David Schulz

CS 4981 021

3/18/2022

**Exercise 1: I/O Optimization**

**Input Optimization**

One Input per Line

1. Report on the runtime for each. Which mechanism worked faster?
   1. Scanner: 14,012,900 nanoseconds
   2. BufferedReader: 1,286,900 nanoseconds
   3. BufferedReader was faster by about 13,000,000 nanoseconds.
2. Research about each input method. Why is one faster than the other?
   1. BufferedReader is faster than Scanner because Scanner is used for parsing tokens from the contents of the stream while BufferedReader just reads the stream and doesn’t do any special parsing.

Multiple Inputs per Line

1. Report on the runtime for each. Which ran faster?
   1. Split: 22,032,300 nanoseconds
   2. StringTokenizer: 20,790,200 nanoseconds
   3. StringTokenizer was faster by about 1,300,000 nanoseconds.
2. Research about each input method. Why is one faster than the other?
   1. Every time you call split, it creates a new Pattern object and compiles an expression into a pattern. Since the String split builds a new Pattern every time, it is bound to be slower than StringTokenizer.
3. Given your experiences, which input mechanism would you recommend using for solving a problem as fast as possible? Why?
   1. Using BufferedReader with StringTokenizer would be the best for solving a problem quickly. The time to implement isn’t the fastest, but it’s still not complicated and the improvement in runtime is worth it. There are some implementations that have a slightly faster runtime, but they’re much more complicated to implement. As far as memory goes, BufferedReader has a slightly larger buffer than Scanner, but only by a few kb, and StringTokenizer uses less memory than split() because of the problem with making new Patterns every time.

Optimize

According to an article on GeeksForGeeks called “Fast I/O in Java in Competitive Programming”, the method using BufferedReader and StringTokenizer is already the most optimal method of reading input for the same reasons I listed in the previous question.

**Output Optimization**

Value Conversion

1. Report on the runtime for each. Which mechanism worked faster?
   1. String.valueOf(): 268,500 nanoseconds
   2. Integer.toString(): 259,400 nanoseconds
   3. Integer.toString() worked a little faster, but they are likely the same on average.
2. Research about each concatenation method. Why does each produce that runtime?
   1. Their average runtimes will be identical because String.valueOf() simply calls Integer.toString() in the background.

String Concatenation

1. Report on the runtime for each. Which mechanism worked faster?
   1. + Operator: 431,100 nanoseconds
   2. String.concat(): 244,600 nanoseconds
   3. StringBuffer: 543,600 nanoseconds
   4. StringBuilder: 521,100 nanoseconds
   5. String.concat() worked the fastest.
2. Research about each concatenation method. Why does each produce that runtime?
   1. The + operator uses StringBuilder under the hood, so its performance will be similar to StringBuilder’s. Also, StringBuffer and StringBuilder work almost the same way under the hood, with the only difference being that StringBuffer is synchronized. I read that String.concat() is actually supposed to perform the slowest, but it’s almost 2x faster than anything else in my implementations. This is likely because the performance gap becomes noticeable with much higher iterations, like in the millions.
3. Report on the runtime for each. Which mechanism worked faster?
   1. StringBuilder: 89,200 nanoseconds
   2. String.join(): 326,800 nanoseconds
   3. StringBuilder worked faster.
4. Research about each concatenation method. Why does each produce that runtime?
   1. Usually String.join() would be faster with much fewer iterations, but String.join must run through each input array several times to allocate the memory for the output string and then combine the strings, so running 1000 iterations slows it down, allowing StringBuilder to perform faster due to its ability to read and allocate memory more dynamically.

Buffered Output

1. Report on the runtime for each. Which mechanism worked faster?
   1. System.out.println(): 13,097,200 nanoseconds
   2. PrintWriter: 953,400 nanoseconds
   3. PrintWriter worked much faster.
2. Research about each output method. Why is one faster than the other?
   1. System.out.println is a PrintStream. All characters printed by a PrintStream are first converted into bytes using the platform's default character encoding. The PrintWriter class is used in situations that require writing characters rather than bytes. Also, allowing the PrintWriter to wait to flush until the end of the program meant the program didn’t need to pause to flush the output for every iteration. We can’t tell System.out.println to not flush after each iteration.