CS 201 HW 2 Report

Orhun Ege Çelik

CS 201 – Section 3

22202321

Task 1:

In this library task, there are three available sorting algorithms, which are bubble sort, merge sort and quicksort. For an average case, in which the books are randomly distributed, quicksort is the most efficient algorithm and uses minimal memory. On the other hand, when the IDs are arranged in descending order, merge sort proves to be the best, although it uses extra memory. In another rare case, in which the IDs are arranged in ascending order, bubble sort is the fastest and requires minimal memory. To see why the efficiency of these sort algorithms differ, the complexities of each must be explained.

Bubble sort moves the largest element towards the end of the array in each iteration. It compares the elements at indexes i and i+1, then swaps the elements if the element at i is greater than the next one, this way moves the greater element one index towards the end of the array. When the greatest element reaches the end of the array, the same process is repeated, where i starts from index 0, however the sort is done for (size – x), where x denotes the count of iterations through the entire array that have been completed. In the worst case, for an array of size n, bubble sort makes $n^2/2 – n/2$ key comparisons, and the same number of swaps, so the time complexity for bubble sort is $O(n^2)$, and the same holds for average case. However, allocation of additional memory is not needed since all the comparisons are made in the original array, therefore bubble sort has a memory complexity of O(n), which is only the size of the initial array. Although bubble sort works slowly in average and worst cases, it works the best when the array is already sorted, since bubble sort iterates only n times through the array, giving us a time complexity of O(n) in the best case.

Merge sort is a recursive algorithm that divides the array until the size reaches 1, creates subarrays of this divided size, and then merges the subarrays by comparing the smallest elements in the arrays and taking the smaller element of the two arrays, forming a new sorted subarray. The merge function makes as many comparisons as the size of the subarray each time it is called so it is an O(n) operation, and for each step, the merge function is called 2^x times where x denotes the step count. (The sorting starts from step 0 and ends in step log(n).) The total time complexity of each merge function for size $n/2^x$ is $O(n/2^x)$. So, since it is known that there are 2^x such subarrays, to sort all subarrays of size $n/2^x$ the total time needed is O(n). There are log(n) such steps in merge sort, hence the total time complexity of merge sort becomes O(nlog(n)). The best, average and worst-case time complexities are all O(nlog(n)) for merge sort, so whether the initial array was sorted or not does not change the performance of merge sort much. Although merge sort is efficient in terms of time, it uses a considerable amount of memory. In each step, 2^x arrays are created and a total of n elements are dynamically stored in the memory. Since there are log(n) steps, the memory complexity of merge sort becomes O(nlog(n)).

Quick sort is another recursive algorithm which partitions the array according to a pivot, such that elements smaller than the pivot are placed in the first half of the array, and the larger or equal ones in the second half, then quick sorts the smaller and larger half, until the array is completely sorted. The partition function makes as many comparisons as the (size of the subarray-1), so it is an O(n) operation. The number of swaps depends on the order of the array. The array is divided log(n) times, hence there are log(n) steps in quick sort. Therefore, the time complexity of quick sort for average and best cases are O(nlog(n)). However, time complexity for worst case is $O(n^2)$ because when the array is descending sorted, it does not get divided in each quicksort call, only pivot gets placed in its correct position, which results in iterations proportional to n^2. Additional memory is used in this algorithm because of the recursive functions stacked in the call stack, however this report considers only dynamically allocated memory, therefore the memory complexity of quick sort is O(n) since no additional arrays are created.

The random array in this experiment corresponds to the average case for any sorting algorithm. In the library task, when the IDs are given in a random order, quicksort proves to be the best, and bubble sort proves to be the worst sorting algorithm among the three algorithms considered in this report, which is consistent with the time complexity theoretically. As array size increases, the run time of bubble sort increases drastically as in Graph 1, resulting in longer sorting durations. For instance, as in Table 1, for the maximum array size in this experiment, which is 16384 in this report, the run time of bubble sort is 0.498304 seconds, which is considerably higher than those of merge and quick sort, which are 0.0118367 and 0.00136307 seconds respectively. Even if both quicksort and merge sort have a time complexity of O(nlog(n)) in this average case, quicksort sorts the fastest since it only modifies the original array, it does not create additional arrays like merge sort, which reduces the duration of execution. The dynamically allocated memory is smaller than that of merge sort as well, since it is equal to the size of the initial array, while merge sort has a memory complexity of O(nlog(n)).

