PA1

Student name: *Chia-Wen Fan*

Course: *Algorithms (EE4033-03)*
Due date: *Oct. 5th, 2024*

Requirement 1: Comparing five algorithms of different input sizes.

CPU time and memory usage of the five algorithms with different input sizes

The sorting runs on EDA union lab machines.

Input Size	IS		MS		BMS		QS		RQS	
	CPU Time (ms)	Memory (KB)	CPU Time (ms)	Memory (KB)	CPU Time (ms)	Memory (KB)	CPU Time (ms)	Memory (KB)	CPU Time (ms)	Memory (KB)
4000.case2	0.094	5908	2.116	6044	1.607	6052	15.905	6036	21.259	5908
4000.case3	8.886	5908	2.188	6044	1.096	6052	16.264	5908	21.314	5908
4000.case1	3.235	5908	2.4	6044	2.414	6052	0.977	5908	21.11	5908
16000.case2	0.101	6060	3.03	6060	2.647	6220	164.013	6940	63.045	6060
16000.case3	62.234	6060	3.643	6060	3.303	6220	124.162	6436	61.036	6060
16000.case1	35.337	6060	5.223	6060	6.01	6220	2.048	6060	63.074	6060
32000.case2	0.111	6192	6.079	6320	5.106	6260	617.731	8008	123.861	6192
32000.case3	246.785	6192	5.677	6320	3.137	6260	482.158	6988	124.878	6192
32000.case1	123.91	6192	8.094	6320	7.505	6260	3.437	6192	121.793	6192
1000000.case2	1.129	12148	143.488	16240	111.877	18292	598877	72476	3731.31	12148
1000000.case3	259964	12148	156.951	16240	121.596	18292	326746	33020	3734.62	12148
1000000.case1	127296	12148	235.991	16240	224.251	18292	85.751	12148	3764.89	12148

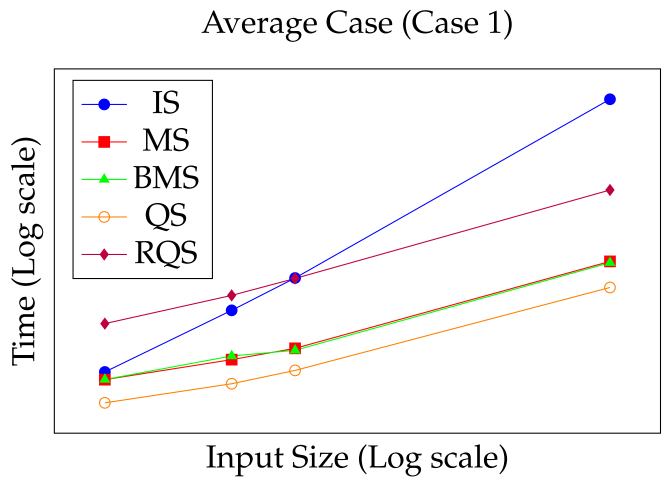

Figure 1: Average Case (Case 1)

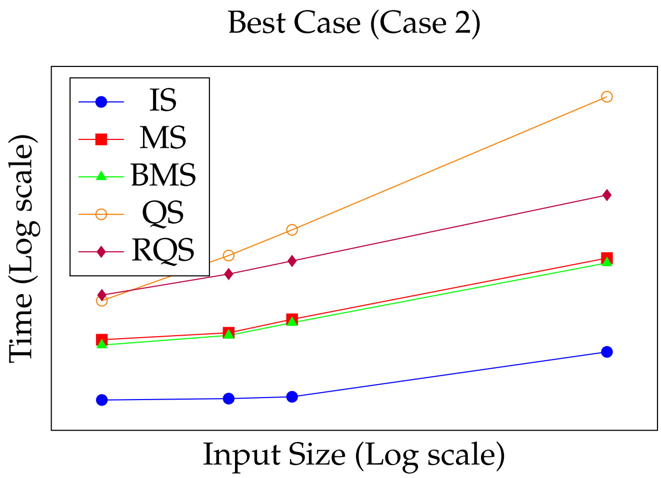

Figure 2: Best Case (Case 2)

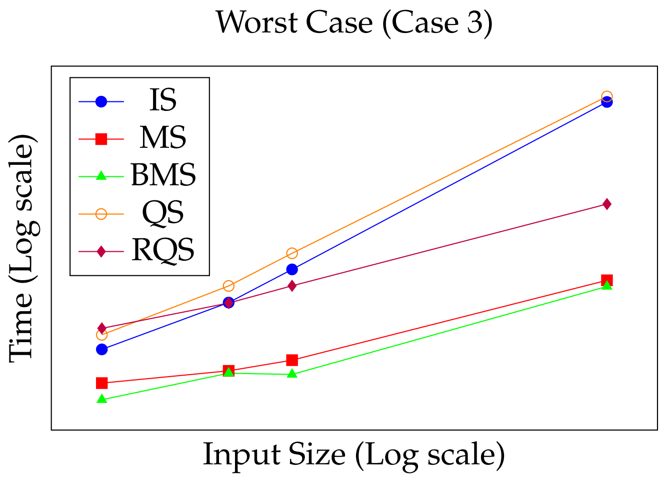

Figure 3: Worst Case (Case 3)

