Project 1 – Two Timing Experiments

Shaivan Bhagat

SCICOMP302 Algorithms and Data Structures

This is all my own work. I have not knowingly allowed others to copy my work. This work has not been submitted for assessment in any other context.

1. Experimental platform

The algorithms were run on a MacBook Air running on a M2 Chip. General specifications of the hardware are as follows:

- Total Number of Cores: 8 (4 performance and 4 efficiency)
- Memory: 8 GB
- Chipset Model: Apple M2
- Type: GPU
- Bus: Built-In

The algorithms were run completely in Python. Python files were run via Spyder, accessed through Anaconda-Navigator 2.4.2. The specific versions of the software are as follows:

- Spyder version: 5.4.3 (conda)
- Python version: 3.11.4 64-bit
- Qt version: 5.15.2
- PyQt5 version: 5.15.7
- Operating System: Darwin 22.5.0

While the algorithms are being run, all cores are being exploited and the CPU load is dedicated specifically to enhancing the performance of the algorithms.

2. Experiment 1

Bubble Sort vs Quick Sort

The performance of bubble sort and quick sort is compared in sorting an array which was generated using a random number generator.

Null Hypothesis (H_0): "The average time complexity and performance of bubble sort and quick sort algorithms, when sorting a randomly ordered dataset, are different with quick sort being more efficient."

Arrays with randomly generated numbers between 0-9999 were sorted by bubble sort and quick sort algorithms. Array sizes used are 10, 50, 100, 1000, 5000, 10000, 20000, 30000,

50000, and 100000. Time (in milliseconds) taken for these two algorithms to sort the various arrays sizes was used as a measure of performance and efficiency. The arrays of the above mentioned sizes were generated randomly using array[i] = int(random.random() * 10000). The program was run until the final array was sorted and then it was repeated for 5 times.

BigOh notation for bubble sort is O(n^2), where n is the number of elements in the array. When elements are already sorted in the array, bubble sort takes the minimum time (best case scenario O(n)). Average case and worst-case time complexities use O(n^2). BigOh notation for quick sort depends on the time complexity. The average case scenario uses O(n log n) while the worst case uses O(n^2).

a. Code:

Bubble Sort:

This sorting algorithm goes through the list one index to the next and compares the values at the two indices. It the order is not in ascending order, the indices are swapped.

Here if index [j] in the array <<arr>> is smaller than at index[j+1], the values are swapped and it continues until all indices in the array are sorted.

```
def bubble_sort(arr):

    n = len(arr)
    for i in range(n - 1):
        flag = 0
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                flag = 1
        if not flag:
            break
```

Quick Sort:

A pivot element is chosen from the array and the rest of the array is divided into two sub-groups: s1 and s2. S1 and s2 are then sorted depending on whether they are higher or lower than the pivot element.

```
def partition(array, first, last):  # function to split array into two parts (s1 < pivot and s2 >= pivot) and
    # determine place of pivot
    last_s1 = first  # sets end of s1
    first_unknown = first + 1  # determines start of part which does not belong to s1 or s2 yet

    while first_unknown <= last:  # loop to go over all unknown part
        if array[first_unknown] < array[first]:
            last_s1 += 1  # increases size of s1 to hold new element
            array[first_unknown], array[last_s1] = array[last_s1], array[first_unknown]  # swaps first element in
            # unknown part with last element in s1
        first_unknown += 1  # point to next element in unknown part

    array[first], array[last_s1] = array[last_s1], array[first]  # swaps first element in array (pivot) with last
    # element in s1 for pivot to be between s1 and s2
    return last_s1

# sorting the array
def sort_by_quicksort(array, first, last):
    if first < last:
        pivot = partition(array, first, last)   # places pivot in the middle of array
        sort_by_quicksort(array, first, pivot - 1)  # sorts first half of array
        sort_by_quicksort(array, pivot + 1, last)  # sorts second half of array
```

b. Data Set

The program was run for a total of 5 times, and the average of time performance was used for this final data table.