In the worst case, in which the given array is sorted in a descending order, bubble sort again proves to be the worst sorting algorithm, while merge sort is the best algorithm. For instance, as in Table 3, again using the values for the maximum array size of 16384, bubble sort takes 0.885557 seconds, while merge sort and quick sort take 0.0320003 seconds and 0.186492 seconds respectively. In the worst case, bubble sort and quick sort algorithms make $n^2/2 - n/2$ comparisons, hence their time complexities are $O(n^2)$ and they sort the slowest, while merge sort algorithm sorts the given array fastest because it makes less comparisons, although it still uses more memory than the other algorithms, as in Table 4, with 245760 dynamic allocations, while the other algorithms make 16384 allocations. The allocation amount is the same as the sorting of random array. The faster sort time makes the merge sort the best choice with O(nlog(n)) time complexity, when the book IDs are given in descending order, so the extra memory usage is an acceptable compromise.

Although bubble sort works slowly in average and worst cases, it works the best in the case in which an ascending sorted array is given, since bubble sort makes only n iterations in the array, so it is of time complexity O(n). From the other two algorithms, merge sort works faster than quicksort, this is because quicksort works with $O(n^2)$ time complexity, even if it does not swap any values and only compares them. As in Table 5, for size 16384, the duration of bubble sort is 0.00001142 seconds, which is much less than the durations of other algorithms. The duration of merge sort is 0.0180689 seconds since it is of time complexity O(nlog(n)), it is less than that of quicksort which is 0.0625414 seconds. The memory complexity is the same as before. Since bubble sort does not use extra memory, it is the best choice for a sorted array of IDs.

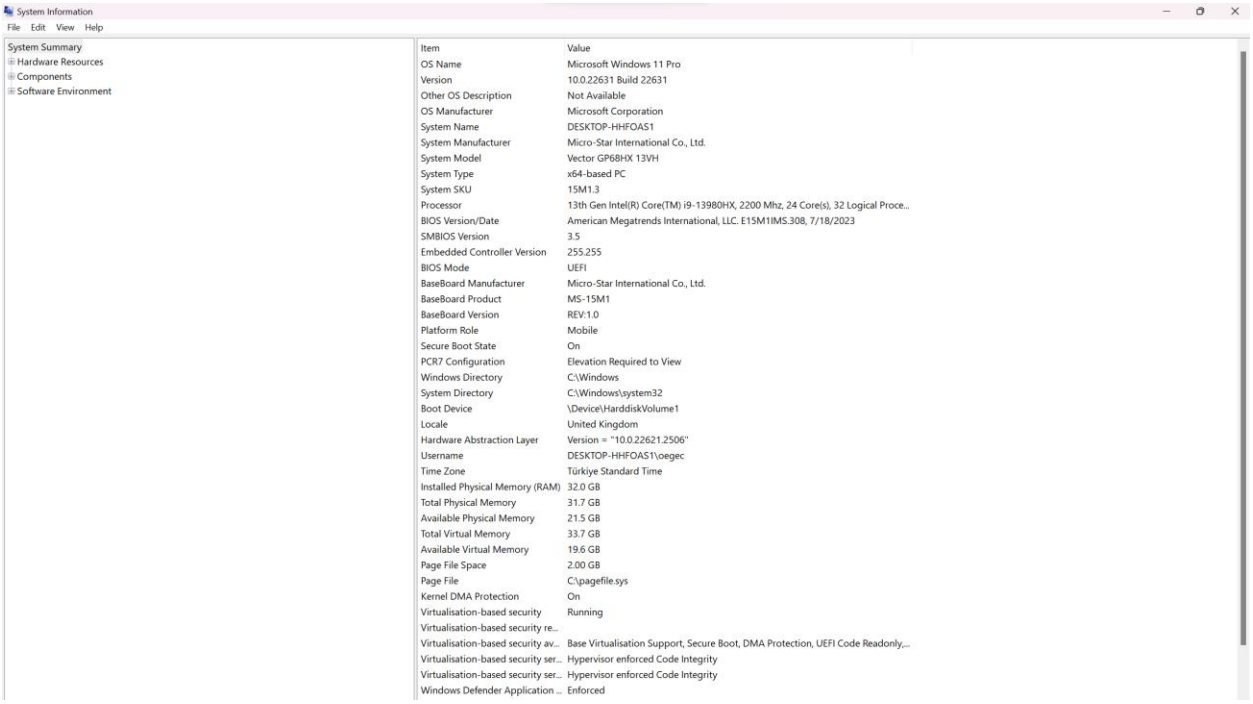
As a result, when the book IDs are given randomly, the best sorting algorithm is quicksort since it runs the fastest with complexity O(nlog(n)) and uses minimal memory, and bubble sort is the worst since it is of time complexity $O(n^2)$. When the book IDs are given as a descending sorted array, the best algorithm is merge sort since it runs much faster with time complexity O(nlog(n)) than the other two which have complexity $O(n^2)$, even though it uses extra memory. The O(nlog(n)) memory complexity is an acceptable compromise for the much faster run time. When the book IDs are given as an ascending sorted array, the best algorithm is bubble sort, since it makes only n iterations, giving the result much faster than the other two algorithms with O(n) complexity. It also does not use extra stack space since it is not a recursive function. From this experiment, it can be concluded that merge sort is the most consistent sort algorithm among all of them, since the run time of merge sort is not much affected by whether the array is sorted or not, but it uses extra memory. It can also be said that quicksort is practically the best sorting algorithm amongst the three overall, because in real life the arrays to be sorted are more likely to be random ones. Finally, bubble sort is useful only when the array is sorted, otherwise other algorithms work better.

