significantly faster than all the other sorting functions.