Big-O Notation
Program Efficiency &
Introduction to Complexity Theory
Finally!

So, what is computer science?

Is it the process of design and implementation of computer programs?
No, that’s software engineering.
Computer science is the study of theory of computational systems.

We’ve just seen two rather different implementations for list: SimpleArrayList and SimpleLinkedList. These are analogous to the java.util.ArrayList and java.util.LinkedList classes in the standard library. Why does the standard Java library have two implementations? Why not just pick the “better” implementation and be done with it?

As we’ll see, the choice of “better” depends on how the list is used.

In this section, we’re going to evaluate those two implementations and characterize the differences.
Overview

- Measuring time and space used by algorithms
- Machine-independent measurements
- Costs of operations
- Asymptotic complexity – O( ) notation and complexity classes
- Comparing algorithms
- Performance tuning
“Which one is better?”

It sounds like such a simple, straight-forward question, doesn’t it?

“Which one is better?”

Lurking in the shadows are a number of real issues. Central among them: What metric should we use?

This has a number of aspects:
• What should we measure it?
• How shall we measure it?
• Does that measurement make sense?
• Is that measurement important in making an informed choice?
One approach is to see which algorithm uses less resources.

Of course, there are more questions:
• Which resource(s) should we measure?
• If we use multiple metrics, how do we combine them?

For this discussion, we will focus on execution time: how long does it take something to run?

This is a very commonly used metric. There was a time that space was a primary consideration. No longer, RAM is cheap and the operating system can use (directly manipulate) gobs of memory.
Do we measure time using a stop watch?

Let’s say we have a very accurate stop watch, and that we can programmatically control the stop watch, so we aren’t limited by human reaction time, which is notoriously variable.

Wait, we have this. We can use `System.nanoTime` to report values from the system clock in nanosecond increments. We can measure elapsed time by making two calls to `nanoTime`, one immediately before the work and one immediately after. A simple subtraction will give us the elapsed time.

Brilliant! … or not.

What’s wrong with this picture?

Example

- What is the running time of the following method?
  ```java
  // Return the sum of the elements in array.
  double sum(double[ ] data) {
    double ans = 0.0;
    for (int k = 0; k < data.length; k++) {
      ans = ans + data[k];
    }
    return ans;
  }
  ```

- How do we analyze this?
- What does the question even mean?
What would affect the elapsed time measurement?

• Characteristics of the computer: amount of RAM, clock speed, etc.
• The other programming running concurrently on the machine
• Background processing, like participating in a network
• Etc., etc., etc. (also, sometimes written &c. [fodder: how is &c the same as etc?]

Ok, so elapsed time may not be the best measurement, but we’re looking at execution
time. So …
Notice that this definition avoids the traps we outlined on the previous page.

Let’s go back and look at that algorithm again.

Did you look at it?

It just works through the array, visiting each element, creating a running total.

Now, in an absolute sense it would be more work for the processor to total doubles rather than ints, just because of the internal representation of these two data types. But, that’s unimportant for this analysis. The time to run sum is going to be essentially proportional to the size of the array.
Here are some examples of operations which we will consider having a cost of one.
Similarly, we can talk about operations which will have no runtime impact. These are operations that take compile-time and/or class initialization time.

Yes, in an odd sort of way, class initialization (loading the class into the JVM and getting it ready to be used) falls into a separate category.

Notice that casts between int and char are zero-cost operations. However, casts between int and double would not be zero-cost; instead they would be constant-time operations, like addition. Notice that this applies even if the cast is “invisible” because of widening.

[fodder: Why the difference between int/char and int/double casting?]
[more fodder: what about float/double casting … constant-time or zero-time? Explain.]
Sequences of Statements

- Cost of
  \[ S_1; S_2; \ldots; S_n \]
  is sum of the costs of \( S_1 + S_2 + \ldots + S_n \)

This is pretty straight forward.
So, the cost of an if statement is normally taken to be the max cost of S1 or S2 (plus cost of evaluating the condition)
As you might expect, this is a combination of a sequence of statements and a conditional.

What about the “jump” back to the top of the loop? Well, that would be a constant-time operation.
Whoa! Expensive stuff here.

Ok, not really expensive, but there are a number of pieces that go into determining the cost of a method call.