Task 2:

If the data is almost sorted, merge sort will be the fastest algorithm, yet bubble sort will be the best fitting algorithm. In the code, we generate an almost sorted array, such that theoretically only 10% of the array is not sorted. Compared to an entirely random array, this means that merge sort will make much less comparisons because either the left array or the right array element will frequently be smaller. Bubble sort will also do a considerably smaller number of comparisons since its boolean value "sorted" will quickly become true and it will stop iterating. Quicksort works worse than both as the array becomes almost sorted, because in this case the partitioning does not divide the array into two equal parts, the part for elements smaller than the pivot has too few elements, and the part for elements larger than the pivot has too many elements, opposite to the aim of quicksort which divides and conquers. This argument can be supported from the data obtained from my experiment. For size >= 1024, merge sort runs the fastest. For instance, as in Table 7, for size 1024 merge sort runs in 0.00027645 seconds, while bubble sort and quicksort run at 0.00030051 and 0.00155922 seconds respectively. However, as in Table 8, the memory allocated for merge sort is 11264, where for bubble sort and quicksort it is 1024. Similarly, when the size gets larger, at size 16384, merge sort runs at 0.0174293 seconds while bubble sort and merge sort run at 0.0549982 and 0.272809 seconds respectively, whereas the memory allocated for merge sort is 245760 and for bubble and quick sort it is 16384, the same as in task 1. So, the time complexity for merge sort is upper bounded by O(nlog(n)) but its memory complexity is also O(nlog(n)). Quick sort's time complexity is upper bounded by $O(n^2)$, but its memory has complexity O(n). Bubble sort's time complexity is greater than merge sort but lower than quicksort, hence it is upper bounded by $O(n^2)$ again but practically it takes much less than quicksort, and it has memory complexity of O(n). These data prove that merge sort runs the fastest when the array is already sorted but it uses a lot more memory, and it also proves that bubble sort works faster than quicksort. For smaller sized arrays, (i.e., size <=512) bubble sort runs the fastest, for instance for size 128 it takes 0.00000443 seconds, while merge and quick sort algorithms take 0.00003111 and 0.00002343 seconds respectively. This is because the access to temporary array in merge sort and uneven partitioning in quick sort make them slower. How the execution duration changes with respect to array size can also be observed from the graph titled "Almost Sorted Array – Time Complexity" (Graph 4). Therefore, it can be said that if the given array is smaller than a certain size, bubble sort maybe the best fitting algorithm, in terms of both time and memory.

