CompSci 201, L8: Asymptotic (Big-O) Analysis
Logistics, Coming up

• Today, Wednesday 9/21
  • APT 3 due
  • Big O / Asymptotic Analysis

• Friday 9/23
  • Discussion: Maps, Big O, hashCode

• Monday 9/26
  • Memory, Pointers, LinkedList
Looking ahead...

• Project 2: Markov available
  • Have two weeks, recommend starting early

• Midterm Exam 1
  • Next Wednesday, 9/28
  • Covers everything through THIS week, up to asymptotic analysis / Big O
  • Does NOT cover linked list, next major topic
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
• No communication about exam on day-of
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.
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...
Problem with empirical runtimes

Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Transistor count

Same code that takes 1 min. in 1990 takes

- ~2 s in 2000?
- ~63 ms in 2010?
- ~2 ms in 2020?
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 .put() when map has 1,000 elements?
  • Then ~1 ms to .put() when map has 1,000,000 values.

• Other constant time operations might be a very different constant.
  • Adding 2+2 is much faster than .put() in a HashMap, but both are constant.
Constant Time Examples

• Index into an array (ar[0] or ar[201])
• Arithmetic (+, -, *, /, %, etc.)
• Comparison <, ==, etc.
• Access an object attribute (e.g. .length)
• ArrayList .get(), .size(), .add() [to end, amortized]
• HashMap/Set .get(), .put() [amortized]

• Non-constant time usually has a loop or method call, may depend on data structure implementation
Why we Big-O Notation

- $O(n^2)$ means at most quadratic growth in the limit, up to constant factors
  - Runtimes increase at same rate up to constants
  - Ignore coefficients and low-order terms

$y = x^2 \sim y = x^2 - 6x + 9 \sim y = 3x^2 + 4x$
Big-O (limit definition)

• Let N be the size of the input
• T(N) = number of constant time operations in the code

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

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 \( N \).
Expectations revisited

• Not expected: Formal proofs/derivations using the formal definition.
  • Might do that in later courses!

• Expected: Ability to reason in two directions:
  • Given an algorithm/code, determine the Big O runtime complexity, and
  • Given a Big O runtime complexity, be able to reason about expected scalability/timing in practice.
Two general 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 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>$O(N^3)$</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>
Some common complexity classes and their growth

<table>
<thead>
<tr>
<th>N</th>
<th>O(log(N))</th>
<th>O(N)</th>
<th>O(N²)</th>
<th>O(N³)</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>8</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>4k</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(N²) roughly quadruples
- O(N³) roughly multiples by 8
- O(2^N) squares each time

Note: Turning these into runtimes depends on your machine.
$10^9$ Instructions/second means ...

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\log N$ time (s)</th>
<th>$N$ time (s)</th>
<th>$N \log N$ time (s)</th>
<th>$N^2$ time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3E-9</td>
<td>1E-8</td>
<td>3.3E-8</td>
<td>0.0000001</td>
</tr>
<tr>
<td>100</td>
<td>7E-9</td>
<td>1E-7</td>
<td>6.64E-7</td>
<td>0.0001</td>
</tr>
<tr>
<td>1,000</td>
<td>1E-8</td>
<td>1E-6</td>
<td>0.00001</td>
<td>0.001</td>
</tr>
<tr>
<td>10,000</td>
<td>1.3E-8</td>
<td>0.00001</td>
<td>0.0001329</td>
<td>0.102</td>
</tr>
<tr>
<td>100,000</td>
<td>1.7E-8</td>
<td>0.0001</td>
<td>0.001661</td>
<td>10.008</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.000000002</td>
<td>0.001</td>
<td>0.0199</td>
<td>16.7 min</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>0.000000003</td>
<td>1.002</td>
<td>65.8</td>
<td>31.8 years</td>
</tr>
</tbody>
</table>
WOTO

Go to duke.is/458uk

Not graded for correctness, just participation.

