Experiment Report

Topic: Sort Detective

Authors: Hai Long Bi and Jingcheng Li

Date: 8th April 2017

Experimental Objective

Design, plan and document the set of tests, to deduce the right sort algorithm from all possible algorithms.

Experimental Principles

Analyse each of the sorting algorithms :

Sort	Best Complexity	Average Complexity	Worst Complexity	Stability	Special Conditions
Obv Bubble	(n^2)	(n^2)	(n^2)	no	
Bubble (early exit)	(n)	(n^2)	(n^2)	yes	O(n) when list is sorted
Insertion sort	(n)	(n^2)	(n^2)	yes	
Insertion sort binary search	(n)	n(logn)	(n^2)	yes	Difference in speed compared to normal insertion sort will be more evident in larger n
Selection sort	(n^2)	(n^2)	(n^2)	no	
Quadratic selection sort	---	(n^½ * n)	(n^2)	no	Faster than selection sort?
Merge sort	n(logn)	n(logn)	n(logn)	Depends on design	
Quick Sort	n(logn)	n(logn)	(n^2)	no	
Quick Sort Median of three	n(logn)	n(logn)	(n^2)	no	Faster than normal QS?
Randomised Quick Sort	n(logn)	n(logn)	(n^2)	no	
Shell sort Powers of two	n(logn)	---	n^(3/2)	no	

Shell sort (Sedgewick Intervals)	n(logn)	---	n^(4/3)	no	
Bogo sort	(n)	(n!)	Never sorted	no	

Experimental Method:

Using time

Compare time to complexity mathematically:

- Plot results on a graph
- Observe trends (such as quadratic or linear nature etc)
- Calculate mathematically the results (eg: doubling n = quadruple increase in time implies a n^2 relation)

Consistency:

- Doing multiple tests and confirm consistency in results

What can be inferred from the results of time:

- Using the assumed knowledge and our own knowledge, refine the possible sorting algorithms using their properties (such as stability or time complexity), eliminating possible options continually until we reached a result we may be certain the sorting algorithm is.

Checking different cases and special cases:

Special cases may result in different complexities, (eg QS on a sorted list = n^2); Test these cases:

- Sorted > some algorithms perform uniquely with a sorted array
- Reverse sorted > likewise, with sorted
- Random > this should give us an average complexity. Use same seed for randomly generated numbers on both sorts.

Determining stability:

To compare stability, both sorts were seeded the same random data as well as the unix sort. The unix sort is stable. Hence by comparing the outputs of both sorts to the unix sort, checking if there was no differences in files (diff command) and the ordering of the elements when sorted (look into the file and observe the ordering of the elements) can most likely determine the sorts are stable

Ensuring reasoning behind our assumptions, we have:

To overview and revise the algorithms to further ensure our knowledge on their performances and results

Experimental Data

The base unit of the first column is the number of elements, the base unit of the remaining columns are the time(seconds) taken for the sorting algorithm to sort the given amount of elements.

Random Sequence

	A - Time (R) 1	A - Time (R) 2	A - Time (R) 3	A - Time (R) Avg.
10,000	0.25	0.24	0.31	0.2667
20,000	1.12	1.16	1.06	1.1133
40,000	4.46	4.44	4.5	4.4667
50,000	7.06	7.11	7.13	7.1
80,000	18.67	18.3	18.28	18.4167
100,000	28.75	28.82	28.69	28.7533
160,000	73.78	74.03	73.65	73.82
200,000	115.29	115.62	114.89	115.2667
500,000	/	/	/	/
	B - Time (R) 1	**B - Time (R) 2**	**B - Time (R) 3**	**B - Time (R) Avg.**
10,000	0.01	0.01	0.01	0.01
20,000	0.02	0.02	0.01	0.0167
40,000	0.01	0.01	0.03	0.0167
50,000	0.04	0.01	0.02	0.0233
80,000	0.04	0.04	0.02	0.0333
100,000	0.04	0.04	0.03	0.0367
160,000	0.06	0.06	0.06	0.06
200,000	0.08	0.07	0.08	0.0767
500,000	0.01	0.01	0.01	0.01

