Report

Time Measurements
-Testing String, StringBuilder, Merge and Insertion Sort in 1 Second

March 12, 2016

Author: Sarpreet Singh Buttar
Supervisor: Jonas Lundberg
Semester: Spring 2016
Course: 1DV507

Contents

1 Introduction

The aim of the report is to explain the process and results of four experiments named as string concatenation, string builder concatenation, insertion sort and merge sort in 1 second. The report also covers the description of why string builder is faster than string during concatenation. The table shows result time in milliseconds[1]. All the results were taken after running the experiment number of times. The result tables also shows the average result.

2 String Concatenation

2.1 Aim

The *aim* of this experiment is to find out how many short strings*(string with 1 character)* and long strings *(string with 80 characters)* can be added to a string in 1 second using + operator.

2.2 Procedure

1) Firstly, one empty string has been created and initialised.

2) Then, one variable *(named as start)* has been created which return the time in milliseconds. This variable used java library class method named System.currentTimeMills() for returning time.

3) Furthermore, a while loop has been established which runs until current milliseconds - variable *(start)* is less than 1000 milliseconds. Inside this loop, short string or long string *(depends on experiment)* was added using + operator.

4) After this loop, one more variable *(named as end)* has been declared which also return time by using System.currentTimeMills() methods. This variable helped to find the runtime. For instance, when the while loop ends, the difference between this variable *(end)* and variable *(start)* is the runtime.

5) During the short string case, the length of the string and number of concatenation was same. However, during the long string I divided the string length by 80 in order to find the number of concatenation.

[1] 1 second = 1000 milliseconds

2.3 Results

String with 1 character

S. No	Time (Milliseconds)	Concatenations	Length
1	1000	20676	20676
2	1000	20780	20780
3	1000	20767	20767
4	1000	20783	20783
5	1000	20807	20807
6	1000	20695	20695
7	1000	20774	20774
8	1000	20750	20750
9	1000	20677	20677
10	1000	20727	20727
Average	1000	20743	20743

String with 80 characters

S.No	Time (Milliseconds)	Concatenations	Length
1	1000	1733	138640
2	1000	1731	138480
3	1000	1740	139200
4	1000	1720	137600
5	1000	1733	138640
6	1000	1730	138400
7	1000	1730	138400
8	1000	1727	138160
9	1000	1720	137600
10	1000	1724	137920
Average	1000	1728	138304

3 StringBuilder Appending

3.1 Aim

The *aim* of this experiment is to find out how many short strings*(string with 1 character)* and long strings *(string with 80 characters)* can be appended to the string builder in 1 second using append method.

3.2 Procedure

The approach is basically the same as string concatenation experiment. The only difference is to add the new string in a string builder using append method and use toString() method at last.

1) Firstly, one string builder has been created.

2) Then, one variable *(named as start)* has been created which return the time in milliseconds. This variable used java library class method named System.currentTimeMills() for returning time.

3) Furthermore, a while loop has been established which runs until current milliseconds - variable *(start)* is less than 1000. Inside this loop, short string or long string *(depends on experiment)* was added using + operator.

4) After this loop, one more variable *(named as end)* has been declared which also return time by using System.currentTimeMills() methods. This variable helped to find the runtime. For instance, when the while loop ends, the difference between this variable *(end)* and variable *(start)* is the runtime.

5) Then, I converted the string builder to string using toString() method and save the time in a variable

6) Again, I run the while loop, but this time it will run until it is smaller than 1000 milliseconds - toString time. Inside this loop, I again added short or long string *(depends on experiment)* in a new string builder.

7) During the short string case, the length of string and number of concatenation is same. However, during long string I divided the string length by 80 in order to find the number of concatenation.

8) The *problem* during the long string case was that it usually takes more time than 1000 milliseconds due to the fact that adding long string takes more time than adding short ones. For instance, if adding long string takes more than 1 milliseconds and the time is 999. The while loop is still true and add the long string but then the runtime become bigger than 1000.

3.3 Results

StringBuilder with 1 character

S.No	Time (Milliseconds)	Concatenations	Length
1	1000	15465049	15465049
2	1000	15665967	15665967
3	1000	15874716	15874716
4	1000	15600816	15600816
5	1000	15200634	15200634
6	1000	15729961	15729961
7	1000	15490223	15490223
8	1000	15300703	15300703
9	1000	15586615	15586615
10	1000	15403652	15403652
Average	1000	15531833	15531833

StringBuilder with 80 characters

S.No	Time (Milliseconds)	Concatenations	Length
1	1158	1074791	85983280
2	1195	1074791	85983280
3	1124	1074791	85983280
4	1197	1074791	85983280
5	1188	1074791	85983280
6	1196	1074791	85983280
7	1256	1074791	85983280
8	1205	1074791	85983280
9	1209	1074791	85983280
10	1047	1074791	85983280
Average	1177	1074791	85983280

4 Insertion Sort

4.1 Aim

The *aim* of this experiment is to find out how many integers and characters can be sorted in 1 second using insertion sort algorithm.

4.2 Procedure

1) Firstly, an array of length 1000 has been created which contains random numbers twice then the length of it. For instance, if the array size is 1000 and it contains random number 2 times array length *(from 0 - 1999)*. The strategy is same even if it is string array because I created my own string which conatins all upper and lower case letters.

2) Then I started the while loop with a condition that if runtime is equal to or greater than 1000 than stop otherwise run continuously.

