Time Measurements

Department of Computer Science, Linnaeus University Växjö.

1DV507-VT20

A by short report done in the course 1DV507-VT20 lab 4.

Sami Mwanje

Mm223kk@student.lnu.se

Introduction

This is a report about exercise six and seven, regarding lab four. In exercise six the user got information about how strings can be constructed. The first is by using the "+"-operator and the second is by using the StringBuilder class. With this information the task was to find the fastest approach by measuring the links and the length of each string in one second. There were two different approaches. One for short strings containing only one character, and the other one a long string representing a row with 80 characters. In exercise seven the task was for the user to use different sorting methods from earlier exercises. Insertion sort and Merge Sort. In this report only Insertion sort was used. The task here was to find out how many integers and strings that could be sorted in one second, using insertion sort. This report is going to describe and discuss the implementation of these tasks.

All the tasks in exercise 6-7 were done using a Windows desktop computer. The computer has a 4-core processor of Intel Core i5 running at (4.0 GHz). With a RAM of 16 Gb. The implementations were done using Eclipse IDE for Java Developers. After some implementations and measurements, a realization that larger values were need for the heap space came up. In this report the used heap space is: -Xmx4096m -Xms4096m. The objective when preforming a process-using experiment is to make sure that no other program was slowing down the measurements. Therefore, the measurements were compiled with only Eclipse IDE and some windows background applications open. When it comes to measuring with the time, System.currentTimeMillis() was used. System.currentTimeMillis() was used because it will return the current time in milliseconds which is needed during measurements with time. In exercise six each string concatenation-method expect to long StringBuilder- concatenation was measured 10 times in around 1000 milliseconds. There will also be a value called "loops" used. This value will measure the for-loops, so the program gets closer 1000 milliseconds. More on this further in the report. In exercise seven the measure was only done one time by the computer. Each

Sami Mwanje
Linnaeus University, Växjö
Student Mail: mm223kk@studnet.lnu.se

measure had to be manually noted before the program was re-booted in a total of 10 times. The implementation and the idea behind it will be discussed further in the report.

Task 1.

The first task was to find out how many concatenations a basic "+-operator" can do in one second. My first idea was to use a while loop that was running for 1000 milliseconds. But after some implementations and runs I released the computer will be having a hard time to finish a while-loop of 1000 milliseconds while it is completing harder tasks. Therefore, I decided to use:

```
start = System.currentTimeMillis();
for(int x = 0; x < loops; x ++)
str += "a";
endTime = System.currentTimeMillis();
loadTime = endTime - start;
```

The first line is to save the current time. In the for-loop I am using a value called loops. This value is measured by the user. After running the program, a couple of times, I realized that the for-loop had to be repeated about 65000 times before it gave a time that was around 1000 milliseconds in this task. There for I created the variable "loops" that will loop through the "+-operator" a given amount of times. "endTime" is used to save the current time when the loop is done, and "loadTime" is used to save time the time it took to go through the loop.

One thing I noticed was that the timing was always off if the program has not been running for a while. This means that even computers need "warm-ups". I tried to fix this using:

```
if(loadTime > 1150 )        // Check if the timing is off.
loops -= warmUp;            // Adjust "loops" if the timing is off.
else if(loadTime < 950 )    // Check if the timing too low.
loops += warmUp;            // Adjust "loops" if the timing was too low.
```

The first line checks if the timing is off. How far from 1000 milliseconds is the program? If it is larger than 150 milliseconds, the program will readjust the loops-value to a lower value. If it is