Ascending Sequence

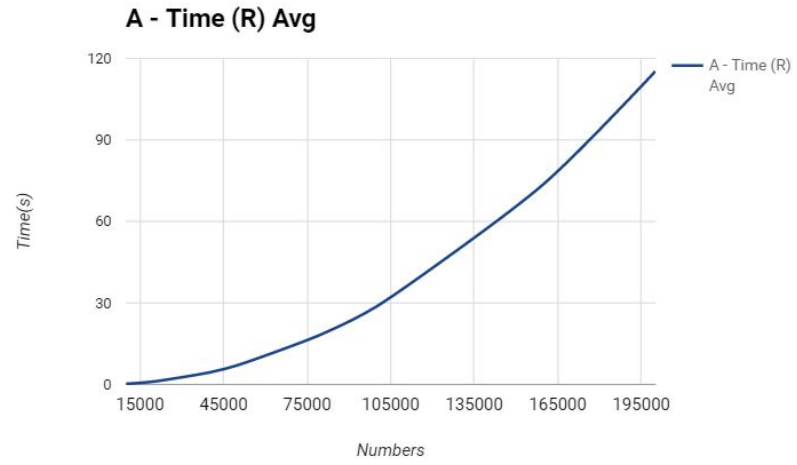
	A - Time (A) 1	A - Time (A) 2	A - Time (A) 3	A - Time (A) Avg.
10,000	0	0	0	0
20,000	0	0	0	0
40,000	0	0	0	0
50,000	0	0	0	0
80,000	0.01	0.01	0.02	0.0133

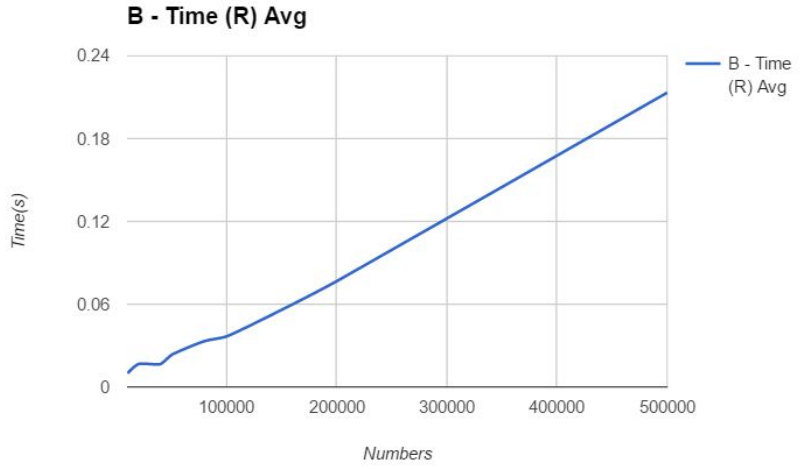
100,000	0.01	0.02	0.02	0.0167
160,000	0.01	0.02	0.02	0.0167
200,000	0.02	0.02	0.02	0.02
500,000	0.06	0.07	0.07	0.0667
	B - Time (A) 1	**B - Time (A) 2**	**B - Time (A) 3**	**B - Time (A) Avg.**
10,000	0	0	0	0
20,000	0	0	0	0
40,000	0	0	0	0
50,000	0	0	0.02	0.0067
80,000	0.02	0.02	0.01	0.0167
100,000	0.02	0.02	0.01	0.0167
160,000	0.03	0.02	0.02	0.0233
200,000	0.03	0.03	0.03	0.03
500,000	0.09	0.09	0.08	0.0867