3) Inside the while loop, I initialised the two variables, one above and one below the insertion algorithm process in order to get the runtime for sorting the array of different lengths.

4) Furthermore, inside the while loop I added some additional if and else if statements in order to get the result as accurate as possible. Each time, I am increasing the size of array if runtime is less than 1000 millesecobds or vice versa.

5) The purpose of these additional statements is to increase or decrease the size of array at different stages. For instance, if runtime is more than 1000 milliseconds than one of this statement will run and decrease the size of array by 1 and initialised the runtime to 0 so that while loop will not stop. Other statements are for general purpose, such as break the loop if runtime is 999 or 1000 or 1001 to get approximate result and increase the size slowly if runtime is near to 1000 milliseconds.

6) Three additional methods have been created. Two for generating a new array with random number and random characters and one for calculating the average time and length. The average method I created for myself to find the approximate time for sorting integers and characters arrays of different lengths. This method is extra work and I am only running it when runtime is between 950 and 1000 milliseconds because there is no need to find the average from beginning of the runtime.

7) This approach takes bit more time as it will only stop if one the statement inside the while statement will be true.

8) The advantage of this approach is to get accurate result on different computers. For instance, if I only run the while loop until runtime is smaller than 1000 milliseconds, I will not get the accurate result. I saw that java compiler always improve my code. For instance, if 100000 integers is sorted in 300 milliseconds, next time the array which is bigger in length can be sorted in less than 300 milliseconds and then even more less. Due to these facts, I added some additional statements as I mentioned above.

4.3 Results

Integer Insertion Sort

S.No	Time (Milliseconds)	Array Length
1	999	92226
2	1001	92245
3	999	92229
4	999	92229
5	1000	92296
6	999	92271
7	999	92224
8	1000	92227
Average	999	92243

String Insertion Sort

S.No	Time (Milliseconds)	Array Length
1	1001	11499
2	1001	11549
3	1001	11516
4	999	11497
5	1001	11506
6	1001	11501
7	1000	11551
8	1001	11505
Average	1000	11515

5 Merge Sort

5.1 Aim

The *aim* of this experiment is to find out how many integers and characters can be sorted in 1 second using merge sort algorithm.

5.2 Procedure

The approach is basically the same as insertion sort. The only difference is to execute merge sort algorithm rather insertion sort.

1) Firstly, an array of length 1000 has been created which contains random numbers twice then the length of it. For instance, if the array size is 1000 and it contains random number 2 times array length *(from 0 - 1999)*. The strategy is same even if it is string array because I created my own string which conatins all upper and lower case letters.

2) Then I started the while loop with a condition that if runtime is equal to or greater than 1000 than stop otherwise run continuously.

3) Inside the while loop, I initialised the two variables, one above and one below the insertion algorithm process in order to get the runtime for sorting the array of different lengths.

4) Furthermore, inside the while loop I added some additional if and else if statements in order to get the result as accurate as possible. Each time, I am increasing the size of array if runtime is less than 1000 millesecobds or vice versa.

5) The purpose of these additional statements is to increase or decrease the size of array at different stages. For instance, if runtime is more than 1000 milliseconds than one of this statement will run and decrease the size of array by 1 and initialised the runtime to 0 so that while loop will not stop. Other statements are for general purpose, such as break the loop if runtime is 999 or 1000 or 1001 to get approximate result and increase the size slowly if runtime is near to 1000 milliseconds.

6) Three additional methods have been created. Two for generating a new array with random number and random characters and one for calculating the average time and length. The average method I created for myself to find the approximate time for sorting integers and characters arrays of different lengths. This method is extra work and I am only running it when runtime is between 950 and 1000 milliseconds because there is no need to find the average from beginning of the runtime.

7) This approach takes bit more time as it will only stop if one the statement inside the while statement will be true.

8) The advantage of this approach is to get accurate result on different computers. For instance, if I only run the while loop until runtime is smaller than 1000 milliseconds, I will not get the accurate result. I saw that java compiler always improve my code. For instance, if 100000 integers is sorted in 300 milliseconds, next time the array which is bigger in length can be sorted in less than 300 milliseconds and then even more less. Due to these facts, I added some additional statements as I mentioned above.

5.3 Results

Integer Merge Sort

S.No	Time (Milliseconds)	Array Length
1	999	400026
2	999	400080
3	1000	400043
4	999	400078
5	999	400075
6	1001	400054
7	1000	400025
8	1000	400055
Average	999	400054

String Merge Sort

S.No	Time (Milliseconds)	Array Length
1	1001	510021
2	999	510003
3	1001	509971
4	1001	509979
5	1001	510003
6	1000	510003
7	1000	509999
8	1001	510002
Average	1000	509997

6 Why StringBuilder is Faster?

Before explaining why string builder is faster than string concatenation using the + operator, I want to tell that string concatenation also use string builder to add another string. For instance,

```
String str1 = "";
String str2 = "Java";
str1 = str1 + str2;
str = "Java";
```

This actually happens like this:

```
StringBuilder sb = new StringBuilder();
str = sb.append(str2).toString();
```

The reason why string builder is faster is that string concatenation using + operator makes a copy during each concatenation which requires memory to save and time to make that copy. Whereas, string buider adds the new string at the end position and only make copy in some cases such as during resize or if inserting element in the middle. In the experiment, I am only adding the string which means string builder only saves copy when the data becomes to big. That is why there is a big difference in the result.