Sami Mwanje
Linnaeus University, Växjö
Student Mail: mm223kk@studnet.lnu.se

under 950 milliseconds, then the program will readjust the value to a larger one. The increase and the lowering is +-250 the is added in a value called “warmUp”. Every task has its own “warmUp”-value because of varying usage of process memory when executing each task.
10 runs of the first task gave:

```
Adding short strings containing  only one character:--------------------------------
-----------------------

Time to execute: 1440 ms ,   Concatenations 1: 65004 ,   Length after 1 sec: 65004
Time to execute: 1215 ms ,   Concatenations 2: 64750 ,   Length after 1 sec: 64750
Time to execute: 1005 ms ,   Concatenations 3: 64500 ,   Length after 1 sec: 64500
Time to execute: 974  ms ,   Concatenations 4: 64500 ,  Length after 1 sec: 64500
Time to execute: 970  ms ,   Concatenations 5: 64500 ,   Length after 1 sec: 64500
Time to execute: 976  ms ,   Concatenations 6: 64500 ,   Length after 1 sec: 64500
Time to execute: 959  ms ,   Concatenations 7: 64500 ,   Length after 1 sec: 64500
Time to execute: 998  ms ,   Concatenations 8: 64500 ,   Length after 1 sec: 64500
Time to execute: 968  ms ,   Concatenations 9: 64500 ,   Length after 1 sec: 64500
Time to execute: 960  ms ,   Concatenations 10:64500 ,   Length after 1 sec: 64500
Average load time: 1046.5
Average length adding one char:   64575.4
Average concatenations adding one char:   64575.4
```

It can here be observed that the computer managed to build a string with the length of 64575. This means that the string contained 64575 chars and 64575 concatenations were done. Looking at our value “loops” that was 65000 from the beginning means that around one concatenation was done for each time the computer looped through “str += "a"”. A relative stable run from an objective viewpoint. We also had an average running time of 1046 milliseconds which is very close 1s. This seems like a pretty simple and smooth task for the computer.

Sami Mwanje
Linnaeus University, Växjö
Student Mail: mm223kk@studnet.lnu.se

Task 2.

Task two was to measure the time it took to link 80-char-strings using the “+-operator”. The implementation for this task is basically the same as in task one. The changes here are the values “loops = 5200”, “warmpUp = 100” and the execution code in the for-loop is now:

str+="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";

.

A notice here is the value loops. We have much smaller value than before. This means that less concatenations will be done in this short amount of time. But the program may finish with a much bigger string-length.

10 runs runs of the second task gave:

```
 Adding long strings containing only 80 characters: --------------------------------
Time to execute: 998 ms Concatenations: 5200.0,    Length after 1 sec: 416000
Time to execute: 1009 ms Concatenations: 5200.0,   Length after 1 sec: 416000
Time to execute: 975 ms Concatenations: 5200.0,    Length after 1 sec: 416000
Time to execute: 984 ms Concatenations: 5200.0,    Length after 1 sec: 416000
Time to execute: 972 ms Concatenations: 5200.0,    Length after 1 sec: 416000
Time to execute: 987 ms Concatenations: 5200.0,    Length after 1 sec: 416000
Time to execute: 1001 ms Concatenations: 5200.0,   Length after 1 sec: 416000
Time to execute: 1013 ms Concatenations: 5200.0,   Length after 1 sec: 416000
Time to execute: 982 ms Concatenations: 5200.0,    Length after 1 sec: 416000
Time to execute: 991 ms Concatenations: 5200.0,    Length after 1 sec: 416000
Average load time: 991.2
Average length, adding string with 80 chars:   416000.0
Average concatenations adding string with 80 chars:   5200.0
```

As it can be observed the computer managed to build a string with the length 416000 a much greater value than the first task which only managed to construct a string with the length 64575. A much smaller value. A close look at this result shows that the first task also looped through the “+-operator” around 64575/s (concatenations -per-second) while, in this task the computer only managed to loop through 5200/s. This means this task is a more process and time taking approach. It seems logic because the computer is working for a greater length.

Sami Mwanje
Linnaeus University, Växjö
Student Mail: mm223kk@studnet.lnu.se

Task 3.

Task is about doing the same approach ask in task one but now by using another method linking the string. The method used here is so called a StringBuilder. A StringBuilder-string is created by using **StringBuilder sB = new StringBuilder();** String can now be linked to the chars and strings can now be linked to the StringBuilder using sB.append("string/char").

The setup in this task was somewhat the same as in the other two. The main difference here was the method inside the for-loop. Used here is **sB.append("a");** which works like "string += "a" " in the previous tasks.

