Sort detective report for week5&6 lab

In this lab, we are going to identify two different sort algorithms without looking at the source code. To achieve this goal, we design an experiment.

Experimental design

1. Do a research of all different kind of sorts that may happen, make a table of all related figures (complexity (best, worst case), stability, adaptively)
2. Base on the tables, starting test the sort A and sort B, using the same seed, comparing different time consuming when running with 5000 and 10000,20000,40000, 80000, 160000,320000,640000 numbers in random, reverse and ascending order. (to get the proper period to be compared)
3. Repeat the second step at ten times and then draw the graph based on the result we get in second step (best case and worst case), analyze the graph, then compared it with the table we make in the first step
4. Get the result, analyze if there is any mistake that may happen during the experiment process and discuss if the mistake can be avoided, give a clear final summary with reasons of the whole experiment.

Milestone 1 ---- Things need to know before tests

After doing all the research for different kind of sorting algorithms, we made a table to clearly compare them. Some more specific details about how they work are included in Appendix 1.

Time complexity for different sorting algorithms					
	Best Case	Average Case	Worse Case	stable	adaptive
Oblivious Bubble sort	N^2	N^2	N^2	No	No
Bubble Sort With Early Exit	N	N^2	N^2	Yes	Yes
Vanilla Insertion Sort	N	N^2	N^2	Yes	Yes
Insertion Sort with Binary Search	N*log(n)	N^2	N^2	Yes	Yes
Vanilla Selection Sort	N^2	N^2	N^2	No	No
Quadratic Selection Sort	N^(3/2)	N^(3/2)	N^(3/2)	Yes	No
Merge Sort	N*log(n)	N*log(n)	N*log(n)	Yes	No
Vanilla Quick Sort	N*log(n)	N*log(n)	N^2	No	Yes
Quick Sort Median of Three	N*log(n)	N*log(n)	N*log(n)	No	Yes
Randomised Quick Sort	N*log(n)	N*log(n)	N^2	No	No
Shell Sort Powers of Two	N*log(n)	N^(3/2)	N^2	No	Yes
Shell Sort Sedgewick	N*log(n)		N^(4/3)	No	Yes
Bogo Sort	n	N*n!	unbounded	No	Yes

Milestone 2&3 ---- Start testing

Things that should be concern about when testing

1. Compare the different performance of different kind of order, for example, random, reverse and ascending. By doing this, we can preliminary estimate whether the sort algorithm is adaptive.
2. Use the same seed while testing two different sort algorithms, especially for random order lists. If not, different random list may affect the performance of sort. Depend on different operation during sorting, using same seed for two sort algorithms will improve the accuracy of the result.

3. When testing for different size of numbers, the gap between sizes better be the older size multiple by 2. By doing this, when finish doing the tests and drawing graphs, will be easier to figure out if the complexity is parabola or linear.
4. To improve the accuracy of result, and because of the way timing works on Unix/Linux, it is necessary to repeat the whole experiment for several times, for this experiment, we decided to repeat it for 10 times to get a better result.
5. Avoid claiming too much accuracy, don't claim significant digits more than two, which is meaningless.
6. To ensure sortA and sortB are working correctly, we can compare the output using 'diff' commend.
7. To check the stability, generate a list of characters, starting with duplicate numbers (because the sorts are sorting numerically, numbers in the beginning are necessary as a sorting key), after sorting, check the letters after the numbers in the output file, if it is in the same order as the input file, then it is stable, vice versa.