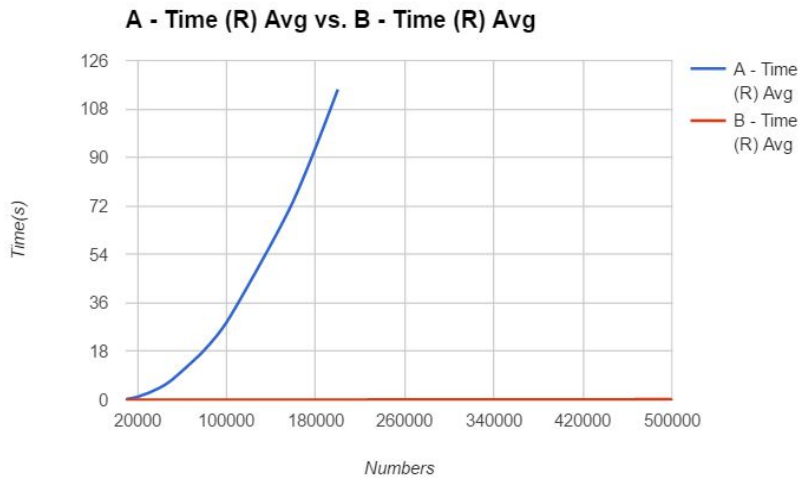
Descending Sequence

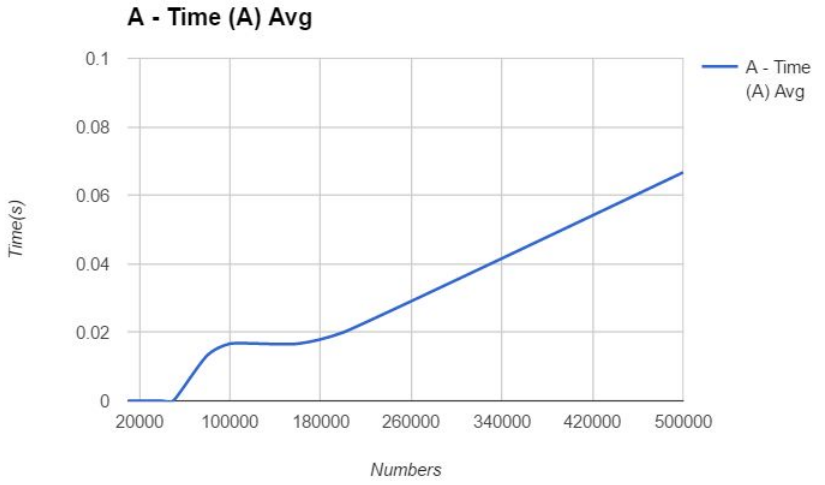
	A - Time (D) 1	**A - Time (D) 2**	**A - Time (D) 3**	**A - Time (D) Avg.**
10,000	0.34	0.23	0.23	0.2667
20,000	0.93	0.93	1.01	0.9567
40,000	3.74	3.71	3.71	3.72
50,000	5.82	5.82	5.84	5.8267
80,000	14.86	14.86	14.9	14.8733
100,000	23.35	23.17	23.43	23.3167
160,000	59.53	59.74	59.51	59.5933
200,000	93.2	93.27	93.62	93.3633
500,000	/	/	/	/
	B - Time (D) 1	**B - Time (D) 2**	**B - Time (D) 3**	**B - Time (D) Avg.**
10,000	0	0	0	0
20,000	0	0	0	0
40,000	0	0	0	0
50,000	0	0	0.02	0.0067
80,000	0.02	0.02	0.01	0.0167
100,000	0.02	0.02	0.01	0.0167
160,000	0.03	0.02	0.02	0.0233

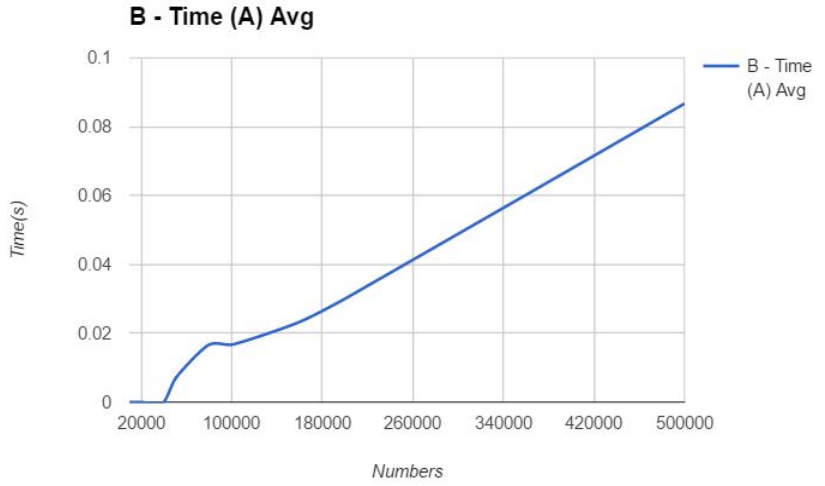
200,000	0.03	0.03	0.03	0.03
500,000	0.09	0.09	0.08	0.0867

