Time Measurements

Nima Safavi

Software Technology Group
Department of Computer Science, Linnaeus University
Ns222tv@student.lnu.se

Abstract— **This is a short report to be submitted regarding Exercise 8 in Assignment 4 and exercise 6 in assignment 3 in course 1DV507 – Programming and Data Structures. The task handled in this report is to compare concatenating strings using + operator as well as StringBuilder append method. For each method two different approaches were used to conduct the experiments. Concatenating a short string as well as a long string.**

Our experiments show that StringBuilder is way faster than + operator. The longest string that could be built in 1 second was 687865920 characters long using append method 8598324 times including the final toString() method call which seems to be a quite expensive method.

I. Exercises

The purpose of this exercise is to become familiar with the code analysis as well as comparing the outcome when tackling problems using different approaches. Also how a code analysis written report might look like.

The problem we handle in this report is to find the fastest approach, the longest string and total number of concatenations that might be used to build the longest string in 1 second.

II. Experimental Setup

All experiments was done on a Lenovo legion with an Intel Core i5 processor (2.2GHz) with 16GB of memory. We used Java SE 1.8.0.241 and Java version "13.0.2" 2020-01-14
Java(TM) SE Runtime Environment (build 13.0.2+8)
Java HotSpot(TM) 64-Bit Server VM (build 13.0.2+8, mixed mode, sharing) and during the experiments we also allowed the Java Virtual Machine to use a heap space of 4GB using the VM arguments: -Xmx4096m. Moreover, all other applications were closed while performing the experiments. Only concatenation, counter increment and timer check were included during the time measurements. We used the clock System.currentTimeMillis() in all our measurements.

We tried to use the same approach for all four experiments. However, including the final toString() method in the time measurement was quite challenging as it was not clear how much time is needed for this method. We ran the program 10 times in order to find the average amounts.

1.1 Concatenating using one-character long string using + operator

The code used for this experiment is shown below:

```
String shortString = "x";
String tempString = "";
int numberOfConcatanations = 0;
long startTime = System.currentTimeMillis();
while ((System.currentTimeMillis() - startTime) < 1000) {
     tempString = tempString + shortString;
      numberOfConcatanations++;
}
System.out.println("Finall String length using + operator while adding short strings: " + tempString.length());
System.out.println("Total number of concatanations using + operator while adding short strings: " + numberOfConcatanations + "\n\n");
```

```
Actual Runs
  string length = 87285, number of concatanations: 87285
  string length = 86855, number of concatanations: 86855
  string length = 87145, number of concatanations: 87145
  string length = 85831, number of concatanations: 85831
  string length = 86836, number of concatanations: 86836
  string length = 86965, number of concatanations: 86965
  string length = 87206, number of concatanations: 87206
  string length = 86915, number of concatanations: 86915
  string length = 86702, number of concatanations: 86702
  string length = 87000, number of concatanations: 87000
```

1.2 Concatenating using 80-characters long string using + operator

The code used for this experiment is shown below:

```
String longString =
"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
String  tempString = "";
numberOfConcatanations = 0;
startTime = System.currentTimeMillis(); // fetch starting time
while ((System.currentTimeMillis() - startTime) < 1000) {
        tempString = tempString + longString;
        numberOfConcatanations++;
}

System.out.println("Finall String length using + operator while adding long strings: " + tempString.length());
System.out.println("Total number of concatanations using + operator while adding long strings: " + numberOfConcatanations + "\n\n");
```

```
Actual Runs
  string length = 557440, number of concatanations: 6968
  string length = 559040, number of concatanations: 6988
  string length = 559920, number of concatanations: 6999
  string length = 559280, number of concatanations: 6991
  string length = 557920, number of concatanations: 6974
  string length = 556640, number of concatanations: 6958
  string length = 551760, number of concatanations: 6897
  string length = 556160, number of concatanations: 6952
  string length = 554560, number of concatanations: 6932
  string length = 557680, number of concatanations: 6971
```

As can be seen from the results, the longer the string that is used for concatenation is, the slower the runtime will be. (Average of about 86000 times when a one-character long string was used vs an average of 6970 time when a 80-character long string was used).