Milestone 4—Get results

For SortA,
we observe that

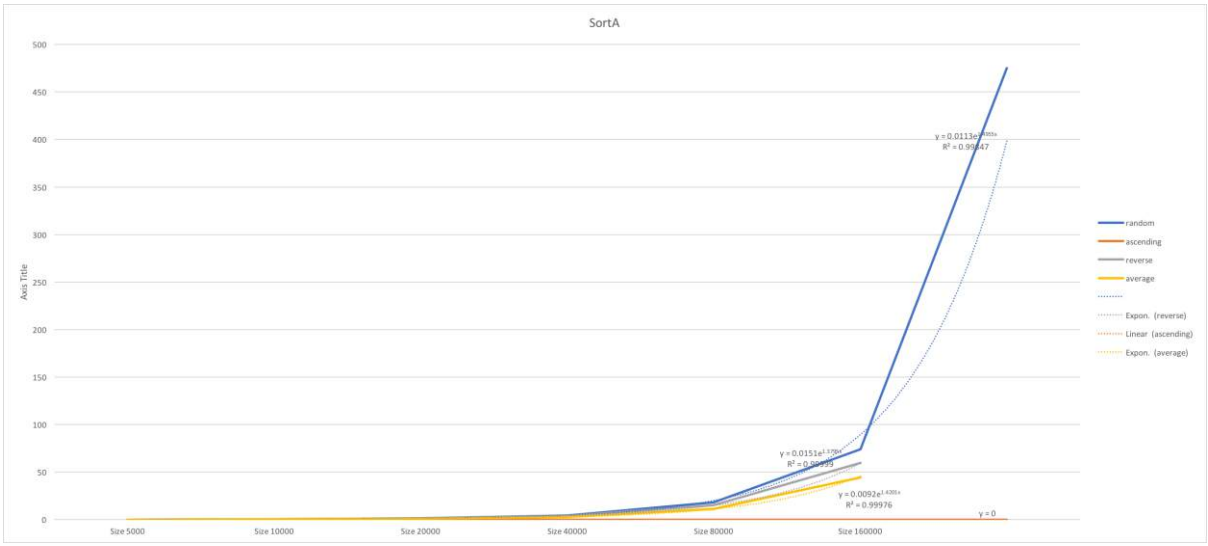
1. Adaptive
   The time for sorting ascending order is significantly quicker than random order, followed by descending order. This suggests that the algorithm is adaptive.
2. Stable
   We try to test the satiability by comparing the output with input, to see if the character after the numbers is in the same order as the input, these are the performance of sortA.

Input:	output:
4 qwer	1 psmg
3 nvod	1 xcno
4 aofm	1 qnvj
1 psmg	3 nvod
9 qivb	4 qwer
5 qovn	4 aofm
1 xcno	4 nidg
6 ango	5 qovn

4 nidg	6 ango
1 qnvj	6 pwmv
6 pwmv	9 qivb

as shown above, in output file, the character after numbers is the same as the input file, hence it is stable.

3.



From the complexity shown in the graph, when sorting the reverse or random order lists, the time complexity is most related to O(n^2), when sorting ascending order lists, the time complexity is nearly O(nlogn).

Hence by the above properties, the algorithm may be Insertion sort with binary search. However, since the complexity for ascending order is also suitable for O(n) but not as good as O(nlogn), it can also be Bubble sort with early exit or vanilla insertion sort.

For SortB:

1. Adaptive

   The time for sorting ascending order is significantly quicker than random order, followed by descending order. This suggests that the algorithm is adaptive.

