Performance Testing Report of Two Sorting Codes

Zixuan Zhao

Oregon State Universy

1 Introduction

1.1 Purpose

This test report provides a summary of test project. And this project is created to learn the software testing and to test the random test in TSTL.

1.2 Project Info

This project tests two sorting dictionary python program. It mainly concerns on the compatibility of input, correctness test as well as the performance of these two codes. The compatibility of input tests how well program can accept the input. The correct test examines whether program can response correct answer. The performance test test from three aspects. On one hand, it tests how string size and dictionary size affect running time. On another hand, it compares these two codes how well they are as the input dictionary size increasing.

1.3 Test Object

The first test object is python-skiplist, written by sepeth. It can be found from https://github.com/sepeth/python-skiplist, and downloaded by "pip install python-skiplist". This is not a pure python code which is implemented by C.

The second test object is sortedcontainers, written by Grant Jenks. It can be found from https://pypi.python.org/pypi/sortedcontainers, and downloaded by "pip install sortedcontainers". This is a pure python code.

These two codes both contains the sorting program for dictionary. And in this project, I mainly test the sorting dictionary part in both codes.

1.4 Test Environment

This project is running on Mac Pro, with 2.7 GHz Intel Core i5 processor and 8GB 1867MHz DDR3 Memory. Python version is 2.7, which is default for OS system.

2 Test Project Design

2.1 Input Test

The input test concerns on integer and string type. A dictionary is a data structure with key-value pair. To satisfy all circumstances, I set four combinations for key-value pair in dictionary, which are

a) Key: Integer type; Value: Integer type
b) Key: Integer type; Value: String type
c) Key: String type; Value: Integer type
d) Key: String type; Value: String type

Specific for string type element, I created it by ASCII code so that I can test whether special code character can be sorted correctly. The input cases are generated randomly, which means these four cases in dictionary as well as the integer value and string value are generated randomly.

To achieve this goal, I create three functions randomly generating different strings and dictionary. And integer type value is created by random function in python in this case.

2.2 Correctness Test

Correctness test examines whether codes can sort dictionary correctly under various circumstances. Since two functions are sorting, when the result of two functions are the same, we can say they can sort dictionary correctly.

2.3 Performance Test

For performance test, I collected all data with different dictionary size and string size. And by comparing the duration in soring process, I will find whether pure python is better or not. What's more, I examine how dictionary size and string size affect the performance of these two sorting problem. These data will be statistically examined by R.

3 Test Summary

3.1 Data Info

There are mainly three important variables in this project, dictionary size, string size and integer value range. The integer value range is fixed in whole test which is from -1000 to 1000. The other two variables is various for certain test.

3.2 Input Test and Correctness Test

In this test, for convenient observation, the generated dictionaries are size 10 and the string elements are size 1. In this case, I can examine some output randomly.

From observation, I got these result which is same with my expectation:

a. Both codes can sort pure string dictionary correctly
b. Both codes can sort pure integer dictionary correctly
c. Both codes can sort dictionary with string and integer correctly

In this case, the correctness means all elements sorted by the sequence of Ascii code for the first character in key. If the first character is the same, then it will compare the next one in key.

3.3 Performance Test

This performance test related with two factors, the string size and dictionary size. In this test, the dictionary size is in collection of 10, 20, 30, 50, 100, 150 and 200, and string size is in collection of 1, 5, 10, 15, 20, 50, 100, 200, 300 and 400. Since integer size is fixed in the memory and I do not test the invalid integer in this project, I do not cover the integer.

3.3.1 Single Code Performance

First, for python-skiplist, here is the average of running time based on different string size and dictionary size (Table 1 and Table 2). From the produced scatter plot (Figure 1) of Table 1 and Table 2, we can see that with the increasing of dictionary size, the average running time is decreased. However, the average running time is increased as the string size being increased. The result of running time vs. dictionary size is not what I expected. I create boxplot (Figure 2) and I found that the outlier in result of of running time vs. dictionary size is serious. Except these outliers, we can see that the dictionary size does not affect the running time a lot. By comparing two boxplots in Figure 2, I estimate that the outliers for low size of dictionary from the data point with higher string size.

