SORTING ALGORITHMS: RELATIONSHIP BETWEEN INPUT AND RUNTIME

IIT ROPAR CS506 LAB 1

Ishan Tripathi – 2021AIM1009

Objective

To implement sorting algorithms and study the effect of input array and array size on the respective execution time of these algorithms. The sorting algorithms to be considered are:

1. Bubble Sort
2. Insertion Sort
3. Selections Sort
4. Merge Sort
5. Quick Sort

Methodology

The sorting algorithms mentioned above are implemented in C programming language. The program is compiled and executed on a personal computer (*refer Appendix*). Given a scenario, identical input array is passed to each algorithm for fair comparison. There are 2 global variables to be defined:

1. CASE – for configuring the nature of the input array
2. ARRAY_SIZE – for configuring the size (or length) of the input array

For overall study of above sorting algorithms, we seek to consider the following types of input arrays that are most passed to a sorting algorithm:

1. Random Array – elements chosen randomly using rand()
2. Sorted Array – elements arranged in ascending order
3. Reverse Sorted Array – elements arranged in non-ascending order
4. Partially Sorted Array – elements arranged in ascending order, except approx. 10-20%

Also, the following array sizes are considered for measuring runtime in milliseconds:

1. 10 thousand
2. 20 thousand
3. 30 thousand

The C program is built keeping modularity and Service Oriented Architecture (SOA) in mind. Moreover, to record the runtime of every scenario (CASE * ARRAY_SIZE) of each algorithm, the sorting technique has been executed at least 5 times and the average runtime is taken as the final reading for that scenario. Refer to the Appendix given at the end of this report for further details.

How to run the code?

To run the C program, we need to configure our input array by defining:

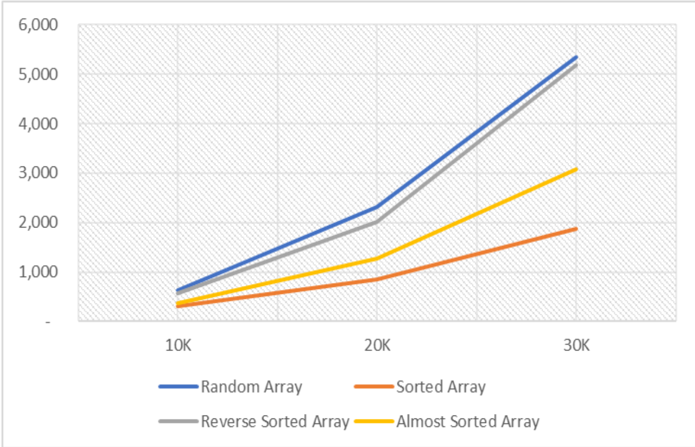
1. CASE: It can have values from 1 to 4 based on the above types of input arrays
2. ARRAY_SIZE: It can be 10000, 20000 or 30000 referring to the above input array sizes

Once the program is executed, the output will contain the execution time taken by each of the five sorting techniques for the identical input array. One can uncomment the printf() statements to verify the results, if needed.

Readings

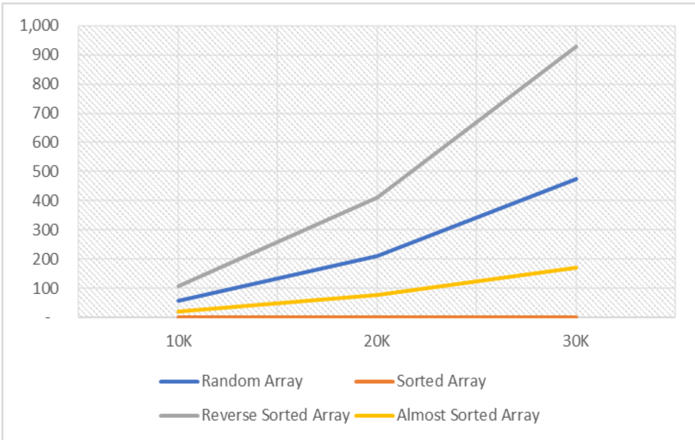
Bubble Sort: The respective average execution time (in milliseconds) observed for each of the four types of input arrays is given below:

BUBBLE SORT RUNTIME (in ms)		*ARRAY_SIZE*		
		10K	**20K**	**30K**
CASE	Random Array	636	2,309	5,335
	Sorted Array	305	842	1,874
	Reverse Sorted Array	577	2,019	5,180
	Almost Sorted Array	370	1,269	3,085