2. Stable

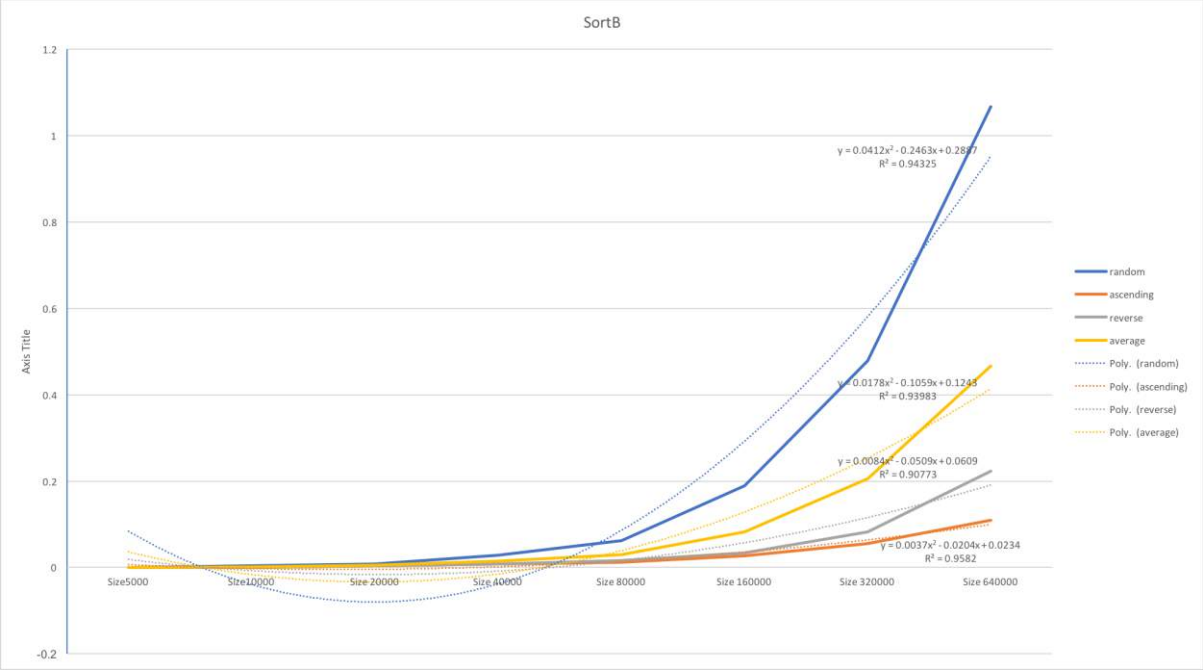
   We try to test the satiability by comparing the output with input, to see if the character after the numbers is in the same order as the input, these are the performance of sortB.

Input:	output:

4 qwer	1 qnvj
3 nvod	1 xcno
4 aofm	1 psmg
1 psmg	3 nvod
9 qivb	4 qwer
5 qovn	4 nidg
1 xcno	4 aofm
6 ango	5 qovn
4 nidg	6 ango
1 qnvj	6 pwmv
6 pwmv	9 qivb

From the output file above, we can clearly see that the order after numbers are not the same as input file, hence it is not stable.

3. complexity



From the graph, the time complexity for random order as well as reverse order is nearly O(n^2), the complexity for ascending order is nearly O(nlogn). However, the time complexity for average is between O(nlogn) and O(n^2); which is O(n^(3/2)).

Hence, by the properties above, SortB is most likely to be Shell sort powers of two. However, Vanilla quick sort could be SortB as well because the performance of these two algorithms are nearly the same.

Summary:

Error estimate:

Error may happen when plotting the graphs, since we don't know the complexity for each line, and the provided trade line may not accurate as expected, hence there may have some error during plotting. However, it cannot be avoided due to the limited of time

Result:

For sortA, it might be Insertion sort with binary search, however it also may be Bubble sort with early exit or vanilla insertion sort, cause the time complexity for ascending order also suit for O(n), but it is more likely to be Insertion sort with binary search after the properties showed above.

For sortB, it is most likely to be Shell sort powers of two. However, Vanilla quick sort could be SortB as well because the performance of these two algorithms are nearly the same.

Appendix1:

Introduction to different sorting algorithms

Oblivious bubble sort

The simplest bubble sort algorithm, without any optimizations. Without regarding if there are any swaps has already made in each pass, hence is not adaptive.

Bubble Sort with early exit

Implement compared with oblivious, swaps only happens if the current element is strictly greater than the neighbour element.

Vanilla insertion sort

In the vanilla insertion sort, it has a copy of the array. If the list is in ascending order, it will just perform as a linear search, however if it is in reverse order, it will perform as bad case which should compare and swap each time.