Notice that this definition will handle things like nested method calls, since “evaluating the argument” will recursively invoke this notion of calling a method to calculated the time expense.

Make sense?
Exercise

- Analyze the running time of printMultTable
- Pick the problem size
- Count the number of steps

```java
// print multiplication table with // print row r with length n of a
// n rows and columns multiplication table
void printMultTable(int n) {
    void printRow(int r, int n) {
        for (int k=0; k<=n; k++) {
            for (int k = 0; k <= r; k++) {
                System.out.print(r*k + " ");
            }
        }
        System.out.println();
    }

    for (int k=0; k<=n; k++) {
        printRow(k, n);
    }
}
```

You have sufficient information now.

How expensive is a call to printMultTable as a function of n?
Analysis

- Yes, you really should try to figure this out.
- Here is the formula to calculate “triangle” numbers.
  \[ \sum_{j=0}^{n} j = 0 + 1 + 2 + 3 + \ldots + n = \frac{1}{2} n(n + 1) \]

Notice that this formula lets us derive the magic number for a square of order \( n \), \( \frac{1}{2} n(n^2+1) \)
[fodder: what is the derivation?]
Comparing Algorithms

- Suppose we analyze two algorithms and get these times (numbers of steps):
  - Algorithm 1: $37n + 2n^2 + 120$
  - Algorithm 2: $50n + 42$

How do we compare these? What really matters?

- Answer: In the long run, the thing that is most interesting is the cost as the problem size $n$ gets large
- What are the costs for $n=10$, $n=100$; $n=1,000$; $n=1,000,000$?
  - Computers are so fast that how long it takes to solve small problems is rarely of interest

<table>
<thead>
<tr>
<th></th>
<th>n=10</th>
<th>n=100</th>
<th>n=1,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 1</td>
<td>690</td>
<td>23,820</td>
<td>2,037,170</td>
</tr>
<tr>
<td>Algorithm 2</td>
<td>542</td>
<td>5,042</td>
<td>50,042</td>
</tr>
</tbody>
</table>

Those initial examples of $n=10$, $n=100$ are nothing. Things start getting interesting at $n=1,000$

What happens when $n=1,000,000$?

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 1</td>
<td>2,000,037,000,120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algorithm 2</td>
<td>50,000,042</td>
<td></td>
<td>~40,000 times bigger</td>
</tr>
</tbody>
</table>

The critical piece to notice here is that we are concerned about how an algorithm responds as the problem becomes huge.
You might wonder if that’s a real concern.

Let’s take a little stroll down memory lane.

Think about the changes in computing over the past few decades … ok, I realize that more or less accounts for the entire history of electronic computing, but still …

When the IBM PC made its debut in summer of 1981, it was configured with 16K of RAM and one (optionally, two) 5¼-inch floppy drive(s), single density, 360K per disk. A couple years later in 1983 with the introduction of the IBM XT, you could get a machine with an internal hard drive … 10 Megabytes of storage! (How could you ever have enough data to fill up all that space?)

Main frames and super computers were crunching “big” problems back then … but that doesn’t even start to compare with the amounts of data that we routinely handle these days.

Wikipedia contains terabytes ($10^{12}$) of data.
YouTube reports that 300 hours of video are uploaded every minute.
The estimate is that, as of 2012, over 2.5 exabytes ($10^{18}$) of data are created every day.

N = 1,000,000 … meh, that’s child’s play, these days.
So, let’s see how the value of various functions change depending on changes in input.

For most of this table, each new row is a doubling in the size of $N$. 

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\log_2 N$</th>
<th>$5N$</th>
<th>$N \log_2 N$</th>
<th>$N^2$</th>
<th>$2^N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>40</td>
<td>24</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>80</td>
<td>64</td>
<td>256</td>
<td>65536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>160</td>
<td>160</td>
<td>1024</td>
<td>$\sim 10^9$</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>320</td>
<td>384</td>
<td>4096</td>
<td>$\sim 10^{19}$</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>640</td>
<td>896</td>
<td>16384</td>
<td>$\sim 10^{38}$</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>1280</td>
<td>2048</td>
<td>65536</td>
<td>$\sim 10^{76}$</td>
</tr>
<tr>
<td>10000</td>
<td>13</td>
<td>50000</td>
<td>$10^5$</td>
<td>$10^8$</td>
<td>$\sim 10^{3010}$</td>
</tr>
</tbody>
</table>

So, let’s see how the value of various functions change depending on changes in input.