Table 1 Average Running Time vs Dictionary Size

	Dictionary Size	Average Running Time
1	1	9.30E-05
2	5	9.11E-05
3	10	9.22E-05
4	20	9.31E-05
5	50	8.52E-05
6	100	9.80E-05
7	150	9.18E-05
8	200	9.21E-05
9	300	8.21E-05
10	400	3.62E-05

Table 2 Average Running Time vs. String Size

	String Size	Average Running Time
1	10	1.42E-05
2	20	2.40E-05
3	30	3.52E-05
4	50	5.57E-05
5	100	1.14E-04
6	150	1.70E-04
7	200	2.26E-04

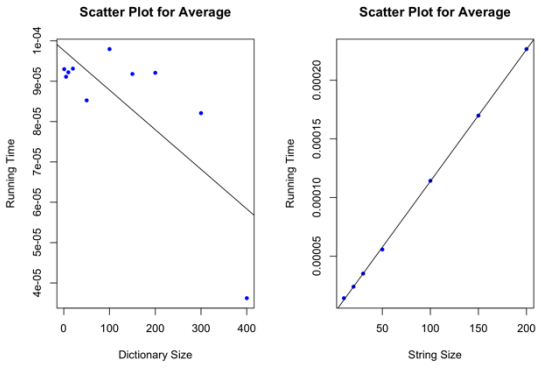

Figure 1

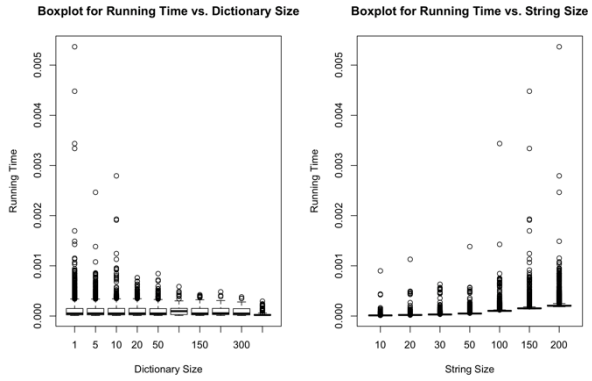

Figure 2

In this case, I create the multiple linear regression by this formula to see how these two factor affect the running time:

$$\mu\{RunningTime|StringSize, DictionarySize\}$$

The result is showed below.

```
Residuals:
       Min         1Q     Median         3Q        Max
-0.0000415 -0.0000124 -0.0000045 -0.0000009  0.0051388

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.903e-06  3.421e-07   5.563 2.67e-08 ***
ssize        1.122e-06  3.172e-09 353.798  < 2e-16 ***
dsize       -1.539e-08  4.021e-09  -3.827  0.00013 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 5.432e-05 on 65618 degrees of freedom
Multiple R-squared:  0.657,   Adjusted R-squared:  0.657
F-statistic: 6.285e+04 on 2 and 65618 DF,  p-value: < 2.2e-16
```

According to this result, we can say that string size and dictionary size significantly affect the running time.

Then, for sortedcontainers, I did the same work. The result is similar with python_skiplist. In this case, I will not talk about it more.

3.3.2 Comparisons between Two Codes

To compare both codes' performance on dictionary size, I created Figure 3.