All in all, even if merge sort runs much quicker, there is the compromise of memory allocation. As it can be seen from the graph "Memory Complexity for all Tests" (Graph 2), as the input array size increases, the memory complexity of merge sort increases much more compared to the other two algorithms. So, if the machine that is going to sort the book IDs has enough memory space and the time must be the smallest, merge sort will be the best choice. However, in terms of time complexity, there is not much difference between bubble sort and merge sort, hence if there needs to be a balance between memory usage and time complexity, bubble sort will be the best choice. Merge sort will only be preferrable if the time complexity difference between bubble sort and merge sort increases due to larger input array size, for instance when $n = 2^{20}$ or larger, otherwise, the memory space compromise is not needed for the library. Considering both time and memory, bubble sort will be the best choice for the library.

Computer Specifications



Item	Value
OS Name	Microsoft Windows 11 Pro
Version	10.0.22631 Build 22631
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	DESKTOP-HHFOAS1
System Manufacturer	Micro-Star International Co., Ltd.
System Model	Vector GP68HX 13VH
System Type	x64-based PC
System SKU	15M1.3
Processor	13th Gen Intel(R) Core(TM) i9-13980HX, 2200 Mhz, 24 Core(s), 32 Logical Proce...
BIOS Version/Date	American Megatrends International, LLC. E15M1IMS.308, 7/18/2023
SMBIOS Version	3.5
Embedded Controller Version	255.255
BIOS Mode	UEFI
BaseBoard Manufacturer	Micro-Star International Co., Ltd.
BaseBoard Product	MS-15M1
BaseBoard Version	REV:1.0
Platform Role	Mobile
Secure Boot State	On
PCR7 Configuration	Elevation Required to View
Windows Directory	C:\Windows
System Directory	C:\Windows\system32
Boot Device	\Device\HarddiskVolume1
Locale	United Kingdom
Hardware Abstraction Layer	Version = "10.0.22621.2506"
Username	DESKTOP-HHFOAS1\oegec
Time Zone	Türkiye Standard Time
Installed Physical Memory (RAM)	32.0 GB
Total Physical Memory	31.7 GB
Available Physical Memory	21.5 GB
Total Virtual Memory	33.7 GB
Available Virtual Memory	19.6 GB
Page File Space	2.00 GB
Page File	C:\pagefile.sys
Kernel DMA Protection	On
Virtualisation-based security	Running
Virtualisation-based security re...	
Virtualisation-based security av...	Base Virtualisation Support, Secure Boot, DMA Protection, UEFI Code Readonly,...
Virtualisation-based security ser...	Hypervisor enforced Code Integrity
Virtualisation-based security ser...	Hypervisor enforced Code Integrity
Windows Defender Application ...	Enforced

Figure 1

Random Array – Time Complexity (in seconds)

n	Bubble Sort	Merge Sort	Quick Sort
2^6	0.00003656	0.00006026	0.00000554
2^7	0.0001057	0.00009158	0.00001184
2^8	0.00036554	0.00019167	0.00002023
2^9	0.00089059	0.00024279	0.00003682
2^{10}	0.00224282	0.00042092	0.0001021
2^{11}	0.00712298	0.00059832	0.00016756
2^{12}	0.0392367	0.00262717	0.00035598
2^{13}	0.178557	0.00345227	0.00073896
2^{14}	0.498304	0.0118367	0.00136307

Table 1

Random Array – Memory Complexity

n	Bubble Sort	Merge Sort	Quick Sort
2^6	64	448	64
2^7	128	1024	128
2^8	256	2304	256
2^9	512	5120	512
2^{10}	1024	11264	1024
2^{11}	2048	24576	2048
2^{12}	4096	53248	4096
2^{13}	8192	114688	8192
2^{14}	16384	245760	16384