Table 1: Average time taken (in milliseconds) after 5 runs for two sorting algorithms: bubble and quick sort

Bubble Sort	Quick Sort	Array Size
0.008821487426757812	0.0040531158447265625	10
0.04315376281738281	0.020265579223632812	50
0.17189979553222656	0.033855438232421875	100
19.73891258239746	0.5130767822265625	1000
529.8168659210205	3.077983856201172	5000
2187.3767375946045	7.699012756347656	10,000
8310.555934906006	14.845848083496094	20,000
19339.173793792725	23.76866340637207	30,000
53033.57005119324	41.86105728149414	50,000
213789.2439365387	88.73605728149414	100,000

c. **Figures**

The graph uses the average values of the 5 performance tests. The first graph was plotted through Python3.11 on a linear scale. The second graph was plotted in Excel (version 16.77) and used an exponential trendline. Graph 2 shows the fitted equations for the two sorting algorithms.

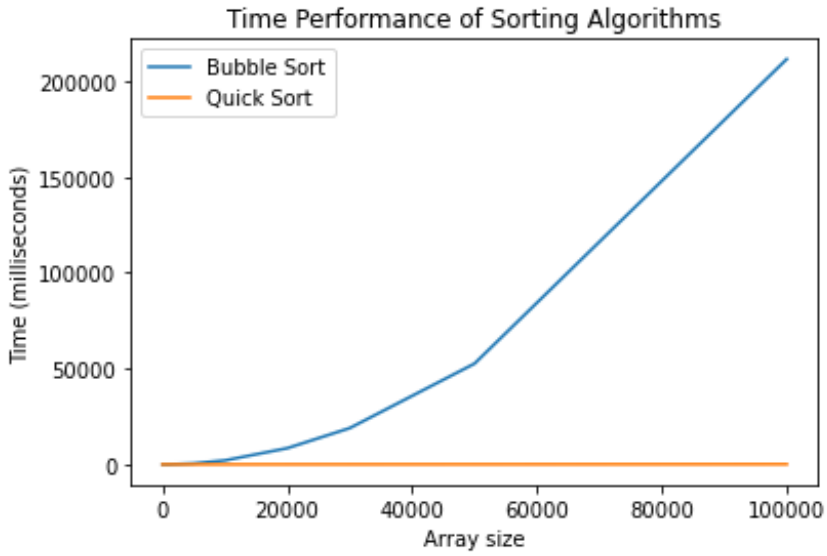

Figure 1: Time performance of bubble sort and quick sort in sorting arrays of various sizes as measured in milliseconds

