Linnéuniversitetet
Kalmar Växjö

Report

Time Measurement for String Concatenation, StringBuilder Appending, Insertion and merge sort.

Author: Musatafa Alsaid
Supervicer: Jonas Lundberg
Semester: Spring 2017
Course code:1DV507

1. Introduction

The aim of this report is to measure the runtime for four processes which including String Concatenation, StringBuilder Appending, and Insertion and Merge sort. Moreover it is calcoluted how many concatenations and length in one seconds for String and StringBuilder with one character and 80 characters. The report contains tables for all experments and it explains how all experments applied. Moreover the report gives an answer why StringBuilder is faster than String Concatenation.

2. String Concatenation

The aim of this experment is to see how many short String and long string contains one character and eighty characters respectively can be added to another String in one second.

2.1 procedures

1- One empty String has been created and inotialized by ”” .
2-Start time has been created which gives the current time in milliseconds.
3- While loop has been created which includes concatenation Strings operation -+- for one second.
4- End time has been created which gives the current time after finishing the concatenation process inside while loop for one second.
5- The run time has been calculated which is the difference between end time and start time.
6- `oneStringLength` is needed to calculate the average of String length and concatenations when one charcter only is concatenated to String by + operation. `eightyStringLength` variable is needed to calculate the average of String length when eighty charcters are concatenated to String by + operation and `eightyStringcon` variable is needed to calculate the average of concatenations when eighty charcters are appenended to String Builder.
7- Print the interested results for our experment which are the run time in milliseconds, concatenation and String length.
8- In main method a for loop has been created to calcolate five times how many concatenations and string length in one second and after that the average is taken for that process as is shown in the table 1 and 2.
9- Notice that two array lists has been created called legthlist and conList to add each values of the string length and concatenation resprctively in one second to use them later to find the length and concatenations average respectively .

2.2 Result

Table 1. String with one character

ExNo	Time (Milliseconds)	Concatenation	Length
1	1000	47182	47182
2	1000	47216	47216
3	1000	49020	49020
4	1000	49503	49503
5	1000	49871	49871
Average	**1000**	**48558**	**48558**

Table 2. String with 80 characters

ExNo	Time (Milliseconds)	Concatenation	Length
1	1000	3535	282800
2	1000	3748	299840
3	1000	3732	298560
4	1000	3847	307760
5	1000	3903	312240
Average	**1000**	**3753**	**300240**

3. StringBuilder Append

The aim of this experment is to see how many short String and long string contains one character and eighty characters respectively can be appended to StringBuilder in one second.

3.1 procedures

1- One StringBuilder has been created and instantiated.
2-Start time has been created which gives the current time in milliseconds.
3- While loop has been created which includes appending process for one second.
4- End time has been created which gives the current time after finishing the appending process inside while loop for one second.
5- The run time has been calculated which is the difference between end time and start time.
6- *oneStringBuilderLength* is needed to calculate the average of String length and concatenations when one charcter only is appenended to String Builder. *eightyStringBuilderLength* variable is needed to calculate the average of String length when eighty characters are appended to StringBuilder and *eightyStringBuilderAppend* variable is needed to calculate concatenations when eighty charcters are appenended to String Builder.
7- Print the interested results for our experment which are the run time in milliseconds, concatenation and String length.
8- In main method a for loop has been created to calcolate five times how many concatenations and string length in one second and after that the average is taken for that process as is shown in the table 3 and 4.
9- Notice that two array lists has been created called legthlist and conList to add each values of the string length and concatenation resprctively in one second to use them later to find the length and concatenations average respectively .

3.2 Result

Table 3. StringBuilder with 1 character

ExNo	Time (Milliseconds)	Concatenation	Length
1	1000	62886739	62886739
2	1000	63351680	63351680
3	1000	74624419	74624419
4	1000	73028637	73028637
5	1000	72398385	72398385
Average	**1000**	**69257972**	**69257972**

Table 4. StringBuilder with 80 characters

ExNo	Time (Milliseconds)	Concatenation	Length
1	1066	2149581	171966480
2	1138	2149581	171966480
3	1058	1074791	85983280
4	1000	3190035	255202800
5	1684	537396	42991680
Average	**1000**	**1820276**	**145622144**

.

4. Integer Insertion Sort

The aim of this experment is to see how many integers can be sorted by insertion sort algorithm in one second.

4.1 procedures

1- An integer array arr has been created and by randomIntegers method we give it 10000 as length and 100000 random integrs to fill the array.
2- RunTime has been created and initiated.
3- while loop has been created with condition that run less or equal 1000 milliseconsm which equals to one seconds and when runtime becomes greater than 1000 milliseconds
4- inside while loop the start time has been created which gives us the current time in milliseconds and then the method *integerInsertionSort(arr)* is called and the end time has been created. The runTime is calculated by the difference between endtime and starttime.
5- Print runtime and array length inside the while loop which shows us the runtime is needed to sort the array with specific length.
6- If runtime for sorting a specific array with specific length is equal 999 or 1000 or 1001 milliseconds then stop.
7- If runtime is between 1001 and 1050 milliseconds then decrease the array length by 1 and the possible integers that fill it is 10 multiplies its current length and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the previous array length - 1.
8- If runtime is greater than 1050 milliseconds then decrease the array length by 1000 and the possible integers that fill it is 10 multiplies its current length and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the array length - 1000
9- If runtime is greater less than or equal 950 milliseconds then increase the array length by 1000 and the possible integers that fill it is 10 multiplies its current length and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the array length + 1000.
10- Else increase the array length by 1 and the possible integers that fill it is 10 multiplies its current length and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the array length + 1.
11- In the main method for loop has been created to find five times how many integers can be sorted in one second and then we find the average.

