L8: Asymptotic (Big-O) Analysis

Alex Steiger
CompSci 201: Spring 2024
5/8/24

Logistics, Coming up

• Today, 2/7
  • APT 3 due

• Next Monday 2/12
  • Midterm Exam 1

• Next Wednesday 2/14
  • APT 4 due

Person in CS: Alan Turing

• 1912-1954 (died at 41)
• English, PhD at Princeton in 1938
• Mathematician, cryptographer, pioneering thinker in AI
  • "Father of modern computer science"
  • Turing machine – helped formalize what is computable
  • Cryptography work in WW2
• Prosecuted in 1952 for homosexuality
  • Given choice of chemical "treatment" or prison, took former
  • Died 2 years later of cyanide poisoning, circumstances debated
2

How many total characters must be copied by the code on lines 8 and 9? Remember that strings are immutable in Java. *

☐ 5
☐ 7
☐ 9
☐ 10
☐ 15
☐ 30

3

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? *

☐ 10 ms
☐ 20 ms
☐ 40 ms
☐ 100 ms

4

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
Asymptotic Analysis and Big O Notation

Runtime and memory

- Two most fundamental resources on a computer:
  - Processor cycles: Number of operations per second machine can perform
    - (2 GHz = 2 billion operations per second).
  - Memory: space for storing variables, data, etc.
    - (esp. working memory, a.k.a. cache and RAM)

- We will mostly focus on runtime complexity
  - Often comes at expense of memory, e.g., HashMap
  - Start by reasoning about empirical runtimes, but...

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?
Problem with empirical runtimes

How do we measure efficiency of the code apart from the machine?

- Let N be the size of the input
  - For some int[] ar, N could be ar.length

- Count T(N) = number of constant time operations in the code as a function of N.

- Reason about how T(N) grows when N becomes large.
  - "Asymptotic" (in the limit) notation

Reminder: What is constant time?

- Running time does not depend on size of the input.
  - If ~1 ms to .get() when ArrayList has 1,000 elements?
  - Then ~1 ms to .get() when ArrayList has 1,000,000 values.

- Other constant time operations might be a very different constant.
  - Adding 2+2 might be faster than .get(), but both are constant.
Constant Time Examples

- Index into an array (ar[0] or ar[201])
- Arithmetic (+, -, *, /, %, etc.)
- Primitive comparison <, ==, etc.
- Accessing an object attribute (e.g. .length)
- ArrayList .get(), .size(), .add() [to end, amortized]

- Non-constant time usually has a loop or method call, may depend on implementation of data structures at hand

Big-O (limit definition)

- Given N (for example, the size of the input)
- Function T(N) (for example, the number of constant time operations in the code)

Definition (big O notation).
T(N) is $O(g(N))$ if \( \lim_{N \to \infty} \frac{T(N)}{g(N)} \leq c \) for some constant \( c \) that does not depend on \( N \).

In other words: \( T(N) \) is $O(g(N))$ if it is at most a constant factor times slower than \( g(N) \) for large input \( N \).

Two general rules

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

2. Can drop lower order terms
   - \( 2N^2 + 3N \to O(N^2) \)
   - \( N + \log(N) \to O(N) \)
   - \( 2^N + N^2 \to O(2^N) \)
Hierarchy of some common complexity class

<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>$O(N^2)$</td>
<td>Cubic</td>
<td>Multiply NxN matrices</td>
</tr>
<tr>
<td>$O(N^3)$</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>

Some common complexity classes and their growth

<table>
<thead>
<tr>
<th>N</th>
<th>$O(\log(N))$</th>
<th>$O(N)$</th>
<th>$O(N^2)$</th>
<th>$O(N^3)$</th>
<th>$O(2^N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>9</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>16</td>
<td>256</td>
<td>65536</td>
<td>65k</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>32</td>
<td>1k</td>
<td>32k</td>
<td>4.2E+9</td>
</tr>
<tr>
<td>64</td>
<td>7</td>
<td>64</td>
<td>4k</td>
<td>262k</td>
<td>1.8E+19</td>
</tr>
</tbody>
</table>

If you double N...
- $O(\log(N))$ adds ~1
- $O(N)$ roughly doubles
- $O(N^2)$ roughly quadruples
- $O(N^3)$ roughly multiples by 8
- $O(2^N)$ squares each time

Relation to Empirical Timing and Lower Order Terms

<table>
<thead>
<tr>
<th>N</th>
<th>$N^2 + 19N + 200$</th>
<th>factor increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>490</td>
<td>NA</td>
</tr>
<tr>
<td>20</td>
<td>980</td>
<td>2.00</td>
</tr>
<tr>
<td>40</td>
<td>2560</td>
<td>2.61</td>
</tr>
<tr>
<td>80</td>
<td>8120</td>
<td>3.17</td>
</tr>
<tr>
<td>160</td>
<td>38840</td>
<td>3.55</td>
</tr>
<tr>
<td>320</td>
<td>108680</td>
<td>3.77</td>
</tr>
<tr>
<td>640</td>
<td>421960</td>
<td>3.88</td>
</tr>
<tr>
<td>1280</td>
<td>1626220</td>
<td>3.94</td>
</tr>
<tr>
<td>2560</td>
<td>6602440</td>
<td>3.97</td>
</tr>
<tr>
<td>5120</td>
<td>26311880</td>
<td>3.99</td>
</tr>
<tr>
<td>10240</td>
<td>105052360</td>
<td>3.99</td>
</tr>
<tr>
<td>20480</td>
<td>419819720</td>
<td>4.00</td>
</tr>
</tbody>
</table>