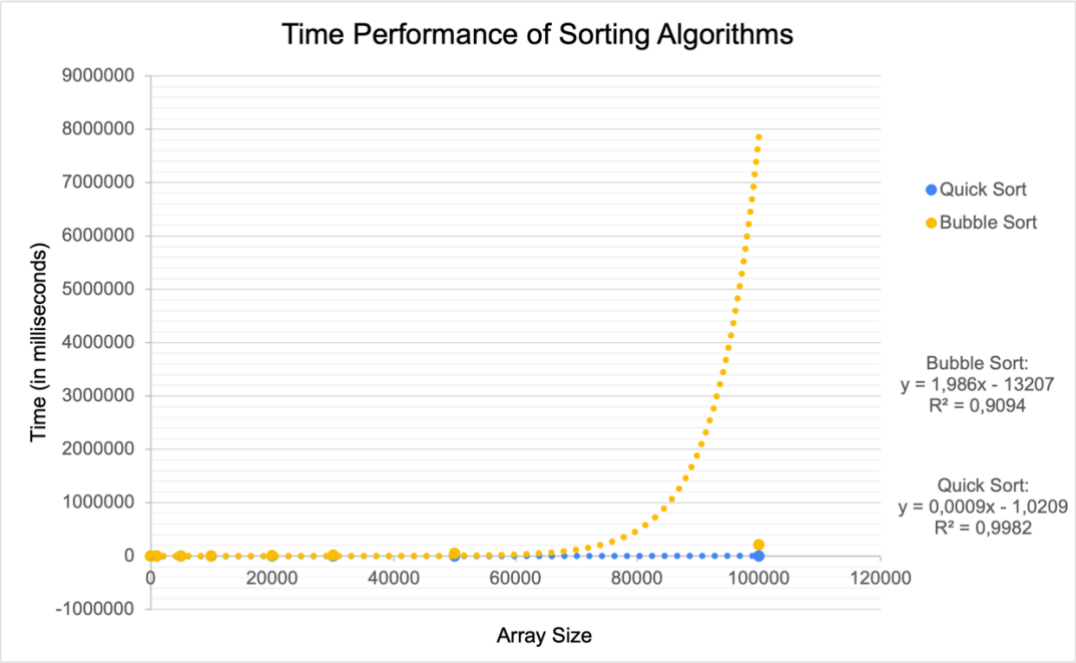

Figure 2: Time Performance of bubble sort and quick sort in sorting arrays of various sizes as measured in milliseconds and plotted with an exponential trendline

d. **Conclusion**:

Based on the graphs above, we can conclude that quick sort is a better sorting algorithm compared to bubble sort. For an array of the same size, bubble sort requires a lot more time to sort the array efficiently. Whereas quick sort returns the result in virtually no time. Based on

this we can say that for larger array sizes, bubble sort should be avoided. We can accept our null hypothesis and conclude that quick sort is indeed more efficient at sorting random arrays.

3. Experiment 2

Selection Sort vs Quick Sort

For experiment 2, I am comparing the performance and efficiency of selection sort with quick sort based on the time taken for these two algorithms to sort arrays of various sizes. The sizes of the arrays are defined by the user, as well as the highest number possible in the array and the number of runs the arrays are to be sorted.

Null Hypothesis (H_o): The average time complexity and performance of selection sort and quick sort algorithms, when sorting a user input-based dataset, would be statistically similar."

For running the program comparing these two algorithms, user input allowed to choose the number of values to be sorted, the highest value possible in the array as well as the number of runs being performed. The maximum value in each run was set to be 10,000. Random numbers were then generated with the maximum value possible being 10,000 which are to be then sorted. The number of values to be sorted (array size) was set from 100, 1000, 5000, 10000, 50000 and 100000. The number of runs performed each time was set to 10. The time taken (in seconds) was saved as csv values on a local file where the time taken was plotted against the array size to determine efficiency and performance of these two algorithms at sorting the same arrays. The Big-Oh for selection sort was O(n^2) where *n* is the number of elements in the array being sorted. Big-Oh notation for quick sort depends on the time complexity. The average case scenario uses O(n log n) while the worst case uses O(n^2).

a. Code:

Selection Sort

The minimum value in the array is selected by searching through all the elements of the array. This minimum value is then placed at the first index [0]. This search continues until all the elements in the array are arranged accordingly.

```
def __init__(self, nums):
    self.nums = nums

def selectionsort(self):
    """
    Runtimes:
    * best case ------- O(n^2)
    * average case ---  O(n^2)
    * worst case ------ O(n^2)
    """
    spot_marker = 0

    while spot_marker < len(self.nums):
        for i in range(spot_marker, len(self.nums)):
            if self.nums[i] < self.nums[spot_marker]:
                self.nums[spot_marker], self.nums[i] = self.nums[i], self.nums[spot_marker]
        spot_marker += 1

    return self.nums
```

Quick Sort

A pivot element is selected in the array. If the element at the index is smaller than the pivot, they are placed in the beginning of the array and if not they are placed after the pivot. This continues until all elements are in ascending order.

```
def quicksorted(self, nums):
    """A helper to the quicksort() function.
    Recursively sorts and combines the 'smaller than', 'equal to', and 'larger than' lists via Quick Sort.
    """
    if len(nums) < 2:
        return nums
    else:
        pivot = nums[-1]
        smaller, equal, larger = [], [], []

        for i in nums:
            if i < pivot:
                smaller.append(i)
            elif i == pivot:
                equal.append(i)
            else:
                larger.append(i)
        return self.quicksorted(smaller) + equal + self.quicksorted(larger)
```

b. Data Set

The table here shows the average amount of time taken in seconds for the two sorting algorithms, selection sort and quick sort, to sort arrays from size 100 – 100000. The test was run for 10 times for each array size and the average of those values are used for plotting the graph.