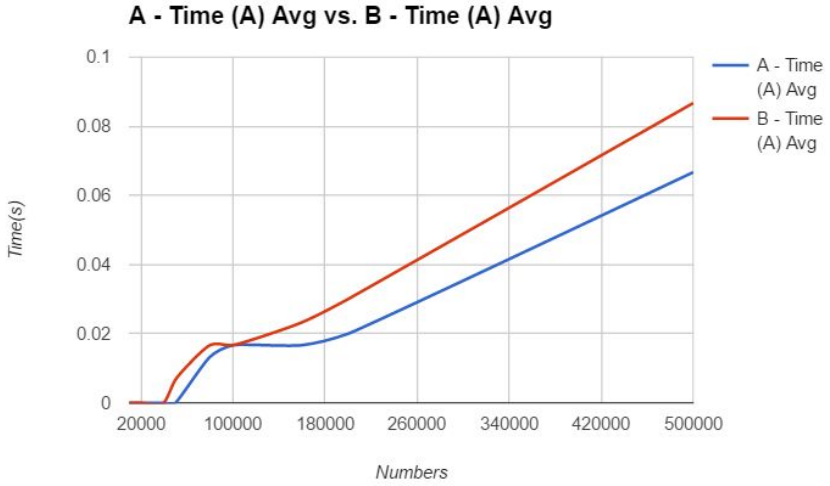

A - Time (R) Avg
120
90
60
30
0
Time(s)
15000
45000
75000
105000
135000
165000
195000
Numbers
A - Time (R) Avg

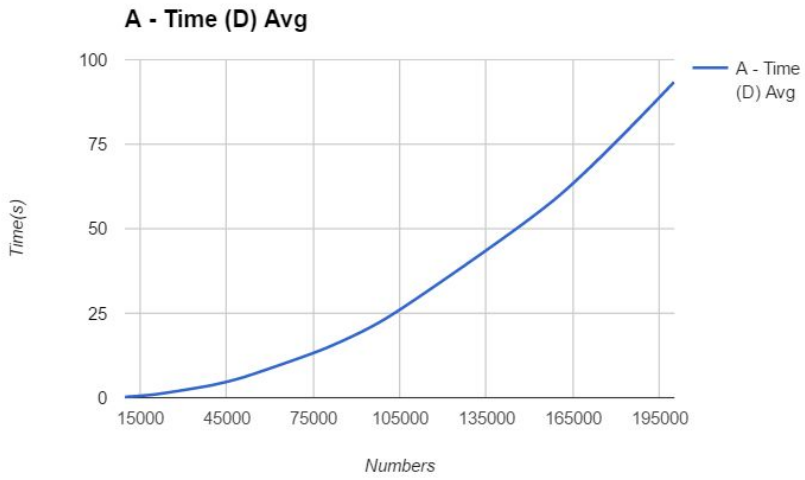

B - Time (R) Avg
0.24
0.18
0.12
0.06
0
Time(s)
100000
200000
300000
400000
500000
Numbers
B - Time (R) Avg