Exceptions here are a somehow different values both in length and Concatenations than task one. The answer is for me unknown. But I guess it has something to with that a StringBuilder is not really a "string" of chars and so on. The values contained by the object do only become what we refer to as a "string" when we call "**sb.toString();**". The method turns a StringBuilder-object to a string, that can be printing in a "Sy**stem.out.print();".**

10 runs of the third task gave:

```
SHORT stringBuilder. Adding short strings containing  one char:----------------------
--------------------------------

Time to execute: 1299 ms Concatenations: 160000000,    Length after 1 sec: 160000000
Time to execute: 1024 ms Concatenations: 159990000,    Length after 1 sec: 159990000
Time to execute: 1006 ms Concatenations: 159990000,    Length after 1 sec: 159990000
Time to execute: 1006 ms Concatenations: 159990000,    Length after 1 sec: 159990000
Time to execute: 1005 ms Concatenations: 159990000,    Length after 1 sec: 159990000
Time to execute: 1006 ms Concatenations: 159990000,    Length after 1 sec: 159990000
Time to execute: 1007 ms Concatenations: 159990000,    Length after 1 sec: 159990000
Time to execute: 1005 ms Concatenations: 159990000,    Length after 1 sec: 159990000
Time to execute: 1016 ms Concatenations: 159990000,    Length after 1 sec: 159990000
Time to execute: 1010 ms Concatenations: 159990000,    Length after 1 sec: 159990000
Average load time: 1038.4
Average length, appending stringBuilder Concatenations one char:   1.59991E8
Average Concatenations appending stringBuilder Concatenations one char:   1.59991E8
```

Sami Mwanje
Linnaeus University, Växjö
Student Mail: mm223kk@studnet.lnu.se

It was much harder getting the timing right on this approach, but after some measurements a good start value for “loops” showed out to be 160000000! This is a very large value compared to the last task this means as can be seen in the result that the program with a StringBuilder can do around 160000000 Concatenations in a second! I would call it a greater but a more memory/processing needing approach. In this approach the “warmUp”-value had to be adjusted up 10000. This means that if the program were off 1000 milliseconds, the loop would have slowed down or speed up with 10000 loops. That is a very large number used just to get the timing right by milliseconds. This means that this approach uses a large amount of computer power, but the result is of course a much more bigger string in both length and concatenations. Googling around “why is StringBuilder is faster” gave me the answer that when using the “+-operator” a new string is created and then linked together with the original string. This will therefore consume much more memory at shorter time, compared to what a string builder will do of that time. StringBuilder uses instead an array of chars. Using append() the builder will check for memory in the string and run the “System.arraycopy”-method to re-construct the preferred string.[1]

Task 4.

[1] https://stackoverflow.com/questions/22439177/why-stringbuilder-is-much-faster-than-string

Sami Mwanje
Linnaeus University, Växjö
Student Mail: mm223kk@studnet.lnu.se

Like in task two it was time for the long string concatenations. The main difference in this task is that the task is now approached using the StringBuilder object. The method used in this task is:

sB.append("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaa");

As seen in the previous tasks it took longer time linking a 80-char-string than a one char string. The result here should be the same. In exercise two the length was obviously much larger but the concatenations were much less than one char. So are the exceptions here. But judging from previous results the concatenations here should be larger than in task two.

Notice that in this task I had some problems with measuring in more than **8589999** loops. I tried bigger values, but the program crashed each time. Therefore, the measurements shown down here are only showing how many concatenations were done in about **431** milliseconds. As earlier stated, I supposed this has to do with the usage of memory/process power when it comes to very, very big string-length. At large string-length not even a heap size change can keep up, maybe that inner array constructed by the StringBuilder no long can keep up?

10 runs of the fourth task gave the average time 431 milliseconds gave us:

```
LONG stringBuilder 80 char:--------------------------------------------

Time to execute: 515 ms Concatenations: 8589999.0,    Length: 687199920
Time to execute: 422 ms Concatenations: 8589999.0,    Length: 687199920
Time to execute: 429 ms Concatenations: 8589999.0,    Length: 687199920
Time to execute: 428 ms Concatenations: 8589999.0,    Length: 687199920
Time to execute: 415 ms Concatenations: 8589999.0,    Length: 687199920
Time to execute: 422 ms Concatenations: 8589999.0,    Length: 687199920
Time to execute: 427 ms Concatenations: 8589999.0,    Length: 687199920
Time to execute: 401 ms Concatenations: 8589999.0,    Length: 687199920
Time to execute: 422 ms Concatenations: 8589999.0,    Length: 687199920
Time to execute: 429 ms Concatenations: 8589999.0,    Length: 687199920
Average load time: 431.0
Average length after appending StringBuilder 80:   6.8719992E8
Average Concatenations appending StringBuilder 80:   8589999.0
```

