COMP2521 Sort Detective Lab Report

by Haoheng Duan Z5248147

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

Experimental Design

There are two aspects to my analysis:

1. determine that the sort programs are actually correct
2. measure their performance over a range of inputs

Correctness Analysis

To determine correctness, I chose data with different key numbers which are out of order because it will show whether the sort algorithms work correctly by checking the order after sorting.

Performance Analysis

In my performance analysis, I measured how each program's execution time varied as the size and initial sortedness of the input varied. I chose different size of data, from small to large, to test the time complexity of the sorting algorithms. I chose 10000, 50000, 100000, 150000 and 200000 as sizes for these programs because these sizes are large enough to show the patterns of the algorithm.

Because of the way timing works on Unix/Linux, it was necessary to repeat the same test multiple times and take the average time to be the performance.

I was able to use up to quite large test cases without storage overhead because (a) I had a data generator that could generate consistent inputs to be used for multiple test runs, (b) I had already demonstrated that the program worked correctly, so there was no need to check the output.

I also investigated the stability of the sorting programs by chosing some data with duplicate key numbers and different words following it.

Experimental Results

Correctness Experiments

An example of a test case and the results of that test is in appendix.

On all of my test cases, the A program and B program sorted data correctly by keeping the key numbers in order.

Performance Experiments

For Program A, we observed that it didn't change the sequence of the word following the same keys, which means that it's stable. It spents 0.35s when the input size is 10000, 9.8s when the input size is 50000, 39.96s when the input size is 100000, 90.45s when the input size is 150000, and 161.35s when the input size is 200000.

These observations indicate that the algorithm underlying the program A has the following characteristics. It's stable and its time complexity is $O(n^2)$. From the description in the requirement, it is clear that it can be implemented by using Bubble Sort With Early Exit algorithm.

For Program B, we observed that it changed the sequence of the word following the same keys, which means that it's not stable. It spents 0 s when the input size is 10000, 0.02s when the input size is 50000, 0.04s when the input size is 100000, 0.06s when the input size is 150000, and 0.09s when the input size is 200000.

These observations indicate that the algorithm underlying the program B has the following characteristics. It's unstable and its time complexity is $O(nlogn)$. From the description in the requirement, it is clear that it can be implemented by using Quick Sort Median of Three algorithm.

Conclusions

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

1. ProgramA implements the Bubble Sort With Early Exit sorting algorithm,
2. ProgramB implements the Quick Sort Median of Three sorting algorithm.

Appendix

MY DATA

91 guw

22 lor

71 oen

43 ljp

87 sgs

84 ord

96 pbt

57 cac

80 cgd

65 jxk

18 yne

32 whs

49 rrw

20 ggx

60 frk

19 cdy

5 arz

88 pqo

4 hcd

67 yxa

33 qmg

98 lns

55 aqx

14 ybl

63 kpq

53 qhf

64 pxr

17 rcb

36 jji

37 vsw

47 jma

13 xsj

69 hki

68 cbh

8 idd

25 apq

34 bbu

51 sbc

2 rbb

59 hzv

6 owk

85 tqk

61 mln

70 cqc

12 wfr

82 tac

40 xix

39 qtb

93 qxn

24 nmp

76 gpx

50 sof

56 wpq

86 vwc

90 boa

78 kuy

46 zqf

73 mfg

99 ade

92 nny

79 tdl

89 qms

42 rrb

62 ozj

16 fsa

7 kyh

27 hop

54 fbs

26 fwk

74 dwd

95 dgw

30 nwn

9 qsc

75 wfc

28 kmc

3 mqb

52 nuv

44 tns

31 kue

81 ewh

58 ehc

48 fad

94 zlg

35 qcl

45 nfw

100 ugu

72 dto

15 dbe

1 nwl

10 dxr

11 jmo

23 ell

97 rwb

77 iqv

83 ioh

21 xpk

29 oqh

66 itz

38 mdk

41 mvt

SORT A

1 nwl

2 rbb

3 mqb

4 hcd

5 arz

6 owk

7 kyh

8 idd

9 qsc

10 dxr

11 jmo

12 wfr

13 xsj

14 ybl

15 dbe

16 fsa

17 rcb

18 yne

19 cdy

20 ggx

21 xpk

22 lor

23 ell

24 nmp

25 apq

26 fwk

27 hop

28 kmc

29 oqh

30 nwn

31 kue

32 whs

33 qmg

34 bbu

35 qcl

36 jji

37 vsw

38 mdk

39 qtb

40 xix

41 mvt

42 rrb

43 ljp

44 tns

45 nfw

46 zqf

47 jma

48 fad

49 rrw

50 sof

51 sbc

52 nuv

53 qhf

54 fbs

55 aqx

56 wpq

57 cac

58 ehc

59 hzv

60 frk

61 mln

62 ozj

63 kpq

64 pxr

65 jxk

66 itz

67 yxa

68 cbh

69 hki

70 cqc

71 oen

72 dto

73 mfg

74 dwd

75 wfc

76 gpx

77 iqv

78 kuy

79 tdl

80 cgd

81 ewh

82 tac

83 ioh

84 ord

85 tqk

86 vwc

87 sgs

88 pqo

89 qms

90 boa

91 guw

92 nny

93 qxn

94 zlg

95 dgw

96 pbt

97 rwb

98 lns

99 ade

100 ugu

SORT B

1 nwl

2 rbb

3 mqb

4 hcd

5 arz

6 owk

7 kyh

8 idd

9 qsc

10 dxr

11 jmo

12 wfr

13 xsj

14 ybl

15 dbe

16 fsa

17 rcb

18 yne

19 cdy

20 ggx

21 xpk

22 lor

23 ell

24 nmp

25 apq

26 fwk

27 hop

28 kmc

29 oqh

30 nwn

31 kue

32 whs

33 qmg

34 bbu

35 qcl

36 jji

37 vsw

38 mdk

39 qtb

40 xix

41 mvt

42 rrb

43 ljp

44 tns

45 nfw

46 zqf

47 jma

48 fad

49 rrw

50 sof

51 sbc

52 nuv

53 qhf

54 fbs

55 aqx

56 wpq

57 cac

58 ehc

59 hzv

60 frk

61 mln

62 ozj

63 kpq

64 pxr

65 jxk

66 itz

67 yxa

68 cbh

69 hki

70 cqc

71 oen

72 dto

73 mfg

74 dwd

75 wfc

76 gpx

77 iqv

78 kuy

79 tdl

80 cgd

81 ewh

82 tac

83 ioh

84 ord

85 tqk

86 vwc

87 sgs

88 pqo

89 qms

90 boa

91 guw

92 nny

93 qxn

94 zlg

95 dgw

96 pbt

97 rwb

98 lns

99 ade

100 ugu