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

• Today, 2/8
  • APT 3 due

• Next Monday 2/13
  • Midterm exam 1

• Next Wednesday 2/15
  • 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
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

<table>
<thead>
<tr>
<th>Year</th>
<th>Transistor Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970</td>
<td>1,000,000</td>
</tr>
<tr>
<td>1972</td>
<td>10,000,000</td>
</tr>
<tr>
<td>1974</td>
<td>100,000,000</td>
</tr>
<tr>
<td>1976</td>
<td>1,000,000,000</td>
</tr>
<tr>
<td>1978</td>
<td>10,000,000,000</td>
</tr>
<tr>
<td>1980</td>
<td>100,000,000,000</td>
</tr>
<tr>
<td>1982</td>
<td>1,000,000,000,000</td>
</tr>
<tr>
<td>1984</td>
<td>10,000,000,000,000</td>
</tr>
</tbody>
</table>

OurWorldInData.org – Research and data to make progress against the world’s largest problems.
Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

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 \textit{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.
- Access 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 data structure implementation
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

\[
\lim_{N \to \infty} \frac{T(N)}{g(N)} \leq c \quad \text{for some constant } c \quad \text{that does not depend on } N.
\]

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 \( N \).
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 <em>pairs</em> 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>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(N^2) roughly quadruples
- O(N^3) roughly multiples by 8
- O(2^N) squares each time

Note: Turning these into runtimes depends on your machine.
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>28840</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>1662920</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><strong>4.00</strong></td>
</tr>
</tbody>
</table>

Asymptotic analysis describes behavior *in the limit as n becomes large*, lower order terms may dominate at small input sizes.

Looks linear?  
Looks quadratic?
WOTO

Go to duke.is/6ezc8

Not graded for correctness, just participation.

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

But do talk to your neighbors!
L08-WOTO1-BigO

1

NetID *

2

n^2 + nlog(n) + (log(n))^2 is... *

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

\(n^2 + 2^n\) is... *

- \(\mathcal{O}(n^2)\)
- \(\mathcal{O}(2^n)\)

4

\(\log(n^2)\) is... *

- \(\mathcal{O}(n^2)\)
- \(\mathcal{O}(n)\)
- \(\mathcal{O}((\log(n))^2)\)
- \(\mathcal{O}(\log(n))\)
Suppose you time an algorithm for different values of $N$ and get the results shown in the table. What is the best characterization of the asymptotic runtime complexity observed in the data?

<table>
<thead>
<tr>
<th>$N$</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.03</td>
</tr>
<tr>
<td>200</td>
<td>0.08</td>
</tr>
<tr>
<td>400</td>
<td>0.24</td>
</tr>
<tr>
<td>800</td>
<td>0.80</td>
</tr>
<tr>
<td>1600</td>
<td>2.87</td>
</tr>
<tr>
<td>3200</td>
<td>10.85</td>
</tr>
<tr>
<td>6400</td>
<td>42.18</td>
</tr>
<tr>
<td>12800</td>
<td>166.28</td>
</tr>
</tbody>
</table>

- $O(N^3)$
- $O(N^2)$
- $O(N)$
- $O(\log(N))$
- $O(1)$
Suppose you time an algorithm for different values of N and M and get the results shown in the table. What is the best characterization of the asymptotic runtime complexity observed in the data? *

<table>
<thead>
<tr>
<th>N</th>
<th>M</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>0.81</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>1.21</td>
</tr>
<tr>
<td>100</td>
<td>400</td>
<td>2.01</td>
</tr>
<tr>
<td>200</td>
<td>100</td>
<td>1.11</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>1.51</td>
</tr>
<tr>
<td>200</td>
<td>400</td>
<td>2.31</td>
</tr>
<tr>
<td>400</td>
<td>100</td>
<td>1.71</td>
</tr>
<tr>
<td>400</td>
<td>200</td>
<td>2.11</td>
</tr>
<tr>
<td>400</td>
<td>400</td>
<td>2.91</td>
</tr>
</tbody>
</table>

- O(N)
- O(M)
- O(NM)
- O(N+M)
Big-Oh for Runtime: Algorithms & Code

- What is the runtime complexity of $\text{stuff}(n)$?
- 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

```
public int stuff(int n) {
    int sum = 0;
    for(int k=0; k < n; k += 1) {
        sum += n;
    }
    return sum;
}
```

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?

```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>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?

```
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:

```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>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>O(n²)</td>
</tr>
<tr>
<td>10</td>
<td>O(1)</td>
<td>O(n²)</td>
</tr>
<tr>
<td>13</td>
<td>O(1)</td>
<td>1</td>
</tr>
</tbody>
</table>

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?

public int nested2(int n) {
    int result = 0;
    for (int i=0; i<n; i++) {
        for (int j=0; j<100; j++)
            result += 1;
    }
    return result;
}

<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

```
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=1,2,4...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>

```
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 \( n = \text{words.size()} \)?

```java
public ArrayList<String> uniqueWords (ArrayList<String> words) {
    ArrayList<String> unique = new ArrayList<>();
    for (String w : words) {
        if (!unique.contains(w)) {
            unique.add(w);
        }
    }
    return words;
}
```

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>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>...</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/pu24z

Not graded for correctness, just participation.

Try to answer without looking back at slides and notes.

But do talk to your neighbors!
What is the big O runtime complexity of the keepHalving method as a function of the parameter n?

- O(1)
- O(log(n))
- O(n)
- O(n^2)
- O(n^3)
- O(2^n)
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(2^n)
What is the big O runtime complexity of the reverse method as a function of \( n \) where \( n \) is the \( \text{size()} \) of the List parameter input? add(0, s) adds 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;
}
```

- O(1)
- O(log(n))
- O(n)
- O(n^2)
- O(n^3)
- O(2^n)
Runtime complexity of composed methods

• Runtime complexity of \( \text{stuff(stuff(n))} \)?

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

• Value returned by \( \text{stuff(n)} \) is \( n^2 \).

• Runtime complexity of \( \text{stuff(n^2)} \)?

• \text{stuff} has linear runtime complexity, so \( \text{stuff(n^2)} \) is \( O(n^2) \)
Composing methods general

• Given two methods:

```java
public static int outer (int n) {
public static int inner(int n) {
```

• What is the runtime complexity of the following?

```java
int result = outer(inner(n));
```

Running this code is equivalent to...

```java
int innerValue = inner(n);
int result = outer(innerValue);
```
Composing methods general

- Given two methods:

  ```java
  public static int outer (int n) {
  public static int inner(int n) {
  ```

- What is the runtime complexity of the following?

  ```java
  int result = outer(inner(n));
  ```

  Three steps: Runtime complexity is 1+3.
  1. Calculate runtime complexity of inner(n)
  2. Calculate value returned by inner(n)
  3. Calculate runtime complexity of outer() on value from step 2.
Composing methods example

```java
int result = outer(inner(n));
```

1. Runtime complexity of `inner(n)` is $O(1)$
2. Value returned by `inner(n)` is $O(n^2)$
3. Runtime complexity of `outer(n^2)` is $O(\log(n^2))$

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

Most of the “work” done executing `outer`
Another composition example

```java
int result = outer(inner(n));
```

1. Runtime complexity of `inner(n)` is now $O(n)$
2. Value returned by `inner(n)` is still $O(n^2)$
3. Runtime complexity of `outer(n^2)` is still $O(\log(n^2))$

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

Now most of the “work” done executing `inner`