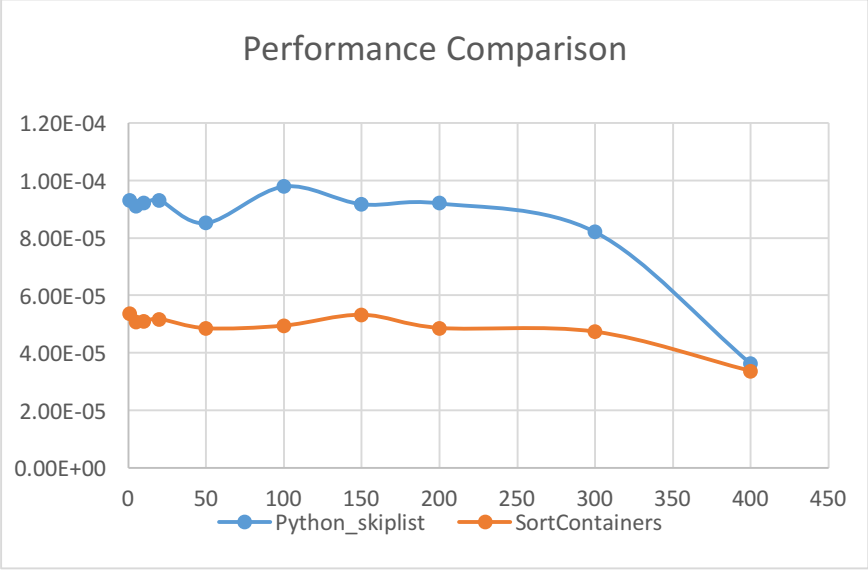

Figure 3

From this figure, we can see that sortcontainer program is better performed than python_skiplist code. Sortcontainer program perform stable until the input size is 400. Python_skiplist takes more time and is unstable. However, both running time are reduced at the input size 400. I assume there are two reasons. One is the collected data is not enough to support the analysis. Another one is the low running time could come from the low string size sample.

3.4 Coverage

Here is the coverage information from my test.

```
Name                                                                                            Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------------------------------------------------------------------------------------
/Library/Python/2.7/site-packages/python_skiplist-0.1-py2.7-macosx-10.11-intel.egg/skiplist/__init__.py      37     24     16      4    36%   1-12, 16,
22-24, 27, 32-41, 44-48, 15->16, 17->19, 21->22, 28->30
/Library/Python/2.7/site-packages/sortedcontainers/sorteddict.py                                  336    299     88      5    10%   5-30, 32-67,
110-111, 116-117, 124, 154-157, 163-214, 220-228, 237-330, 345-746, 109->110, 115->116, 121->124, 153->154, 340->345
/Library/Python/2.7/site-packages/sortedcontainers/sortedlist.py                                 1293   1278    624      3     2%   5-56, 76-78,
99-159, 165-173, 182-769, 778-2418, 75->76, 96->99, 164->165
sut.py                                                                                           1240   1212    567      1     2%   1-12, 20,
34-47, 53, 59, 65-1483, 62->65
-------------------------------------------------------------------------------------------------------------------------------------------
TOTAL                                                                                            2906   2813   1295     13     3%
```

As showed above, my coverage is lower whether in both codes or in total. The reason is that in my test, it does not cover these codes in this project.

4 Problems in Test

In this project, both programs work well on the correctness. I do not find bugs in this test. However, I had these problems in this project.

First, it is difficult to produce the dictionary with size over 300 on my computer. And when string size is over 300, tester cannot output data in correct format. In this case, I cannot have pressure test for these two codes and I could produce wrong data for analysis, like what I showed in 3.3.1.

Second, I realized that there are lots of dictionary are tested in consecutive time when I do the random test by TSTL. I believe it is not due to the random function used to generate input data from python, since these data are tested in not so short time. I would assume that when randomtester picks up the actions, it may use the data which is from previous test.

5 Conclusion

In this project, I learned how to use TSTL to do random test. This is a very interesting process to explore the attracts of TSTL. This is really convenient for automatically random testing. While I create the TSTL file, I really like the part which I can set different size for input. Since I can output all data into file in format and analysis data statistically in R in very easy way. This is really easy and convenient to do perform test.

And while testing, I had one question, which is whether TSTL support multitasks. Due to the heavy running data, my software temporarily does not response for a while. And when I open the activity monitor in Mac, I realize some processors work heavily, but some of in a long idle. In this case, I think, since TSTL concerns on generating test case and test automatically, it would be better to adjust better on multitask if TSTL does not have.

In a word, TSTL is a really convenient tool for random testing.