Table 2

Sorted Array (Descending) – Time Complexity

n	Bubble Sort	Merge Sort	Quick Sort
2^6	0.00001104	0.0000192	0.00000681
2^7	0.0000425	0.00005065	0.00002998
2^8	0.00025426	0.00016647	0.00015191
2^9	0.00100056	0.0002832	0.00055336
2^{10}	0.00471981	0.00040359	0.00171268
2^{11}	0.00979439	0.00048898	0.00480748
2^{12}	0.0260946	0.00144093	0.0177088
2^{13}	0.287662	0.00309722	0.0502626
2^{14}	0.885557	0.0320003	0.186492

Table 3

Sorted Array (Descending) – Memory Complexity

n	Bubble Sort	Merge Sort	Quick Sort
2^6	64	448	64
2^7	128	1024	128
2^8	256	2304	256
2^9	512	5120	512
2^{10}	1024	11264	1024
2^{11}	2048	24576	2048
2^{12}	4096	53248	4096
2^{13}	8192	114688	8192
2^{14}	16384	245760	16384

Table 4

Sorted Array (Ascending) – Time Complexity

n	Bubble Sort	Merge Sort	Quick Sort
2^6	0.00000044	0.00004833	0.00000701
2^7	0.00000069	0.00010182	0.00002501
2^8	0.00000102	0.0002367	0.00007925
2^9	0.00000143	0.00031981	0.00020364
2^{10}	0.00000205	0.00059955	0.00071325
2^{11}	0.00000478	0.00134581	0.00174331
2^{12}	0.00000651	0.00225189	0.00494851
2^{13}	0.00000694	0.00342669	0.0179551
2^{14}	0.00001142	0.0180689	0.0625414

Table 5

Sorted Array (Ascending) – Memory Complexity

n	Bubble Sort	Merge Sort	Quick Sort
2^6	64	448	64
2^7	128	1024	128
2^8	256	2304	256
2^9	512	5120	512
2^{10}	1024	11264	1024
2^{11}	2048	24576	2048
2^{12}	4096	53248	4096
2^{13}	8192	114688	8192
2^{14}	16384	245760	16384

Table 6

Almost Sorted Array – Time Complexity

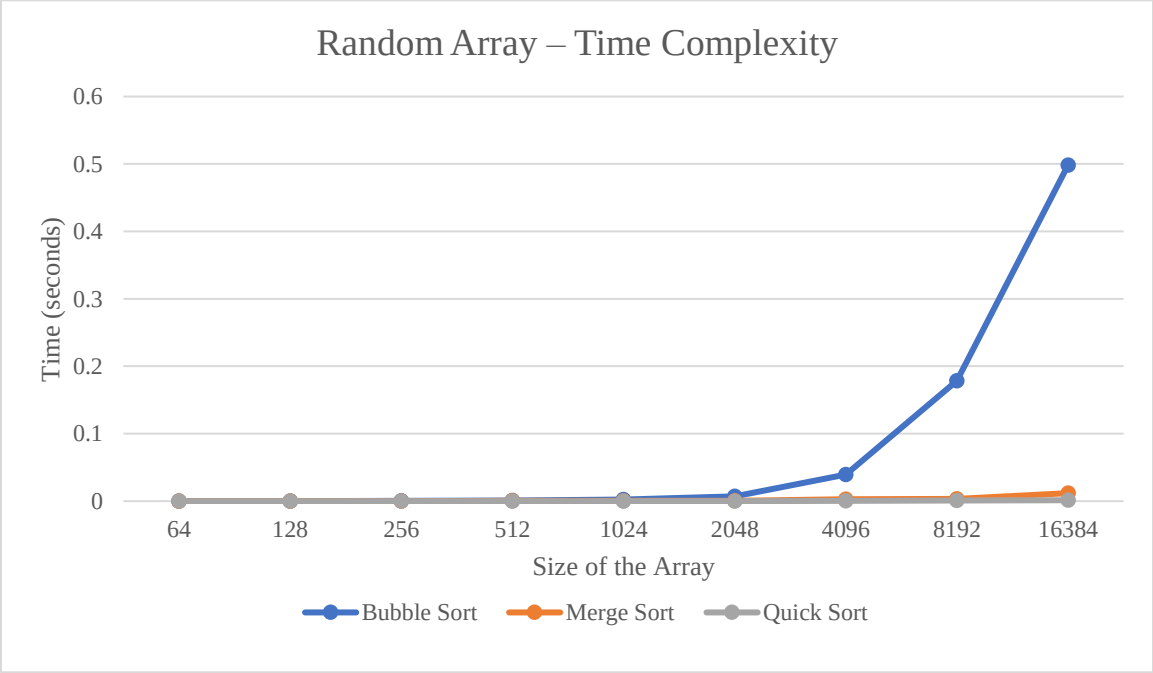
n	Bubble Sort	Merge Sort	Quick Sort
2^6	0.00000153	0.00001338	0.00000549
2^7	0.00000443	0.00003111	0.00002343
2^8	0.00001971	0.0005749	0.00007771
2^9	0.00005782	0.0001073	0.00020364
2^{10}	0.00030051	0.00027645	0.00155922
2^{11}	0.00135297	0.000708	0.00588122
2^{12}	0.00495785	0.00190941	0.0220772
2^{13}	0.0143904	0.00260497	0.0692753
2^{14}	0.0549982	0.0174293	0.272809