Table 2: Time taken in seconds for selection and quick sort to sort arrays of different sizes

Selection Sort	Quick Sort	Array Size
0,0005	0,0003	100
0,01952	0,00989	1000
0,441137	0,23439	5000

1,69061	0,77767	10000
34,02965	6,54152	50000
123,2859	13,895	100000

c. Figures

The following figure uses the average values of the 10 performance tests. It was plotted in Excel (version 16.77) and uses a linear trendline. Figure 3 also shows the fitted equations for the two sorting algorithms.

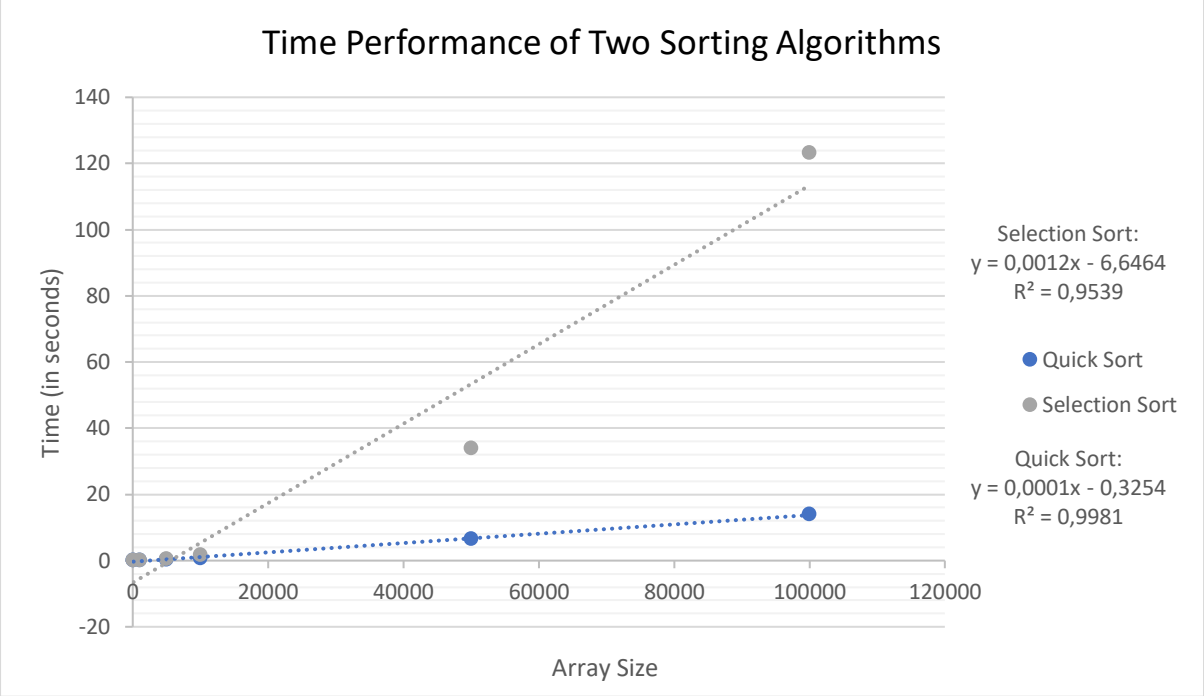

Figure 3 Graph showing the relation between the time taken in seconds for selection and quick sort to sort arrays of various sizes and plotting a linear trendline

d. **Conclusion**

From Figure 3, we can say that the time taken (in seconds) for both selection and quick sort increases as the size of the array increases. However, quick sort is still considerably more efficient at sorting arrays. For selection sort, as the array size increases, the amount of time taken increases linearly as well to the point where it takes almost 100 seconds more to sort the same array. We can reject our null hypothesis and say that selection sort and quick sort do not have the same time complexity and performance, instead quick sort is a lot more efficient at sorting larger arrays.