CompSci 201, L7: Runtime Efficiency

(Also more about String vs StringBuilder)
Logistics, Coming up

• Today 2/6
  • Project 1 Nbody due today
  • Project 2 Markov releasing tomorrow (due in 2 weeks)

• Wednesday 2/8
  • APT 3 due

• Next Monday 2/13
  • Midterm Exam 1
    • Covers everything through THIS week, up to and including asymptotic analysis / Big O
    • Example/Practice exams available this evening
Midterm Exams

See details on [course website assignments and grading page](#)

- 60 minutes, in-class exam.
- Short-answer problems. Reason about algorithms, data structures, code.
- Can bring 1 double sided reference sheet (8.5x11 in), write/type whatever notes you like.
- No electronic devices out during exam
Exam Grades and Missing Exams

• Three midterm exams scheduled (E1, E2, E3).
• Final exam has 3 corresponding parts (F1, F2, F3).
• Overall course exam grade is:
  \[ \text{AVG}( \max(E1, F1), \max(E2, F2), \max(E3, F3) ) \]

• Meaning the final exam serves:
  • As a makeup, if you need to miss a midterm, and/or
  • As an opportunity to demonstrate more learning, if
    you’re unhappy with your midterm score.
Problem Solving with Maps
Word Pattern Problem

Live Coding

Word Pattern Problem
Runtime Efficiency, an Empirical Look at String Concatenation
Two methods for repeated concatenation

```java
public static String repeatConcatA(int reps, String toConcat) {
    String result = new String();
    for (int i=0; i<reps; i++) {
        result += toConcat;
    }
    return result;
}

public static String repeatConcatB(int reps, String toConcat) {
    StringBuilder result = new StringBuilder();
    for (int i=0; i<reps; i++) {
        result.append(toConcat);
    }
    return result.toString();
}
```

methodA: Using String object and basic + operator

methodB: Using StringBuilder object and append method
Empirical timing experiment

Can see the code on gitlab here.

```java
public class StringConcatTiming {
    static final int NUM_TRIALS = 100;
    static final int REPS_PER_TRIAL = 1024;
    static final String TO_CONCAT = "201";

    public static void main(String[] args) {
        long totalTime = 0;
        for (int trial=0; trial<NUM_TRIALS; trial++) {
            long startTime = System.nanoTime();
            //repeatConcatA(REPS_PER_TRIAL, TO_CONCAT);
            repeatConcatB(REPS_PER_TRIAL, TO_CONCAT);
            long endTime = System.nanoTime();
            totalTime += (endTime - startTime);
        }
        double avgTime = (double)totalTime / NUM_TRIALS;
        System.out.printf("Avg time per trial is %f ms", avgTime*1E-6);
    }
}
```
Empirical results

![Graph showing average runtime in milliseconds against number of string concatenation repetitions for Method A (String) and Method B (StringBuilder)]

- **Method A [String] (ms)**
- **Method B [StringBuilder] (ms)**

The graph illustrates the performance comparison between Method A and Method B as the number of string concatenation repetitions increases. Method A shows a significant increase in runtime as the number of repetitions grows, whereas Method B maintains a relatively stable runtime.
Empirical results in more detail

<table>
<thead>
<tr>
<th>Reps</th>
<th>MethodA (ms)</th>
<th>MethodB (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.384</td>
<td>0.050</td>
</tr>
<tr>
<td>2048</td>
<td>1.136</td>
<td>0.061</td>
</tr>
<tr>
<td>4096</td>
<td>3.443</td>
<td>0.077</td>
</tr>
<tr>
<td>8192</td>
<td>12.244</td>
<td>0.099</td>
</tr>
<tr>
<td>16384</td>
<td>41.754</td>
<td>0.143</td>
</tr>
<tr>
<td>32768</td>
<td>147.719</td>
<td>0.207</td>
</tr>
</tbody>
</table>

Multiply reps by 2 multiplies runtime by 4. Quadratic complexity.