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

But do talk to your neighbors!
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 += 1) {
        sum += n;
    }
    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:

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?

```
6  public int nested(int n) {
7      int result = 0;
8      for (int i=0; i<n; i++) {
9          for (int j=0; j<i; j++) {
10             result += 1;
11         }
12      }
13      return result;
```

<table>
<thead>
<tr>
<th>Line</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?
Nested loop example

How many times does line 10 execute?

```java
public int nested(int n) {
    int result = 0;
    for (int i=0; i<n; i++) {
        for (int j=0; j<i; j++) {
            result += 1;
        }
    }
    return result;
}
```

<table>
<thead>
<tr>
<th>when i is</th>
<th>Line 10 executes this many times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n-2</td>
<td>n-2</td>
</tr>
<tr>
<td>n-1</td>
<td>n-1</td>
</tr>
</tbody>
</table>

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

Putting it together:

```
public int nested(int n) {
    int result = 0;
    for (int i=0; i<n; i++) {
        for (int j=0; j<i; j++) {
            result += 1;
        }
    }
    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 \( n \)?

```
16       public int nested2(int n) {
17             int result = 0;
18             for (int i=0; i<n; i++) {
19                 for (int j=0; j<100; j++)
20                     result += 1;
21             }
22         }
23         return result;
24     }
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Complexity</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>O(1)</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>O(1)</td>
<td>( n )</td>
</tr>
<tr>
<td>19</td>
<td>O(1)</td>
<td>100n</td>
</tr>
<tr>
<td>20</td>
<td>O(1)</td>
<td>100n</td>
</tr>
<tr>
<td>23</td>
<td>O(1)</td>
<td>1</td>
</tr>
</tbody>
</table>

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++) {
        result += 1;
    }
    for (int i=0; i<n; i++) {
        result += 1;
    }
    return result;
}
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Complexity</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>O(1)</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>O(1)</td>
<td>n</td>
</tr>
<tr>
<td>31</td>
<td>O(1)</td>
<td>n</td>
</tr>
<tr>
<td>33</td>
<td>O(1)</td>
<td>n</td>
</tr>
<tr>
<td>34</td>
<td>O(1)</td>
<td>n</td>
</tr>
<tr>
<td>36</td>
<td>O(1)</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Big-O Runtime complexity of calc(N) is...

• How many times does the loop iterate?
  • Concrete to abstract: calc(16), calc(32), ...
• Inside loop? O(1) operations

```java
    public int calc(int n) {
        int sum = 0;
        for(int k=1; k < n; k *= 2) {
            sum += k;
        }
        return sum;
    }
```
Generalizing: Concrete to Abstract

<table>
<thead>
<tr>
<th>N</th>
<th># loop iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>k=1,2...2 iters</td>
</tr>
<tr>
<td>8</td>
<td>k=1,2,4...3 iters</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th># loop iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>5 iterations</td>
</tr>
<tr>
<td>32</td>
<td>k=1,2,4,8,16..5 iters</td>
</tr>
<tr>
<td>33</td>
<td>6 iterations</td>
</tr>
<tr>
<td>63</td>
<td>6 iterations</td>
</tr>
</tbody>
</table>

```java
public int calc(int n) {
    int sum = 0;
    for(int k=1; k < n; k *= 2) {
        sum += k;
    }
    return sum;
}
```

$O(\log(N))$
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 \( O(n) \) contains: \( O(n^2) \)
Exponential time algorithm?

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

```java
12    public static String concatAlot(int reps, String s) {
13        for (int i=0; i<reps; i++) {
14            s += s;
15        }
16        return s;
17    }
```

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

Examine how the length of s grows by iterations.
Exponential time algorithm?

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

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

<table>
<thead>
<tr>
<th>Iteration</th>
<th><code>s.length()</code></th>
<th><code>O(1)</code> 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>...</td>
<td>...</td>
<td>...</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!
WOTO

Go to duke.is/97k32

Not graded for correctness, just participation.

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

But do talk to your neighbors!