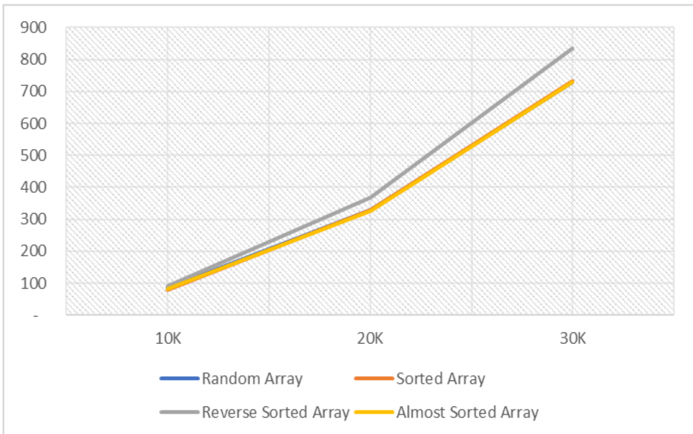
Insertion Sort: The respective average execution time (in milliseconds) observed for each of the four types of input arrays is given below:

INSERTION SORT RUNTIME (in ms)		*ARRAY_SIZE*		
		10K	**20K**	**30K**
CASE	Random Array	57	210	476
	Sorted Array	-	-	-
	Reverse Sorted Array	108	410	929
	Almost Sorted Array	20	78	169



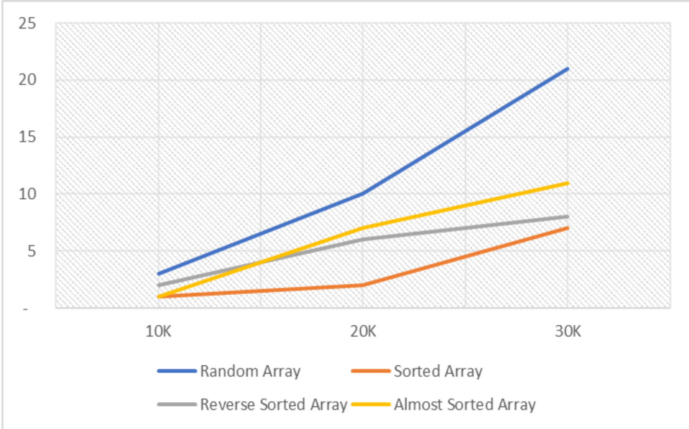
Selection Sort: The respective average execution time (in milliseconds) observed for each of the four types of input arrays is given below:

SELECTION SORT RUNTIME (in ms)		*ARRAY_SIZE*		
		10K	**20K**	**30K**
CASE	Random Array	85	328	729
	Sorted Array	80	330	732
	Reverse Sorted Array	92	367	834
	Almost Sorted Array	83	327	730



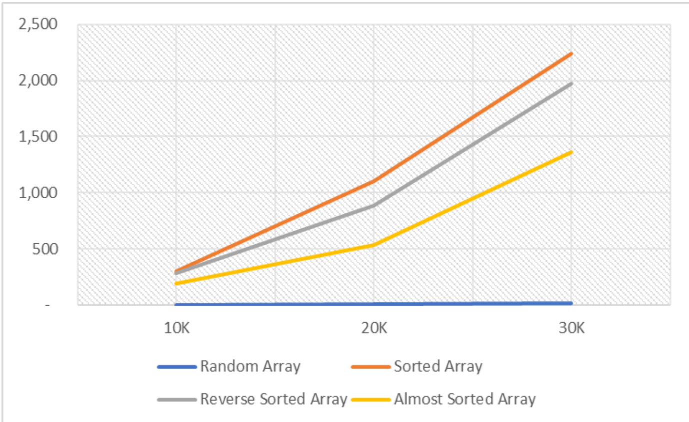
Merge Sort: The respective average execution time (in milliseconds) observed for each of the four types of input arrays is given below:

MERGE SORT RUNTIME (in ms)		*ARRAY_SIZE*		
		10K	**20K**	**30K**
CASE	Random Array	3	10	21
	Sorted Array	1	2	7
	Reverse Sorted Array	2	6	8
	Almost Sorted Array	1	7	11



Quick Sort: The respective average execution time (in milliseconds) observed for each of the four types of input arrays is given below:

QUICK SORT RUNTIME (in ms)		*ARRAY_SIZE*		
		10K	**20K**	**30K**
CASE	Random Array	1	4	15
	Sorted Array	301	1,103	2,239
	Reverse Sorted Array	286	886	1,968
	Almost Sorted Array	193	530	1,362



Observations

Based on the above readings, we observe that Selection Sort and Merge Sort are not 'smart' sorting techniques as they show similar runtime trends irrespective of the nature of the input array passed to them. On the other hand, Bubble Sort, Insertion Sort and Quick Sort are 'smart' as they show significant variation in runtimes depending upon the nature of input array passed to them.

A summary of the order of runtime (big-O) for the above algorithms can be given as:

Time Complexity	Best Case	Average Case	Worst Case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$

Conclusion

Best case of Bubble Sort and Insertion Sort is an already sorted array passed as an input. However, Insertion Sort takes significantly lower runtime due to minimal write-operations (swapping). As each swap operation typically takes 3 write-operations, Bubble Sort runtime increases significantly with increase in array size. Similarly, Selection Sort offers even faster results as it keeps track of the index of elements and performs the minimum number of swaps, i.e. write operations. For an already sorted array, Bubble Sort proves to be more efficient than Selection Sort as it compares with the next element, whereas Selection Sort will traverse the entire subarray to the right to find the minimum element.