Multiply reps by 2 multiplies runtime by ~2. Linear complexity.
Empirical results in more detail

<table>
<thead>
<tr>
<th>Reps</th>
<th>MethodA ns/rep</th>
<th>MethodB ns/rep</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.375</td>
<td>0.048</td>
</tr>
<tr>
<td>2048</td>
<td>0.555</td>
<td>0.030</td>
</tr>
<tr>
<td>4096</td>
<td>0.841</td>
<td>0.019</td>
</tr>
<tr>
<td>8192</td>
<td>1.495</td>
<td>0.012</td>
</tr>
<tr>
<td>16384</td>
<td>2.548</td>
<td>0.009</td>
</tr>
<tr>
<td>32768</td>
<td>4.508</td>
<td>0.006</td>
</tr>
</tbody>
</table>

Runtime / rep increasing, greater than linear complexity.

Runtime / rep not increasing, at most linear complexity.
What’s going on? Documentation?

docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String

**Class String**

java.lang.Object
   java.lang.String

**All Implemented Interfaces:**
Serializable, CharSequence, Comparable<String>, Constable, ConstantDesc

```java
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc
```

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as Strings. Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings.
methodA revisited

```java
19     public static String repeatConcatA(int reps, String toConcat) {
20         String result = new String();
21         for (int i=0; i<reps; i++) {
22             result += toConcat;
23         }
24         return result;
25     }
```

How many characters will be copied per iteration if `toConcat == "201"`?

- i=0: 3
- i=1: 6
- i=2: 9
- ...
- On iteration i, need to copy 3*(i+1) characters!

String is immutable, line 22 creates a new string and copies result then `toConcat`.
How many total characters are copied? Algebra!

**methodA:** i goes from 0 to reps-1, copy $3(i+1)$ characters per iteration.

$$\sum_{i=0}^{reps-1} 3(i + 1) = 3(reps) + 3 \left( \sum_{i=0}^{reps-1} i \right)$$

$$= 3(reps) + 3 \left( \frac{reps}{2} \right) (0 + reps - 1)$$

$$= \frac{3}{2} (reps^2 + reps)$$

**Arithmetic series formula:**

$$\sum_{i=1}^{n} a_i = \left( \frac{n}{2} \right) (a_1 + a_n)$$
Abstracting, Intro to Big O Notation (Preview for next time)

• The $\frac{3}{2}$ in $\frac{3}{2} \text{reps}^2$ doesn’t tell us much about how the performance scales with the size of reps.

• Often, we use asymptotic notation, especially Big O notation to abstract away constants.

• For example: let $N = \text{reps}$, then we say that the asymptotic runtime complexity is $O(N^2)$.
  • If you ~double $N$, you ~quadruple the runtime
Two general Big O rules

1. Can drop constants
   • \(2N+3 \rightarrow O(N)\)
   • \(0.001N + 1,000,000 \rightarrow O(N)\)

2. Can drop lower order terms
   • \(2N^2+3N \rightarrow O(N^2)\)
   • \(N+\log(N) \rightarrow O(N)\)
   • \(2^N + N^2 \rightarrow O(2^N)\)
# Hierarchy of some common complexity classes

<table>
<thead>
<tr>
<th>Big O</th>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(2^N)$</td>
<td>Exponential</td>
<td>Calculate all subsets of a set</td>
</tr>
<tr>
<td>$O(N^3)$</td>
<td>Cubic</td>
<td>Multiply NxN matrices</td>
</tr>
<tr>
<td>$O(N^2)$</td>
<td>Quadratic</td>
<td>Loop over all pairs from N things</td>
</tr>
<tr>
<td>$O(N \log(N))$</td>
<td>Nearly-linear</td>
<td>Sorting algorithms</td>
</tr>
<tr>
<td>$O(N)$</td>
<td>Linear</td>
<td>Loop over N things</td>
</tr>
<tr>
<td>$O(\log(N))$</td>
<td>Logarithmic</td>
<td>Binary search a sorted list</td>
</tr>
<tr>
<td>$O(1)$</td>
<td>Constant</td>
<td>Addition, array access, etc.</td>
</tr>
</tbody>
</table>
How does StringBuilder work?