For most of this table, each new row is a doubling in the size of $N$. 

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\log_2 N$</th>
<th>$5N$</th>
<th>$N \log_2 N$</th>
<th>$N^2$</th>
<th>$2^N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>40</td>
<td>24</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>80</td>
<td>64</td>
<td>256</td>
<td>65536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>160</td>
<td>160</td>
<td>1024</td>
<td>$\sim 10^9$</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>320</td>
<td>384</td>
<td>4096</td>
<td>$\sim 10^{19}$</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>640</td>
<td>896</td>
<td>16384</td>
<td>$\sim 10^{38}$</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>1280</td>
<td>2048</td>
<td>65536</td>
<td>$\sim 10^{76}$</td>
</tr>
<tr>
<td>10000</td>
<td>13</td>
<td>50000</td>
<td>$10^5$</td>
<td>$10^8$</td>
<td>$\sim 10^{3010}$</td>
</tr>
</tbody>
</table>
Ah … the math prerequisite comes in handy now.

We simplify all of the calculations about efficiency by concerning ourselves with only order-of-magnitude changes.
This is an important definition.

Now, the math purists in the crowd (yes, I know you’re out there) are probably wincing right now.

Not to worry, we’ll get to see a messier definition later that has more mathematical rigor.
Exercise 1

- Prove that $5n+3$ is $O(n)$

How do you prove this?

Well, $f(n)$ is $5n+3$, $g(n)$ is $n$.

Find a value for $c$, such that $5n+3 \leq cn$, for all $n$ bigger than some $n_0$, which you also get to choose.

<spacer>

<spacer>
Yes, it’s pretty easy if you make \( n_0 = 5 \). Then \( c \) can be 6.
Exercise 2

- Prove that $5n^2 + 42n + 17$ is $O(n^2)$

Here’s one that’s a little harder.
Some notes about the notation.

**Implications**

- The notation \( f(n) = O(g(n)) \) is *not* an equality
  - (yet another abuse of the = sign; cf., assignment operator)
- Think of it as shorthand for
  - “\( f(n) \) grows at most like \( g(n) \)” or
  - “\( f \) grows no faster than \( g \)” or
  - “\( f \) is bounded by \( g \)”
- \( O(\ ) \) notation is a *worst-case* analysis
  - Generally useful in practice
  - Sometimes want *average-case* or *expected-time* analysis if worst-case behavior is not typical (but these are often much harder to analyze)
Complexity Classes

- Several common complexity classes (problem size $n$)
  - Constant time: $O(k)$ or $O(1)$
  - Logarithmic time: $O(\log n)$
    [Base doesn't matter. Why?]
  - Linear time: $O(n)$
  - “$n \log n$” time: $O(n \log n)$
  - Quadratic time: $O(n^2)$
  - Cubic time: $O(n^3)$
  - ...
  - Exponential time: $O(k^n)$
- $O(n^k)$ is often called *polynomial time*

Understand this table.

Memorize this table.

Grok this table.

[fodder: “grok”?]
You might be wondering how easy, or difficult, it would be to create an algorithm that’s really bad.

Remember your friend and mine, Lenny? Perhaps better known as Leonardo Fibonacci, the most talented Western mathematician of the Middle Ages.

And, of course, you know of the sequence of Fibonacci numbers … which he introduced into Europe in his *magnum opus*, Liber Abici (Book of Calculations).

The Fibonacci sequence, as it is also known, is often given recursively.

\[ F_n = F_{n-1} + F_{n-2}, \text{ where } F_0 = 0 \text{ and } F_1 = 1. \]

It’s a lovely recursive statement, which we can easily translate into simple recursive Java:
public static int fibonacci(int n) {
    if (n < 0) {
        String msg = "Argument cannot be negative.";
        throw new IllegalArgumentException(msg);
    }
    if (n == 0 || n == 1) {
        return n;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}

It’s lovely. Notice we even put in some precondition checking.

What’s the Big-O for this beautiful recursive function?

