COMP2521 Sort Detective Lab Report

by Jinghan Wang (z5286124)

In this lab, the aim is to measure the performance of two sorting programs, without access to the code, and determine which sorting algorithm each program uses.

Experimental Design

To find out what sort the two programs are, I use the control variable method to solve. There are three programs in this experiment. First, Since the three orders of input data generated by the gen program are the variables of the project, I checked whether the data generated by the gen program matched the conditions I need. In addition, I check the output of program A and program B under the same input to ensure the validity of the program. These checks are used to ensure that everything except variables is working properly.

Due to less data, it is impossible to compare the time difference between each program in different sorts, I measured how each program's (Program A and Program B) execution time required by providing a large quality of data in ascending, descending and random conditions.

I provide each program 10,000, 20,000, 40,000, 80,000, 160,000 numbers of data as input in sorted, reversed and random conditions, and because of the way timing works on Unix/Linux, I repeat the same test five times to reduce the misjudgment of the results caused by error.

Experimental Results

Program A

For Program A, I observed when the same amount of data is entered, the time required by the sorted input is close to or equal to 0s. The time required by the input which is arranged in random order is approximately twice that required by the input in reverse order.

In addition, for the size of an adjacent input, the size of first is twice as long as the second, and the time required for the first is approximately twice as long as the second, except for ascending input.

In my opinion, For bubble sort, when in ascending order, it's the best-case performance. The program iterates through all the elements, and it takes less time. When in reverse order, it is the worst-case performance for bubbling sort, the first cycle, the elements exchange (n – 1) times, the second loop, the elements exchange (n – 1) times, until the last cycle, the elements exchange 1 times. Therefore, the program needs to exchange

$\frac{n\,(n-1)}{2}$ times. The average performance of bubble sort is a half time of ascending order and reverse order. When n becomes twice, the average time required is $\frac{2n\,(2n-1)}{2}$, which is $\frac{4n-2}{n-1}$ times of the processing time when size is n. When n approaches infinity, this value approaches 4. The other two cases are the same, but because the ascending speed is too fast, they are not reflected in this experiment. Therefore, when size increases by twice, the time required increases by four. Program A fits the characteristics of Bubble sort.

For Insertion sort, the character is similar to bubble sort. When in reverse order, during the first loop, insert 1 time, during the second loop, 2 times… During the last element insert (n - 1) times. Therefore, the program needs to insert $\frac{n\,(n-1)}{2}$ times. The character of Insertion sort is similar to Bubble sort. The average performance of bubble sort is also a half time of ascending order and reverse order. When size increases by twice, the time required also increases by four. Program A fits the characteristics of Insertion sort.

These observations indicate that the algorithm underlying the program is Insertion sort and Bubble sort which has the ascending order time is extremely short, the random order time is approximately half of reverse order time. Time is a perfect square of the growth of size.

Program B

For Program B, I observed that when the input data is in random order, the program takes extremely short time, but when the input data is in ascending and reverse order, it takes a lot of time. What is more, the time required for ascending and reverse order is close, and the ascending time is slightly less than the reverse time.

In my opinion, the character of program B is suitable for Naive quicksort. Because the worst case of quicksort is that the initial sequence has been ordered. When the pivot is the leftmost element, for ascending order, the first sequence is still fixed in the original position after (n – 1) comparison, and a subsequence of length (n - 1) is obtained. In the second sorting, after (n - 2) times of comparison, the second element is determined in its original position, and then a subsequence of length (n - 2) is obtained. And so on, the final total number of comparisons is $\frac{n(n-1)}{2}$. For descending order, it is similar to ascending order, but as all of the right elements are smaller than pivot, it takes a little time to exchange the elements to the left of the pivot, therefore, it will take more time than ascending order.

These observations indicate that the algorithm underlying the program is Naive quicksort which has the worst-case performance is O (n^2) when the input data is in sorted order and the average performance is O (n*log n) when input data is in random order.

Conclusions

On the basis of our experiments and our analysis above, I believe that

- sortA implements the Bubble sort or Insertion sort algorithm
- sortB implements the Naive quicksort algorithm

Appendix

Program A

Testing for input, size 10,000

time	random	sorted	reversed
1	0.09	0.00	0.16
2	0.08	0.00	0.17
3	0.10	0.00	0.17
4	0.08	0.00	0.17
5	0.08	0.00	0.20
Average	**0.09**	**0.00**	**0.17**

Testing for input, size 20,000

time	random	sorted	reversed
1	0.33	0.00	0.68
2	0.33	0.00	0.73
3	0.34	0.00	0.69
4	0.32	0.00	0.68
5	0.32	0.00	0.68
Average	**0.33**	**0.00**	**0.69**

Testing for input, size 40,000

time	random	sorted	reversed
1	1.32	0.00	2.77
2	1.36	0.00	2.72
3	1.64	0.00	2.83
4	1.40	0.00	2.92
5	1.57	0.00	2.80
Average	**1.46**	**0.00**	**2.81**

Testing for input, size 80,000

time	random	sorted	reversed
1	5.33	0.00	11.30
2	5.57	0.00	11.10
3	5.22	0.00	11.11
4	5.30	0.00	11.40
5	5.36	0.00	11.06
Average	**5.36**	**0.00**	**11.19**

Testing for input, size 160,000

time	random	sorted	reversed
1	21.35	0.01	44.20
2	21.27	0.02	44.98
3	21.31	0.02	45.08
4	21.24	0.01	44.04
5	21.24	0.02	44.01
Average	**21.28**	**0.02**	**44.46**

Program B

Testing for input, size 10,000

time	random	sorted	reversed
1	0.00	0.12	0.13
2	0.00	0.12	0.13
3	0.00	0.12	0.13
4	0.00	0.12	0.13
5	0.00	0.12	0.13
Average	**0.00**	**0.12**	**0.13**

Testing for input, size 20,000

time	random	sorted	reversed
1	0.00	0.50	0.54
2	0.00	0.50	0.54
3	0.00	0.50	0.55
4	0.00	0.51	0.52
5	0.00	0.50	0.53
Average	**0.00**	**0.50**	**0.54**

Testing for input, size 40,000

time	random	sorted	reversed
1	0.01	2.02	2.16
2	0.00	2.03	2.17
3	0.01	2.05	2.14
4	0.01	2.06	2.17
5	0.01	2.02	2.16
Average	**0.01**	**2.04**	**2.16**

Testing for input, size 80,000

time	random	sorted	reversed
1	0.02	8.09	8.71
2	0.02	8.09	8.64
3	0.02	8.10	8.68
4	0.02	8.06	8.69
5	0.02	8.12	8.66
Average	**0.02**	**8.09**	**8.68**

Testing for input, size 160,000

time	random	sorted	reversed
1	0.04	32.49	34.76
2	0.04	32.44	34.81
3	0.04	32.40	34.84
4	0.04	32.60	34.66
5	0.05	32.70	34.67
Average	**0.04**	**32.53**	**34.75**