Insertion Sort with binary search

Insertion sort with binary search uses binary search instead of linear search when finding the insertion point. So in the best case just need log(n) complexity to traverse, with a constant cost for insertion n, the overall complexity will be n*log(n) for the best case and n^2 for the bad case.

Vanilla Selection sort

it will select the smallest item and swap it to correct position each time, regardless of the input, hence it is quadratic in each pass, it is also unstable as the swapping disturbs order.

Quadratic selection sort

It breaks the initial array into sqrt(n) subarrays, then find the minimum value from each subarray and put in to a final array, then keep finding the minimum item than write in another array, and keep finding until it sorted, regardless of the input, but stable.

Merge sort
Merge sort splits the array into two equal sized partitions and each recursion of the partition is further split into two equal sized arrays. In other words, merge sort is like the binary search of a sorting algorithm. As a result, the complexity is of O(NlogN). As a result, merge sort is unadaptive but stable.

Vanilla quick sort
Since this implementation of quicksort chooses the last element to be the pivot point, the best case scenario is O(NlogN) where the pivot point gives two equally sized partitions. Worst case scenario being that the highest/lowest point is chosen which would result in $O(n^2)$ complexity. Since the last element is always chosen to be the pivot point, this causes already sorted lists and reverse sorted lists to have $O(n^2)$.

Quick sort median of three
Median of three quicksort is one of the 3 possible options of choosing a pivot point to reduce the incidence a worst case scenario for quicksort. The other two choices are picking a random pivot point and picking the middle point. Median of three refers to the median value between the first middle and last element as the pivot point. The normal number of comparisons needed when choosing a random pivot is higher than the number of comparisons needed when using the median of three as the pivot. However, there is a three percent increase in the number of swaps. Either way, the complexity is O(NlogN).

Randomised quick sort
Shuffling the list before implementing vanilla quicksort is the same as choosing a random point as the pivot. The complexity is therefore O(NlogN) as stated above. Randomized quicksort should have a fairly noticeable variation in running time over multiple runs on the same input as the sequence the input gets randomized to initially heavily affects its time complexity (assuming that the randomizer doesn't use the sequence as a seed).

Shell sort powers of two
Select the gap sequence as power of two, based on insertion sort, sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared.

Shell sort Sedgewick
Shell sort with the general term $4^k + 3 * 2^{k-1} + 1$ where K is >= 1. Sedgewick suggested using lowest common divisors or numbers that are pairwise coprime. As a result, this lowers the complexity to $O(Nlog^2N)$.

Bogo sort
This algorithm randomly chooses two elements and swaps them if they are out of order, repeating this process until the list is sorted. This process makes the algorithm inherently unstable as there is no regard to the initial relative ordering of identical elements. Since there is no guarantee that all pairs of out of order elements will be chosen in the correct sequence at some point in time (since pairs are chosen randomly), the complexity is $O(\infty)$ for all inputs except sorted input, as sorted input

terminates immediately with no swaps taking place. As the complexity for checking sortedness of a list is O(n), this makes it the best-case scenario.