Table 7

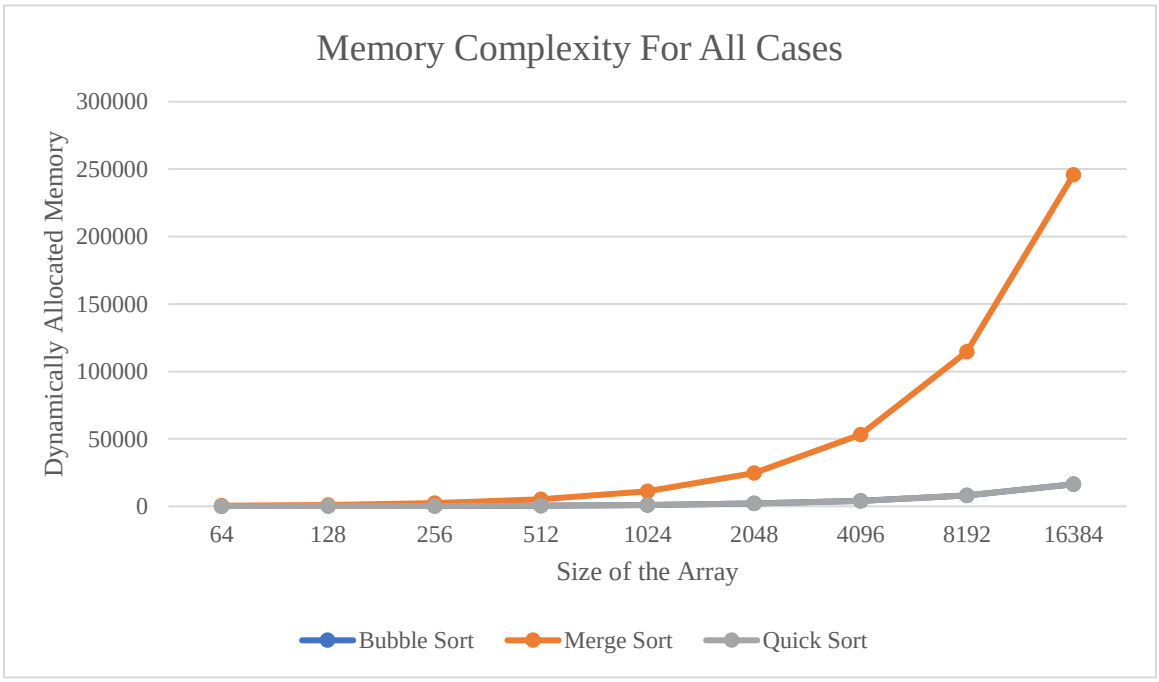
Almost Sorted Array – Memory Complexity

n	Bubble Sort	Merge Sort	Quick Sort
2^6	64	448	64
2^7	128	1024	128
2^8	256	2304	256
2^9	512	5120	512
2^{10}	1024	11264	1024
2^{11}	2048	24576	2048
2^{12}	4096	53248	4096
2^{13}	8192	114688	8192
2^{14}	16384	245760	16384