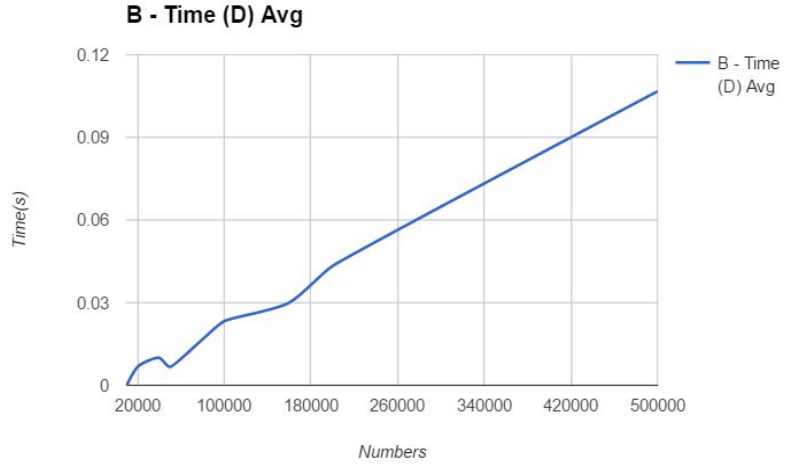

A - Time (R) Avg vs. B - Time (R) Avg
126
108
90
72
54
36
18
0
Time(s)
20000
100000
180000
260000
340000
420000
500000
Numbers
A - Time (R) Avg
B - Time (R) Avg

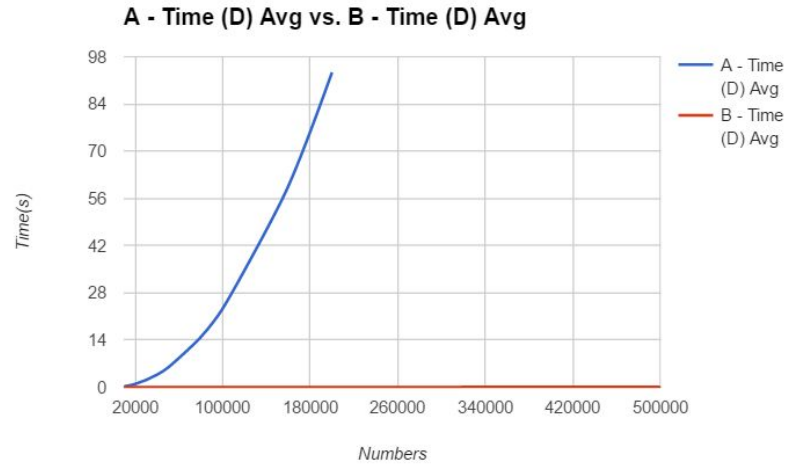

A - Time (A) Avg
0.1
0.08
0.06
0.04
0.02
0
Time(s)
20000
100000
180000
260000
340000
420000
500000
Numbers
A - Time (A) Avg