Appendix 2: Time for different size of input using SortA

Time for SORT**A**(sec)											
times	1	2	3	4	5	6	7	8	9	10	average
Size 5000											
random	0.136	0.064	0.056	0.056	0.056	0.056	0.056	0.056	0.056	0.056	0.047
ascending	0	0	0	0	0	0	0	0	0	0	0
reverse	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06
Size 10000											
random	0.248	0.252	0.248	0.268	0.248	0.244	0.244	0.248	0.244	0.244	0.25
ascending	0	0	0	0	0	0	0	0	0	0	0
reverse	0.240	0.240	0.232	0.232	0.238	0.236	0.232	0.232	0.248	0.244	0.24
Size 20000											
random	1.08	1.128	1.120	1.072	1.06	1.056	1.06	1.076	1.06	1.056	1.0768
ascending	0.04	0	0	0	0	0	0	0	0	0	0.004
reverse	0.968	0.936	0.932	0.932	0.932	0.932	0.932	0.932	0.932	0.932	0.936
Size 40000											
random	4.452	4.444	4.456	4.456	4.448	4.724	4.508	4.452	4.440	4.728	4.5108
ascending	0.008	0.004	0.004	0.004	0	0.004	0.004	0.004	0.004	0.004	0.004
reverse	3.732	3.716	3.716	3.716	3.716	3.708	3.712	3.728	3.720	3.968	3.7432
Size 80000											
random	18.388	18.416	18.292	18.380	18.284	18.296	18.276	18.308	18.276	18.240	18.3156
ascending	0.008	0.008	0.012	0.008	0.012	0.012	0.012	0.012	0.008	0.012	0.0104
reverse	14.884	14.892	14.896	14.896	14.908	14.860	14.916	14.832	14.928	15.018	0.016

Size 160000											
random	73.716	73.732	73.772	73.860	73.788	73.732	73.836	73.884	73.800	73.836	73.7956
ascending	0.024	0.024	0.020	0.016	0.020	0.020	0.020	0.020	0.020	0.020	0.0204
reverse	59.660	59.548	59.720	59.592	59.480	59.500	59.544	59.692	59.792	59.764	59.6292
Size 320000											
random	475.060	475.796	475.385	475.274	475.294	475.350	475.294	475.028	475.294	475.274	475.3049

Appendix 2: Time for different size of output using SortB

Time for SORT**B**(sec)											
times	1	2	3	4	5	6	7	8	9	10	average
Size5000											
random	0.008	0	0	0	0	0	0	0	0	0	0
ascending	0	0	0	0	0	0	0	0	0	0	0
reverse	0	0	0	0	0	0	0	0	0	0	0
Size10000											
random	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004
ascending	0	0	0	0	0	0	0	0	0	0	0
reverse	0	0	0	0	0	0	0	0	0	0	0
Size 20000											
random	0.012	0.008	0.008	0.008	0.008	0.008	0.008	0.008	0.008	0.008	0.0084
ascending	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004
reverse	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004
Size 40000											
random	0.028	0.028	0.028	0.028	0.028	0.028	0.028	0.028	0.028	0.028	0.028
ascending	0.004	0.008	0.008	0.008	0.008	0.008	0.008	0.008	0.008	0.008	0.008
reverse	0.008	0.008	0.008	0.008	0.008	0.008	0.008	0.008	0.008	0.008	0.008
Size 80000											
random	0.06	0.068	0.064	0.064	0.06	0.064	0.06	0.06	0.06	0.06	0.062
ascending	0.016	0.012	0.012	0.012	0.012	0.012	0.012	0.012	0.012	0.012	0.0124
reverse	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016	0.016
Size 160000											
random	0.192	0.188	0.188	0.188	0.192	0.188	0.188	0.188	0.188	0.188	0.188
ascending	0.024	0.020	0.028	0.024	0.028	0.028	0.028	0.028	0.028	0.028	0.0264
reverse	0.036	0.032	0.032	0.036	0.036	0.032	0.032	0.036	0.032	0.032	0.0336

Size 320000											
random	0.508	0.476	0.476	0.476	0.476	0.476	0.476	0.476	0.476	0.476	0.4792
ascending	0.052	0.056	0.056	0.052	0.056	0.056	0.056	0.056	0.056	0.056	0.0552
reverse	0.080	0.084	0.084	0.080	0.084	0.084	0.080	0.084	0.084	0.080	0.0824
Size 640000											
random	1.076	1.072	1.064	1.064	1.056	1.072	1.060	1.072	1.068	1.072	1.0676
ascending	0.12	0.104	0.108	0.108	0.108	0.112	0.112	0.112	0.104	0.108	0.1096
reverse	0.224	0.224	0.228	0.228	0.224	0.224	0.228	0.216	0.220	0.212	0.2228