1.3 Concatenating using one-character long string and StingBuilder Append method

The code used for this experiment is shown below:

```
long endTimeWithToStringMethod = 0;
long endTimeWithoutToStringMethod = 0;
StringBuilder sb = new StringBuilder();
numberOfConcatanations = 0;
startTime = System.currentTimeMillis(); // fetch starting time
while (true) {
     sb.append(shortString);
     numberOfConcatanations++;
     if((System.currentTimeMillis() - startTime) < 970)
          continue;
      else
          {
         endTimeWithoutToStringMethod = System.currentTimeMillis() -
         startTime;
          sb.toString();
          endTimeWithToStringMethod = System.currentTimeMillis() -
          startTime;
          break;
          }
  }

          System.out.println("Finall String length using StringBuilder while
          adding short strings: " + sb.length());
          System.out.println("Total number of concatanations using
          StringBuilder while adding short strings: " +
          numberOfConcatanations);
          System.out.println("Total time in milliSeconds excluding
          StringBuilder ToString method " +
          endTimeWithoutToStringMethod);
          System.out.println("Total time in milliSeconds including
          StringBuilder ToString method " + endTimeWithToStringMethod
          + "\n\n");
```

For this experiment we started with a try-and-error approach to find a size interval (Min,Max) that covers an execution time of about 1 second. We found that the largest string possible to be built using string builder and a one-character long string is around 114074568 characters long with the total of 113129293 times appending.

```
Actual Runs
string length = 115344295, number of concatanations: 115344295
string length = 111342543, number of concatanations: 111342543
string length = 115144998, number of concatanations: 115144998
string length = 113304961, number of concatanations: 113304961
string length = 111217914, number of concatanations: 111217914
string length = 111000474, number of concatanations: 111000474
string length = 113129293, number of concatanations: 113129293
string length = 113013132, number of concatanations: 113013132
string length = 113352283, number of concatanations: 113352283
string length = 114887215, number of concatanations: 114887215
```

As can be seen from the results String builder append method is way faster that the + operator.

1.4 Concatenating using 80-characters long string and StingBuilder Append method

The code used for this experiment is shown below:

```
sb = new StringBuilder();
numberOfConcatanations = 0;
startTime = System.currentTimeMillis(); // fetch starting time
while (true) {
          sb.append(longString);
          numberOfConcatanations++;
          if((System.currentTimeMillis() - startTime) < 770)
                    continue;
          else
          {
              endTimeWithoutToStringMethod = System.currentTimeMillis()
              - startTime;
               sb.toString();
               endTimeWithToStringMethod = System.currentTimeMillis() -
               startTime;
                break;
          }
}

System.out.println("Finall String length using StringBuilder while adding long
strings: " + sb.length());
System.out.println("Total number of concatanations using StringBuilder while
adding long strings: " + numberOfConcatanations);
System.out.println("Total time in milliSeconds excluding StringBuilder
ToString method " + endTimeWithoutToStringMethod);
System.out.println("Total time in milliSeconds including StringBuilder
ToString method " + endTimeWithToStringMethod);
```

```
Actual Runs
  string length = 687865920, number of concatanations: 8598324
  string length = 687865920, number of concatanations: 8598324
  string length = 687865920, number of concatanations: 8598324
  string length = 687865920, number of concatanations: 8598324
  string length = 687865920, number of concatanations: 8598324
  string length = 687865920, number of concatanations: 8598324
  string length = 687865920, number of concatanations: 8598324
  string length = 687865920, number of concatanations: 8598324
  string length = 687865920, number of concatanations: 8598324
  string length = 687865920, number of concatanations: 8598324
```

The results were the same in every run. We noticed the same thing in this experiment as the one in the previous experiment. That is the longer the string that is used for concatenation is, the slower the StringBuilder append method runtime will be.

In this part my experiments with integers and String array with different length using insertion sort and merge sort will be discussed. I used the same approach as previous exercise. I will show the time for each run although I have changed the length of each array many times to come to a number for which the process time is approximately one second.

2.1 Implementing insertion sort for random integers array

Here is the code for this experiment and the output after 10 times run:

```
SortingAlgorithms sortingAlgorithms = new SortingAlgorithms();
        Random random = new Random();
        var c = new Comparator<String>();
       @Override
       public int compare(String o1, String o2) {
          return o1.compareTo(o2);
       }};
        int[] randomIntegerArray = new int[80000];
        for (int i = 0; i < randomIntegerArray.length; i++) {
                randomIntegerArray[i] = random.nextInt();
        }
long startTime = System.currentTimeMillis(); // fetch starting time
        var tempArray3
=sortingAlgorithms.insertionSort(randomIntegerArray);
        long timeEllapsed = System.currentTimeMillis() - startTime;
        System.out.println("Sorting int array with size 80000 and with
random generated integers using insertion sort took: "  + timeEllapsed +
"milliSeconds. \n\n");
```

I started my array with length of 100,000 but because we intend it to be operated in 1 second, so I decrement the length to 90,000 and then 80,000 to match our expected time.