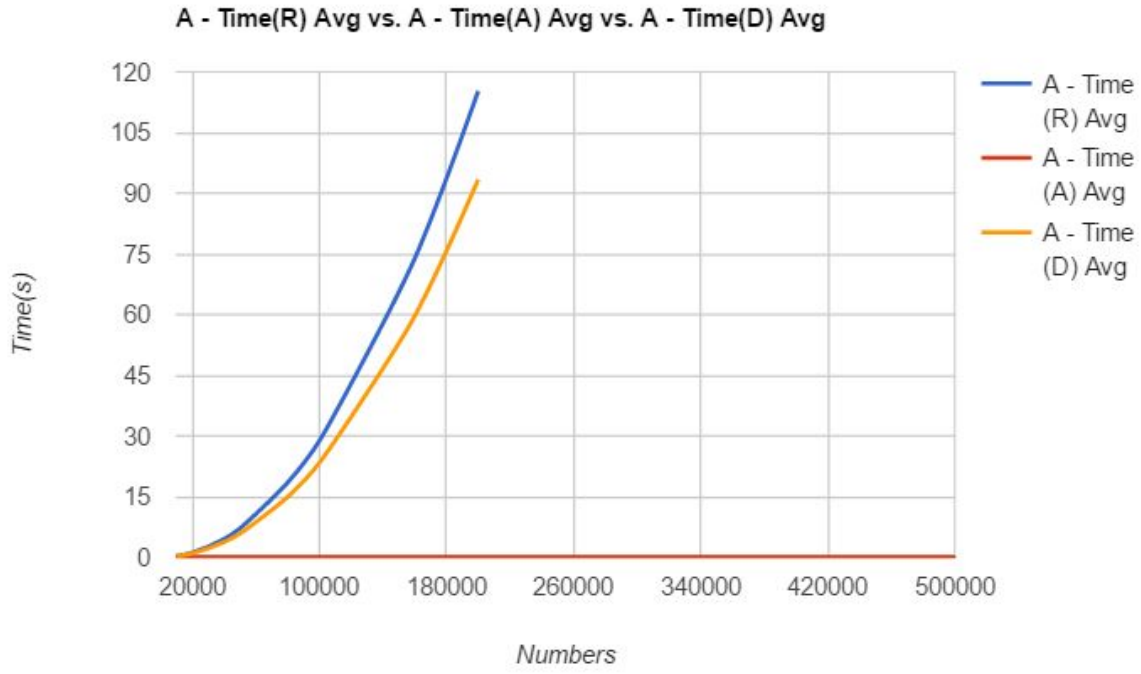

B - Time (A) Avg
0.1
0.08
0.06
0.04
0.02
0
Time(s)
20000
100000
180000
260000
340000
420000
500000
Numbers
B - Time (A) Avg

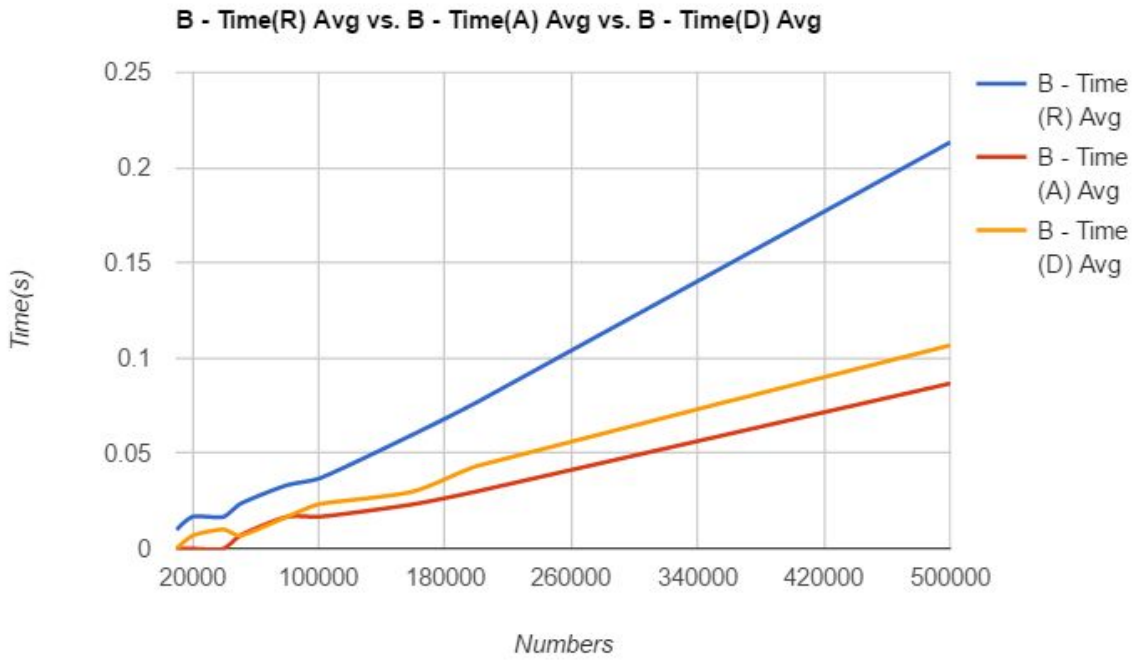

A - Time (A) Avg vs. B - Time (A) Avg
0.1
0.08
0.06
0.04
0.02
0
Time(s)
20000
100000
180000
260000
340000
420000
500000
Numbers
A - Time (A) Avg
B - Time (A) Avg