Analysis of Differences Between the Provided Graphs and the Sample Graph

The trend lines generated with my data are generally as expected. **In the average case**, insertion sort shows a steeper slope, which match the theoretical complexity $O(n^2)$. The other four algorithms all show more gradual growth of complexity $O(n \lg n)$. **In the best case**, since the data has been sorted in ascending order, insertion sort is as fast as expected, which runs in $O(n)$. Merge sort, bottom-up merge sort, and randomized quick sort are as expected as $O(n \lg n)$. **In the worst case**, since the data has been sorted in descending order, insertion sort is as slow as expected, which runs in $O(n^2)$.
Only only different from the textbook is quick sort have the same time tendency with insertion sort in the worst case as well as in the best case. The reason is that, data is ordered reversely in the worst case and ordered in increasing order, which means that every time it picks the pivot, the smallest number or the largest number is picked and the array is split into $1 : n-1$. Hence, the worst runtime by quick sort is as bad as insertion sort, which is $\Theta(n^2)$.

Requirement 2: Comparing MS and BMS

Observe the runtime of merge sort and bottom-up merge sort, we found that the trend lines are almost identical in the three cases. The primary reason for their similarity is that they both perform the same sequence of merges on subarrays of increasing size. The following are the comparisons of the two algorithms:

Merging process: In the merging process, both merge sort and bottom-up merge sort takes $O(n)$ to sort the subarrays with n elements.

Number of levels: Moreover, both algorithms operate in $\lg n$ levels. Merge sort finishes this through recursive halving, and bottom-up merge sort finishes this by iteratively doubling the size of the sublists being merged. In both cases, the total number of levels required to reach a fully sorted array is proportional to $\lg n$.

Total numbers of operations: At each level of recurrence or iteration, n comparisons are performed and there are $\lg n$ levels, causing a total $O(n \lg n)$ runtime.

Requirement 3: Comparing QS and RQS

Quick sort and randomized quick sort have an average time complexity of $O(n \lg n)$, meaning for most inputs, they both perform similarly and scale well with the input size. However, we would observe distinct trends in worst-case performance. Quick sort shows a clear worst-case scenario when the pivot selection is consistently poor, especially with ordered or nearly ordered data, which would cause a runtime of $O(n^2)$. On the other hand, since the pivot in randomized quick sort is selected randomly, this algorithm is able to prevent the worst case scenario and maintain a runtime of $O(n \lg n)$. However, we should notice that the randomized process would have some extra cost on runtime than the normal quick sort.

Requirement 4: Data structure used and other findings.

I use vector as the primary data structure. I found two ways to improve my program. The first method is changing the data structure used in merge sort from vectors to arrays. The second is to use a more efficient random number generator in randomized quick sort. (To keep the homework format as required, all my revised code will be on this GitHub repo.)

Replace vectors with arrays in MS and BMS

I found that using pre-allocated memory would make the sorting slightly more efficient than dynamically allocated memory. I replaced the primary data structure used with array in merge sort, and the following is the result (the program is executed on my local terminal).

Input Size	MS CPU Time (ms)		BMS CPU Time (ms)	
	Vector	Array	Vector	Array
4000.case2	10.559	11.902	6.389	4.555
4000.case3	6.115	7.022	5.687	5.756
4000.case1	7.026	5.892	7.01	5.549
16000.case2	14.501	12.521	14.095	11.917
16000.case3	13.309	11.567	10.888	8.936
16000.case1	14.595	12.792	13.871	11.805
32000.case2	22.149	17.726	16.951	15.968
32000.case3	25.027	18.976	19.042	18.021
32000.case1	24.433	22.99	24.556	17.307
1000000.case2	302.028	266.473	283.538	238.428
1000000.case3	307.498	270.324	292.513	240.553
1000000.case1	355.314	308.568	342.016	265.853

As shown above, using pre-allocated arrays causes a slight improvement on the runtime performance of the sorting. Despite of that, we still should carefully choose data structure used since manually managing memory would cause extra time to maintain. Hence, although using arrays is slightly more efficient than using vectors, there is not necessarily a better way to implement the algorithms in real world. It should depend on different scenarios.

More efficient random number generator

Also, I found that my randomized quicksort is less efficient than expected. In other words, it seems that at each iteration of the randomized process, some costly operations are slowing down the sorting. I eventually figure out that the random number generator I used in my original code ($random\_device$ and $mt19937$) are used for strong randomness, which can be significantly slower than the traditional $std :: rand()$. I fixed this by replace them with $std :: rand()$.

Input Size	RQS CPU Time (ms)	
	Strong randomness	Weak randomness
4000.case2	11.912	9.621
4000.case3	8.97	6.462
4000.case1	8.646	4.999
16000.case2	29.952	12.191
16000.case3	28.902	11.965
16000.case1	26.444	11.133
32000.case2	40.844	17.185
32000.case3	43.972	20.925
32000.case1	46.813	21.754
1000000.case2	770.335	250.776
1000000.case3	778.96	250.176
1000000.case1	805.06	286.047

As shown above, using the traditional random number generator ($std :: rand$) would be more efficient in runtime.