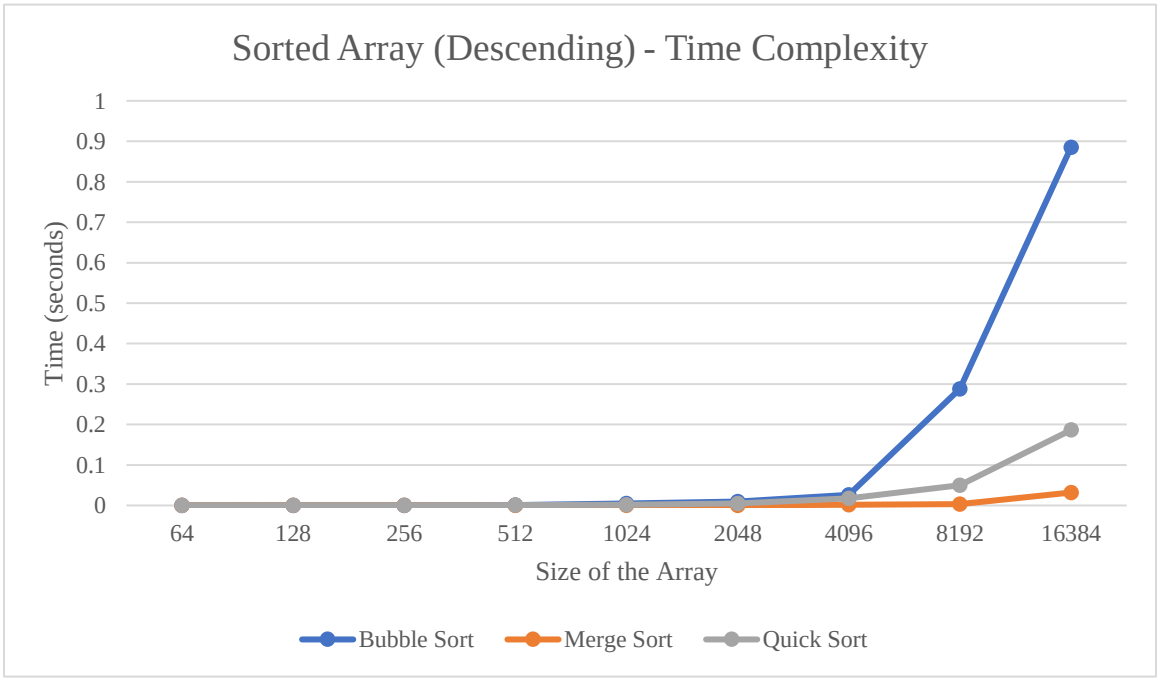
Table 8



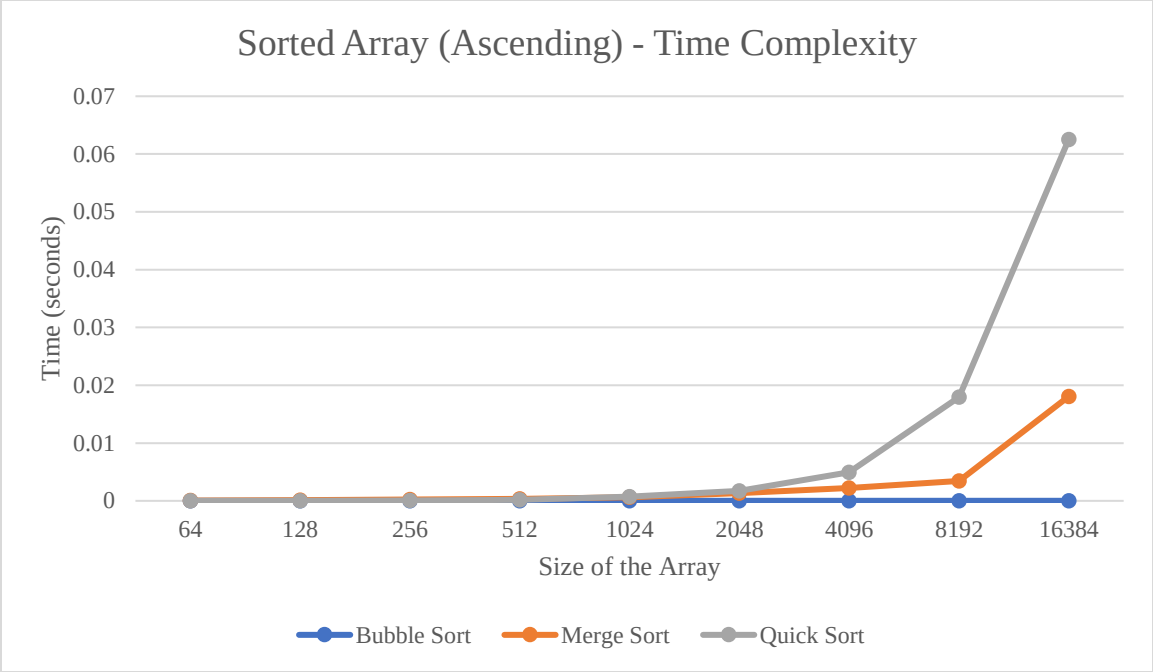