A - Time (D) Avg
100
75
50
25
0
Time(s)
15000
45000
75000
105000
135000
165000
195000
Numbers
A - Time (D) Avg


B - Time (D) Avg
0.12
0.09
0.06
0.03
0
Time(s)
20000
100000
180000
260000
340000
420000
500000
Numbers
B - Time (D) Avg


A - Time (D) Avg vs. B - Time (D) Avg
98
84
70
56
42
28
14
0
Time(s)
20000
100000
180000
260000
340000
420000
500000
Numbers
A - Time (D) Avg
B - Time (D) Avg


A - Time(R) Avg vs. A - Time(A) Avg vs. A - Time(D) Avg
120
105
90
75
60
45
30
15
0
Time(s)
20000
100000
180000
260000
340000
420000
500000
Numbers
A - Time (R) Avg
A - Time (A) Avg
A - Time (D) Avg


B - Time(R) Avg vs. B - Time(A) Avg vs. B - Time(D) Avg
0.25
0.2
0.15
0.1
0.05
0
Time(s)
20000
100000
180000
260000
340000
420000
500000
Numbers
B - Time (R) Avg
B - Time (A) Avg
B - Time (D) Avg

Experimental Conclusions:

Stability
By comparing the outputs of the two sorts, we found that both of them were stable.
This means there are 3 possible sorting algorithms that match the complexities determined from the graphs and results.

Hypotheses
Sort A could be insertion sort (binary search or normal), or bubble sort with early exit

- Insertion sort is stable because as it iterates in only an ascending order, it swaps at a defined point (first unsorted array element). The list items retain their ordering after swaps.
- Bubble sort is stable if it is implemented in a way so it does not swap equal elements (only >), as a result the list retains the order as the largest value reaches its appropiate position.

Sort B could be a stable merge sort or quick sort

Determining the sorts
Sort A is most likely **insertion sort:**

- The time complexity from **descending and random** items is quadratic, demonstrating $O(n^2)$ nature and average case complexity (double the elements = quadruple the time taken)

- Already sorted items **(ascending)** takes a very small amount of time, with complexity O(n) (double the elements = double the time taken, and the time taken was very small). Bubble sort with early exit and insertion sort are both sorts that have this property.

- Bubble sort and insertion sort have very similar complexities, stability and properties when sorting already sorted items. In order to determine the sort to be insertion sort instead of bubble sort, it can be inferred from the time taken to sort descending items. The time taken to sort descending items is still quadratic ($O(n^2)$), but was 20% faster than randomised.
    - The 20% is a noticeable amount present through all the various amounts of elements. For example, at 500000 elements, it on average takes 25~ seconds less.
    - This could be because sort A is performing less swaps.
        - Bubble sort would have n^2 swaps it compares each item as it passes, swapping at every comparison. This could increase sorting time.
        - Whereas insertion sort would swap only once as it finds the position for the element to be, and hence would have n swaps

        - That means insertion sort would be faster than bubble sort in the descending case.
    - Since it is faster on the descending case, but not overall (there is a difference from random and descending), it may have insertion sort with binary search, but that cannot be certainly evaluated.

- Sort A is most likely **insertion sort**

Sort B is a most likely a **stable merge sort:**

- The linear relationship was present across all cases (**ascending, descending, random)**. Doubling the amount of elements meant double the time taken. See below.

- The results of time may be too insignificant to determine what complexity the algorithm is.
    - The increases in time in relation to the increase in elements is **linear**, which could be O(n), O(n(logn)) or some variation. However there are no O(n) algorithms that could possibly match all cases, so an assumption is made that the complexity is a linear O(n(logn)).
    - A general statement can be made by the results of time. While it is not wise to base conclusions on the absolute speed of the results (due to the many various optimisation factors), the times were slower than what is believed to be the best O(n) case of Sort A by a notable amount (4x slower). This further supports that the algorithm is not O(n)

- The sort was also stable. Sorts with a similar complexity (quicksort and shell sort) are not stable. Merge sort can be stable or not, depending on implementation.

- This could mean this is a **stable merge sort.**