“Every string builder has a capacity. As long as the length of the character sequence contained in the string builder does not exceed the capacity, it is not necessary to allocate a new internal buffer. If the internal buffer overflows, it is automatically made larger.” - StringBuilder JDK 17 documentation.

• But how does it grow?

• Geometrically! Like ArrayList, HashMap, ...
  • Still linear amortized complexity, for same reasons
StringBuilder is like an ArrayList of characters

• Suppose we run the code:
StringBuilder() sb = new StringBuilder(3);
sb.append(“hi”);
sb.append(“ya”);
How many total characters are copied with a StringBuilder?

Suppose we start with capacity 3 and double when out of capacity, appending a length 3 string reps times...

\[ 3 \cdot \text{reps} + \sum_{i=0}^{\approx \log_2(3 \cdot \text{reps})} 2^i = 3 \cdot \text{reps} + 6 \cdot \text{reps} \]

\[ = 9 \cdot \text{reps} \]

The “good case” copies

From doubling and copying the array

Geometric series formula:

\[ \sum_{i=0}^{n} a r^i = a \frac{1 - r^{n+1}}{1 - r} \]
Memory/Runtime Tradeoff

```java
public static String repeatConcatB(int reps, String toConcat) {
    StringBuilder result = new StringBuilder();
    for (int i=0; i<reps; i++) {
        result.append(toConcat);
    }
    System.out.printf("String builder capacity is %d characters\n", result.capacity());
    System.out.printf("Result length is %d characters\n", result.length());
    return result.toString();
}
```

Final StringBuilder is using about 146k / 98k ~= 1.5 times as much memory as necessary. Very common tradeoff in data structures!
What’s the real difference between methodA and methodB?

- methodA: Copies roughly \( \frac{3}{2} (\text{reps}^2 - \text{reps}) \)
- methodB: copies roughly \( 9 \cdot \text{reps} \) characters.

<table>
<thead>
<tr>
<th>Reps</th>
<th>~MethodA char copies (millions)</th>
<th>MethodB char copies (millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1.5</td>
<td>0.009</td>
</tr>
<tr>
<td>2000</td>
<td>6</td>
<td>0.018</td>
</tr>
<tr>
<td>4000</td>
<td>24</td>
<td>0.036</td>
</tr>
<tr>
<td>8000</td>
<td>95</td>
<td>0.072</td>
</tr>
<tr>
<td>16000</td>
<td>383</td>
<td>0.144</td>
</tr>
<tr>
<td>32000</td>
<td>1535</td>
<td>0.288</td>
</tr>
</tbody>
</table>
WOTO

Go to duke.is/8qfjk

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!
How many total characters must be copied by the code on lines 8 and 9? Remember that Strings are immutable in Java. *

Suppose method A has linear complexity and takes 10 ms to run on an input of size N. About what would you expect the runtime to be for an input of size 2*N? *
Suppose method B has quadratic complexity and takes 10 ms to run on an input of size N. About what would you expect the runtime to be for an input of size 2*N?

- 10 ms
- 20 ms
- 40 ms
- 100 ms
Here is another String concatenation method. Suppose the input string s has a small number of characters, say 3. As a function of the parameter reps, how would you characterize the runtime complexity of the method? Hint: As a function of reps, how many total characters will be copied across all iterations of the loop? *

```java
public static String concatAlot(int reps, String s) {
    for (int i=0; i<reps; i++) {
        s += s;
    }
    return s;
}
```

☐ Constant
☐ Linear
☐ Quadratic
☐ Exponential