Graph 1



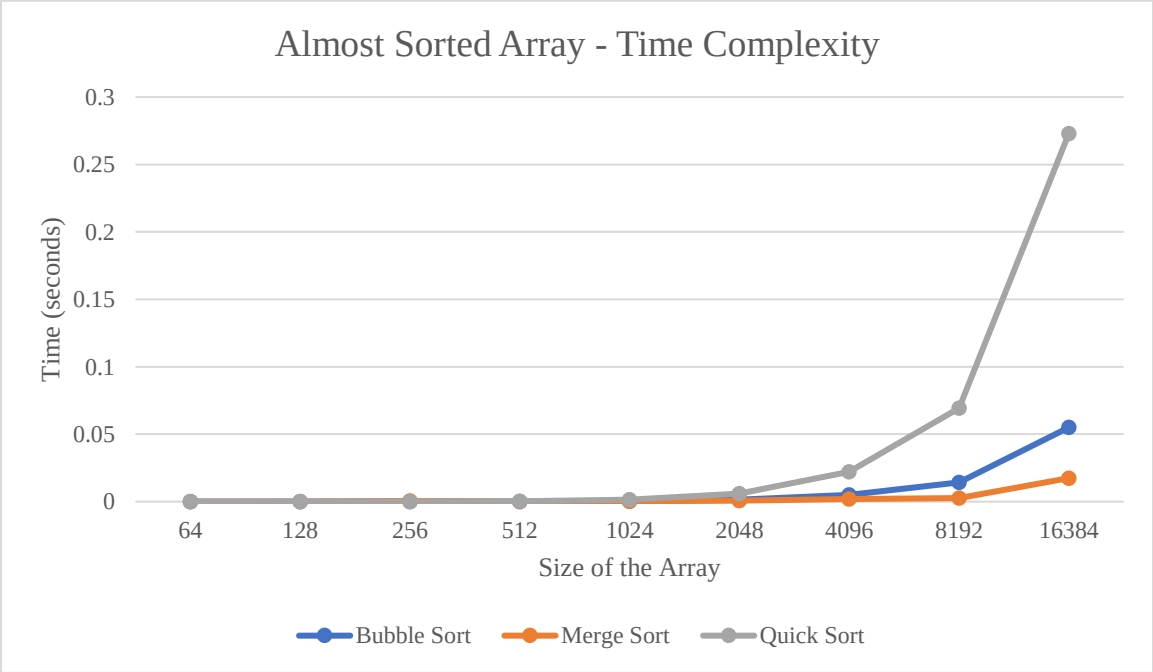
Graph 2



Graph 3



Graph 4



Graph 5