Now, let’s create an analogous function but using a looping algorithm rather than recursive to calculate the nth Fibonacci number. [left as an exercise for the reader]

What’s the Big-O of the for-loop version?
More simplification.

Notice, this is why we could lump all the constant-time operations together into a single category, even though we know that some of them, like instantiation, take considerably longer than others, like assignment.

All of that detail is lost in Big-O.
Analyzing List Operations (1)

- We can use $O(\ )$ notation to compare the costs of different list implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Dynamic Array</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct empty list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size of the list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>isEmpty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>clear</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now, let's apply our knowledge.

Fill out this chart. Know how and why those are the answers.
Analyzing List Operations (2)

- Operation
  - List
    - Add item to end of list
  - Dynamic Array
  - Linked
  - Locate item (contains, indexOf)
  - Add or remove item once it has been located
Here is a mathematically more rigorous definition of Big-O.

Notice that two forms are given.

Cultural note: Don’t try to out nerd the nerds by being snooty about Big-Theta. It won’t be pretty.
Wait! Isn’t this totally bogus??

- Write better code!!
  - More clever hacking in the inner loops
    - (assembly language, special-purpose hardware in extreme cases)
  - Moore’s law: Speeds double every 18 months
    - Wait and buy a faster computer in a year or two!

- But ...

Yeah … what about all this?
How long is a Computer-Day?

- If a program needs $f(n)$ microseconds to solve some problem, how big a problem can it solve in a day?
- One day = $1,000,000 \times 24 \times 60 \times 60 = 9 \times 10^{10}$ (approx)
- $f(n)$ is $n$ such that $f(n) = \text{one day}$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$9 \times 10^{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5n$</td>
<td>$2 \times 10^{10}$</td>
</tr>
<tr>
<td>$n \log_2 n$</td>
<td>$3 \times 10^{9}$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$3 \times 10^5$</td>
</tr>
<tr>
<td>$n^3$</td>
<td>$4 \times 10^3$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>36</td>
</tr>
</tbody>
</table>

Notice what happens to through-put as the size of the problem grows.

Now, look at the next chart …
### Speed Up The Computer by 1,000,000

Suppose technology advances so that a future computer is 1,000,000 fast than today’s

<table>
<thead>
<tr>
<th>f(n)</th>
<th>original n</th>
<th>speedup on future machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>$9 \times 10^{10}$</td>
<td>million times larger</td>
</tr>
<tr>
<td>5n</td>
<td>$2 \times 10^{10}$</td>
<td>million times larger</td>
</tr>
<tr>
<td>n log₂n</td>
<td>$3 \times 10^9$</td>
<td>60,000 times larger</td>
</tr>
<tr>
<td>n²</td>
<td>$3 \times 10^5$</td>
<td>1,000 times larger</td>
</tr>
<tr>
<td>n³</td>
<td>$4 \times 10^3$</td>
<td>100 times larger</td>
</tr>
<tr>
<td>2ⁿ</td>
<td>36</td>
<td>+20</td>
</tr>
</tbody>
</table>

Woohoo … see how much bang we get for a million-fold increase in throughput.

Recursive fibonacci is still pretty bad.
Practical Advice For Speed Lovers

- First pick the right algorithm and data structure
- Implement it carefully, insuring correctness
- Then optimize for speed – but only where it matters
  - Constants do matter in the real world
  - Clever coding can speed things up, but the result is likely to be harder to read, modify
- Use tools to find hotspots – concentrate on these

“Premature optimization is the root of all evil”

– Donald Knuth
"It is easier to make a correct program efficient than to make an efficient program correct"

— Edsger Dijkstra
Summary

- Analyze algorithm sufficiently to determine complexity
- Compare algorithms by comparing asymptotic complexity
- For large problems, an asymptotically faster algorithm will always trump clever coding tricks
- Optimize/tune only things that actually matter, once you’ve picked the best algorithm
Computer Science Note

- Algorithmic complexity theory is one of the key intellectual contributions of Computer Science
- Typical problems
  - What is the worst/average/best-case performance of an algorithm?
  - What is the best complexity bound for all algorithms that solve a particular problem?
- Interesting and (in many cases) complex, sophisticated math
  - Probabilistic and statistical as well as discrete
- Still some key open problems
  - Most notorious: P \neq NP