4.2 Result

Table 5. Integer Insertion Sort

ExNo	Time (Milliseconds)	ArrayLength
1	1000	90190
2	1000	91091
3	1000	90191
4	1000	90035
5	1000	90502
Average	**1000**	**90401**

5. String Insertion Sort

The aim of this experment is to see how many characters can be sorted by insertion sort algorithm in one second.

5.1 procedures

1- An String array arr has been created and by randomString method we give it 1000 as length.
2- RunTime has been created and initiated.
3- while loop has been created with condition that run less or equal 1000 milliseconsm which equals to one seconds and when runtime becomes greater than 1000 milliseconds
4- inside while loop the start time has been created which gives us the current time in milliseconds and then the method *stringInsertionSort(arr)* is called and the end time has been created. The runTime is calculated by the difference between endtime and starttime.
5- Print runtime and array length inside the while loop which shows us the runtime is needed to sort the array with specific length.
6- If runtime for sorting a specific array with specific length is equal 999 or 1000 or 1001 milliseconds then stop.
7- If runtime is between 1001 and 1050 milliseconds then decrease the array length by 1 and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the previous array length - 1.
8- If runtime is greater than 1050 milliseconds then decrease the array length by 100 and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the array length – 100.

9- If runtime is greater less than or equal 950 milliseconds then increase the array length by 100 and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the array length + 100.
10- Else increase the array length by 1 and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the array length + 1.
11- In the main method for loop has been created to find five times how many String consist of 10 characters can be sorted in one second and then we find the average.

5.2 Result

Table 6. String Insertion Sort

ExNo	Time (Milliseconds)	ArrayLength
1	1000	11413
2	1000	11305
3	1000	11506
4	1000	11401
5	1000	11502
Average	**1000**	**11425**

6. Integer Merge Sort

The aim of this experment is to see how many integers can be sorted by merge sort algorithm in one second.

4.1 procedures

1- An integer array arr has been created and by randomIntegers method we give it 100000 as length and 1000000 random integrs to fill the array.
2- RunTime has been created and initiated.
3- while loop has been created with condition that run less or equal 1000 milliseconsm which equals to one seconds and when runtime becomes greater than 1000 milliseconds
4- inside while loop the start time has been created which gives us the current time in milliseconds and then the method *integerMergeSort(arr)* is called and the end time has been created. The runTime is calculated by the difference between endtime and starttime.

5- Print runtime and array length inside the while loop which shows us the runtime is needed to sort the array with specific length.

6- If runtime for sorting a specific array with specific length is equal 999 or 1000 or 1001 milliseconds then stop.

7- If runtime is between 1001 and 1050 milliseconds then decrease the array length by 1 and the possible integers that fill it is 10 multiplies its current length and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the previous array length - 1.

8- If runtime is greater than 1050 milliseconds then decrease the array length by 1000 and the possible integers that fill it is 10 multiplies its current length and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the array length - 1000

9- If runtime is greater less than or equal 950 milliseconds then increase the array length by 10000 and the possible integers that fill it is 10 multiplies its current length and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the array length + 10000.

10- Else increase the array length by 1 and the possible integers that fill it is 10 multiplies its current length and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the array length + 1.

11- In the main method for loop has been created to find five times how many integers can be sorted in one second and then we find the average.

6.2 Result

Table 7. Integer Merge Sort

ExNo	Time (Milliseconds)	ArrayLength
1	1000	3529168
2	1000	3530053
3	1000	3528582
4	1000	3532141
5	1000	3528061
Average	**1000**	**3529601**

7. String Merge Sort

The aim of this experment is to see how many characters can be sorted by Merge sort algorithm in one second.

7.1 procedures

1- An String array arr has been created and by randomString method we give it 1000 as length.
2- RunTime has been created and initiated.
3- while loop has been created with condition that run less or equal 1000 milliseconsm which equals to one seconds and when runtime becomes greater than 1000 milliseconds
4- inside while loop the start time has been created which gives us the current time in milliseconds and then the method *stringMergeSort(arr)* is called and the end time has been created. The runTime is calculated by the difference between endtime and starttime.
5- Print runtime and array length inside the while loop which shows us the runtime is needed to sort the array with specific length.
6-If runtime for sorting a specific array with specific length is equal 999 or 1000 or 1001 milliseconds then stop.
7-If runtime is between 1001 and 1050 milliseconds then decrease the array length by 1 and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the previous array length - 1.
8- If runtime is greater than 1050 milliseconds then decrease the array length by 100 and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the array length – 100.
9- If runtime is greater less than or equal 950 milliseconds then increase the array length by 1000 and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the array length + 1000.
10- Else increase the array length by 1 and reset runtime to zero. When runtime is equal zero the sorting process will start again but the new array length is the array length + 1.
11- In the main method for loop has been created to find five times how many String consist of 10 characters can be sorted in one second and then we find the average.

7.2 Result

Table 8. String Merge Sort

ExNo	Time (Milliseconds)	ArrayLength
1	1000	720010
2	1000	730010
3	1000	730102
4	1000	730017
5	1000	733111
Average	**1000**	**728650**

8. Why StringBuilder is faster than the String concatenation + operation?

String concatenation + operator makes a copy for each concatenation which requires memory and time while StringBuider 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, String is added which means StringBuilder saves a copy when the data

becomes too big and that is why there is a big difference in the result.