Merge Sort and Quick Sort follow divide and conquer approach to sort the input array. But Merge Sort performs blind partitioning as opposed to Quick Sort which performs smart partitioning based on a pivot element. Merge Sort is, therefore, not the best sorting technique when it comes to small size input arrays. The success of Quick Sort depends heavily on the ordering of elements rather than the elements themselves. If the chosen pivot is an extreme value (as first/last element in any kind of sorted array), the algorithm fails to efficiently divide the array – kind of like a skewed tree.

Here are the case-wise best algorithms based on this study:

1. Random Array – Quick Sort
2. Sorted Array – Insertion Sort
3. Reverse Sorted Array – Merge Sort
4. Partially Sorted Array – Insertion Sort and Merge Sort

Note: These deductions are made assuming large size of input array (n>>0).

Appendix

As the machine used for implementing these algorithms was a personal computer, here are its basic details for deeper understanding:

- Model Name: Acer Nitro AN515-54
- Processor: Intel Core i5-9300H CPU @ 2.4 GHz 2.4 GHz
- RAM: 8 GB
- Operating System: 64-bit; x64-based processor

Given below are the 5 readings recorded for each algorithm for every scenario. For gaining better confidence on the above readings, the average of these readings was considered as the final value.

BUBBLE SORT												
Input Array Type	**Random Array**			**Sorted Array**			**Reverse Sorted Array**			**Almost Sorted Array**		
Array Size	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**
RUNTIME (ms)	684	2,518	5,646	301	733	1,859	558	1,888	5,318	315	1,257	3,144
	635	2,796	5,119	275	845	2,202	579	2,064	4,628	350	1,100	2,825
	596	2,075	5,515	369	833	1,556	668	1,631	5,634	381	1,241	3,536
	535	1,693	4,592	219	990	1,891	505	2,041	5,658	320	1,605	2,717
	728	2,464	5,801	363	809	1,864	577	2,470	4,663	483	1,141	3,205
Avg Runtime (ms)	**636**	**2,309**	**5,335**	**305**	**842**	**1,874**	**577**	**2,019**	**5,180**	**370**	**1,269**	**3,085**

INSERTION SORT												
Input Array Type	**Random Array**			**Sorted Array**			**Reverse Sorted**			**Almost Sorted**		
Array Size	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**
RUNTIME (ms)	46	218	468	-	-	-	108	400	952	15	77	170
	51	218	480	-	-	-	110	408	916	15	78	174
	62	203	479	-	-	-	112	420	934	16	79	174
	62	203	468	-	-	-	103	410	923	24	77	156
	62	207	486	-	-	-	109	412	922	31	79	171
Avg Runtime (ms)	**57**	**210**	**476**	**-**	**-**	**-**	**108**	**410**	**929**	**20**	**78**	**169**

SELECTION SORT												
Input Array Type	**Random Array**			**Sorted Array**			**Reverse Sorted**			**Almost Sorted**		
Array Size	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**
RUNTIME (ms)	81	335	727	78	327	737	91	359	833	84	341	728
	78	312	717	80	328	717	92	366	825	84	328	727
	93	335	735	81	323	732	91	370	824	81	314	730
	93	330	732	80	339	730	91	368	827	85	329	732
	81	330	736	80	331	743	94	374	859	83	325	732
Avg Runtime (ms)	**85**	**328**	**729**	**80**	**330**	**732**	**92**	**367**	**834**	**83**	**327**	**730**

MERGE SORT												
Input Array Type	**Random Array**			**Sorted Array**			**Reverse Sorted**			**Almost Sorted**		
Array Size	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**
RUNTIME (ms)	1	10	20	4	5	14	4	8	8	-	8	10
	5	11	21	2	1	1	-	3	5	-	9	14
	1	15	18	-	2	14	5	10	9	5	8	9
	7	6	21	-	-	1	-	1	10	-	9	10
	1	10	23	-	3	5	-	8	6	-	-	14
Avg Runtime (ms)	**3**	**10**	**21**	**1**	**2**	**7**	**2**	**6**	**8**	**1**	**7**	**11**

QUICK SORT												
Input Array Type	**Random Array**			**Sorted Array**			**Reverse Sorted Array**			**Almost Sorted Array**		
Array Size	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**	**10K**	**20K**	**30K**
RUNTIME (ms)	-	-	15	266	1,100	2,193	223	978	2,020	192	711	1,616
	-	8	10	284	1,008	2,321	296	802	1,949	216	467	1,383
	5	5	15	313	1,234	1,962	338	947	1,483	195	454	1,108
	-	9	17	276	1,040	2,381	294	1,025	2,247	174	512	1,284
	-	-	16	365	1,134	2,337	280	680	2,143	186	508	1,421
Avg Runtime (ms)	**1**	**4**	**15**	**301**	**1,103**	**2,239**	**286**	**886**	**1,968**	**193**	**530**	**1,362**

-----*xxx*-----