Asymptotic analysis describe behavior in the limit as N becomes large. Lower order terms may dominate at small input sizes.
2

\[ n^2 + n \log(n) + (\log(n))^2 \text{ is...} \]

- \( O(n^2) \)
- \( O(n \log(n)) \)
- \( O((\log(n))^2) \)

3

\[ n^2 + 2^n \text{ is...} \]

- \( O(n^2) \)
- \( O(2^n) \)

4

\[ \log(n^2) \text{ is...} \]

\[ = 2 \log(n) = O(\log(n)) \]

- \( O(n^2) \)
- \( O(n) \)
- \( O((\log(n))^2) \)
- \( O(\log(n)) \)
Big-Oh for Runtime: Algorithms & Code

- What is the runtime complexity of \texttt{stuff(n)}?
- How many times does the loop iterate?
  - In terms of \( n \), the parameter
- Loop body is \( O(1) \)?
  - Constant time
  - Independent of \( n \)
  - Add \( n \) same as add 1

\begin{verbatim}
public int stuff(int n) {
    int sum = 0;
    for(int k = 0; k < n; k++) {
        sum += k;
    }
    return sum;
}
\end{verbatim}

Linear, \( O(n) \)
General strategy for determining Big-O runtime complexity

Most general: Determine T(N), the number of constant time operations as a function of the size of the input, N. Then simplify using Big-O.

Practically, covers common cases:
1. For each line of code, label:
   a) Complexity of that line, and
   b) Number of times the line is executed
2. Add up over all lines, multiplying the two labels

Nested loop example

What about the big-O runtime complexity of this code as a function of n?

<table>
<thead>
<tr>
<th></th>
<th>Complexity</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>O(1)</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>O(1)</td>
<td>n</td>
</tr>
<tr>
<td>9</td>
<td>O(1)</td>
<td>?</td>
</tr>
<tr>
<td>10</td>
<td>O(1)</td>
<td>?</td>
</tr>
<tr>
<td>13</td>
<td>O(1)</td>
<td>1</td>
</tr>
</tbody>
</table>

How many times does line 10 execute?

\[
\text{In total? } 1 + 2 + \cdots + (n - 2) + (n - 1) \approx \frac{n^2}{2} \text{ is } O(n^2) \text{ iterations}
\]
Nested loop example

Putting it together:

```java
public int nested(int n) {
    int result = 0;
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++)
            result += i;
    }
    return result;
}
```

Total runtime complexity: $(1) + (n) + (n^2) + (n^2) + (1)$ is $O(n^2)$

Not all nested loops are quadratic

What about the big-O runtime complexity of this code as a function of $n$?

```java
public int nested(int n) {
    int result = 0;
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++)
            result += i;
    }
    return result;
}
```

Total runtime complexity: $(1) + (n) + (200n) + (1)$ is $O(n)$

Reminder: $200n$ is 200 times slower than $n$, but their runtimes both scale linearly.

Not all loops are nested

What about the big-O runtime complexity of this code as a function of $n$?

```java
public int parallel(int n) {
    int result = 0;
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++)
            result += i;
    }
    return result;
}
```

Total runtime complexity: $(1) + (4n) + (1)$ is $O(n)$
Not all loops increment by 1

Big-O Runtime complexity of \text{calc}(N) is...

- How many times does the loop iterate?
- Concrete to abstract: \text{calc}(16), \text{calc}(32), ...
- Inside loop? \( \mathcal{O}(1) \) operations

\begin{verbatim}
public int calc(int n) {
    int sum = 0;
    for(int k=1; k < n; k = 2) {
        sum += k;
    }
    return sum;
}
\end{verbatim}

Generalizing: Concrete to Abstract

\begin{table}
\begin{tabular}{|c|c|}
\hline
\( N \) & \# loop iterations \\
\hline
1 & 0 \\
2 & 1 \\
4 & k=1,2... iters \\
8 & k=1,2,4... iters \\
16 & 5 iterations \\
32 & \( k=1,2,4,8,16... \) iters \\
33 & 6 iterations \\
63 & 6 iterations \\
\hline
\end{tabular}
\end{table}

\[ \mathcal{O}(\log(N)) \]

Accounting for iteration and non-constant time operations

What about the big-O runtime complexity of this code as a function of \( n = \text{words.size()} \)?

Total: Make \( n \) calls to \( \mathcal{O}(n) \) contains: \( \mathcal{O}(n^2) \)
Exponential time algorithm?

Problem from previous WOTO: What is the runtime complexity of `concatAllot` as a function of `reps`?

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

Runtime of line 14 is $O(s.length())$. And this doubles every iteration through the loop.

Examine how the length of $s$ grows by iterations.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>$s.length()$</th>
<th>$O(1)$ operations (char copies)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (input s)</td>
<td>1 (suppose)</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>... reps-1</td>
<td>$2^{reps-1}$</td>
<td>$2(2^{reps-1}) = 2^{reps}$</td>
</tr>
</tbody>
</table>

So runtime has to be at least $2^{reps}$, exponential complexity!
What is the big O runtime complexity of the moreLooping method as a function of the parameter n? *

- O(1)
- O(log(n))
- O(n)
- O(n^2)
- O(n^3)
- O(n^2log(n))

What is the big O runtime complexity of the reverse method as a function of n where n is the size of the list parameter input? * added s to the front of the list.

```java
public static List<String> reverse(List<String> input) {
    ArrayList<String> result = new ArrayList<>();
    for (String s : input) {
        result.add(0, s);
    }
    return result;
}
```