Sami Mwanje
Linnaeus University, Växjö
Student Mail: mm223kk@studnet.lnu.se

Compared to task two which had concatenations value of **5200** which means 5200 concatenations/s, this is a much larger value. The program does **8589999** concatenations which is about 165 times bigger than concatenations in task two, and this is only done in 431 milliseconds! If the program was running at the same rate for one second it has may had come up to approximately **1717998** concatenations and a length of **687199920*2.**

The conclusion that StringBuilder is a much, much faster approach when it comes to large concatenations. But keep in mind that it also requires a large amount of power from the computer when linking up long strings. An approach can be using "+operator "for easy task that are not dependent on time. This will then be done smoothly, but when we come up to big arrays and time depending projects, the StringBuilder is always to approach.

Task 5.

This task was about finding out how many integers that could be sorted user the method "**insertionSort**();". This is a method that will sort an un-sorted integer-array. In this exercise the program constructed an array containing random integers. The implementation that was used was to create a integer array that expanded, sorted and then measured. So, the time it took to sort was measured. Not to expand. A peek of the code is:

```
start = System.currentTimeMillis(); // Start the time.
numArray = insertionSort(numArray);
end = System.currentTimeMillis();   // Stop the time.
LoadTime = end - start;
```

Sami Mwanje
Linnaeus University, Växjö
Student Mail: mm223kk@studnet.lnu.se

The time measure starts before running the sort method and ends after running the sort method. The values are then saved in “loadTime”. Here is where the while-loop come in:

```
int z = 1;  // Used to increase the size of the array.

while(LoadTime < 1000) {// Check if it took 1000 ms ( 1 s ) to sort.
// If not... re-try with a bigger array that contains more integers.
int[]  numArray = new int[10*z]; // Create new array with size 10
z += 200;   // On my machine this was the perfect increase-array-value that gave 1000
ms was.
for(int x = 0; x < numArray.length; x++) {
numArray[x] = (int)(Math.random() * ( numArray.length+5)); }           // Generate
random numbers between 0 and the length of the Int-array.
```

The while loop will check if the sort took less then 1000 milliseconds. If it takes more than 1000 milliseconds expand the array with the value*z, resort and measure the time once again and so on. Using this approach, the while-loop will end when it takes about 1000 milliseconds to sort the integers in the array. A print of the result is:

```
InsertionSort--------------------

Number of integers sorted: 62010 integers.
Time spent to sort integers: 1027 ms.

InsertionSort--------------------
```

The result used is not an average because of difficulties saving a value after each approach. The print screen was clear after some 100s of lines so no values could be founded from there. Also, after re-running the code a couple of times I noticed that I **always** got the value **62010 sorted** integers no matter what. The timing could be +-30 ms, but that did not seem to affect the result. So, the conclusion is this is a pretty “linear”-approach and each loop takes more then 30 ms to sort. Otherwise I should have noticed other values.

Sami Mwanje
Linnaeus University, Växjö
Student Mail: mm223kk@studnet.lnu.se

Task 6.

On task 6 the same idea as in task 5 was used. An array that expands and then sorted if it takes under 1000 milliseconds. The main difference here is how the random strings containing 10 chars were constructed:

```
final static String[] alphabet = { "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r",s", "t", "u", "v", "w", "x", "y", "z" };
```

Here and array with each char in the alphabet was constructed. A random number generator will then pick a random element from the array. That element will than be adad to a string. This loop is done 10 times, for 10 chars in the string. The string is then added to array. It's now about how big the array is needed to be for the program to take 1000 milliseconds to sort it.

The result was:

```
InsertionSortStrings---------------------

Amount of strings sorted: 12010 Strings.
Time spent to sort strings: 1054 ms.

InsertionSortStrings---------------------
```

Even here the result was only done one time. It seems like the value **12010** needs more than 1054+-30 ms before a change. Like in the previous task we than understand that big array takes time to sort. Too much time for the program to make it 12011 in just milliseconds. Comparing sorting strings and integers we can see that it goes much faster when sorting integers. This is understandable because the compare-To method needs to go through every char in a string to check if it is less than or bigger than another one. Which obviously is more memory/process approach.

Sami Mwanje
Linnaeus University, Växjö
Student Mail: mm223kk@studnet.lnu.se

Sami Mwanje
Linnaeus University, Växjö
Student Mail: mm223kk@studnet.lnu.se