Actual runs :

```
Array with size 80000 took 1041 Milliseconds.
Array with size 80000 took 792 Milliseconds.
Array with size 80000 took 830 Milliseconds.
Array with size 80000 took 920 Milliseconds.
Array with size 80000 took 791 Milliseconds.
Array with size 80000 took 819 Milliseconds.
Array with size 80000 took 886 Milliseconds.
Array with size 80000 took 749 Milliseconds.
Array with size 80000 took 754 Milliseconds.
Array with size 80000 took 832 Milliseconds.
```

2.2 Implementing insertion sort for string array

Here is the code for this experiment and the output after 10 times run:

```
String[] randomStringArray = new String[18000];
        for (int i = 0; i < randomStringArray.length; i++) {
                byte[] array = new byte[10]; // length is bounded by 10
                random.nextBytes(array);
                String generatedString = new String(array,
Charset.forName("UTF-8"));
                randomStringArray[i] = generatedString;
startTime = System.currentTimeMillis(); // fetch starting time
        var tempArray4
=sortingAlgorithms.insertionSort(randomStringArray, c);}
        timeEllapsed = System.currentTimeMillis() - startTime;
        System.out.println("Sorting String array with size 18000 and with
random generated strings using insertion sort took: "  + timeEllapsed +
"milliSeconds. \n\n")
```

Actual runs:

```
String array with size 18000 took 1037 Milliseconds.
String array with size 18000 took 1019 Milliseconds.
String array with size 18000 took 823 Milliseconds.
String array with size 18000 took 991 Milliseconds.
String array with size 18000 took 838 Milliseconds.
String array with size 18000 took 1008 Milliseconds.
String array with size 18000 took 958 Milliseconds.
String array with size 18000 took 852 Milliseconds.
String array with size 18000 took 854 Milliseconds.
String array with size 18000 took 866 Milliseconds.
```

2.3 Implementing merge sort for random integers array

Here is the code for this experiment and the output after 10 times run:

```
int[] xlargeSizeRandomIntegerArray = new int[5000000];
        for (int i = 0; i < xlargeSizeRandomIntegerArray.length; i++) {
                xlargeSizeRandomIntegerArray[i] = random.nextInt();
        }
startTime = System.currentTimeMillis(); // fetch starting time
        tempArray3
=sortingAlgorithms.mergeSort(xlargeSizeRandomIntegerArray);
        timeEllapsed = System.currentTimeMillis() - startTime;
        System.out.println("Sorting int array with size 5 000 000 and with
random generated integers using merge sort took: "  + timeEllapsed +
"milliSeconds. \n\n");
```

Actual runs:

```
Array with size 5000000 took 940 Milliseconds.
Array with size 5000000 took 942 Milliseconds.
Array with size 5000000 took 928 Milliseconds.
Array with size 5000000 took 947 Milliseconds.
Array with size 5000000 took 944 Milliseconds.
Array with size 5000000 took 945 Milliseconds.
Array with size 5000000 took 931 Milliseconds.
Array with size 5000000 took 913 Milliseconds.
Array with size 5000000 took 927 Milliseconds.
Array with size 5000000 took 982 Milliseconds.
```

2.4 Implementing merge sort for random generated string array

Here is the code for this experiment and the output after 10 times run:

```
String[] xlargeSizeRandomStringArray = new String[1300000];
        for (int i = 0; i < xlargeSizeRandomStringArray.length; i++) {
                byte[] array = new byte[10]; // length is bounded by 10
                random.nextBytes(array);
                String generatedString = new String(array,
Charset.forName("UTF-8"));
                xlargeSizeRandomStringArray[i] = generatedString;
        }
startTime = System.currentTimeMillis(); // fetch starting time
        tempArray4
=sortingAlgorithms.mergeSort(xlargeSizeRandomStringArray, c);
        timeEllapsed = System.currentTimeMillis() - startTime;
        System.out.println("Sorting String array with size 1 300 000 and
with random generated strings using merge sort took: "  + timeEllapsed +
"milliSeconds. \n\n");}
```

```
String array with size 1,300,000 took 944 Milliseconds.
String array with size 1,300,000 took 923 Milliseconds.
String array with size 1,300,000 took 909 Milliseconds.
String array with size 1,300,000 took 917 Milliseconds.
String array with size 1,300,000 took 893 Milliseconds.
String array with size 1,300,000 took 938 Milliseconds.
String array with size 1,300,000 took 900 Milliseconds.
String array with size 1,300,000 took 933 Milliseconds.
String array with size 1,300,000 took 943 Milliseconds.
String array with size 1,300,000 took 884 Milliseconds.
```

III. Possible Improvements

This experiment could produce more accurate results if tested with a higher number of runs and also on more computers with similar hardware. Including the final StringBuilder toString() method in the time measurement was also challenging. An improvement suggestion is to measure the time for the appendings and the final toString() method separately.

IV. Conclusion

After this experiment, it turned out that Concatanation using StringBuilder append method is much faster that the + operator. Also the longer the string that is used for concatenation is, the slower the runtime will be for both + operator and StringBuilder append method. For the second exercise (insertion sort and merge sort) with short survey on the size of arrays and the type of value we can conclude that merge sort is sorting data much faster than insertion sort.