• What are algorithms?
• Why analyze them?
• Then what?

Updated: 2018-05-14
Algorithm

So what’s this “algorithm” thing I keep hearing about?

- A process or set of steps to follow to solve a problem.
- These days it usually refers to something a computer follows specifically.
- They’re named after Muhammad ibn Musa al-Khwarizmi
  - Muhammad, son of Musa, the dude from Khwarizm
  - Transliterated to Latin as “Algoritmi”
- Algebra is also named after one of his books
Algorithm Example

An algorithm is the sequence of steps you take to go from a problem to a solution.

PROBLEM: I’ve a bunch of numbers and wanna know what’s the biggest one!

SOLUTION: Look at each number and remember the biggest you’ve seen so far. Once you’ve seen all of them, the one you’re remembering is the biggest one of all!
Algorithms and Programming

Problem: Input / Arguments to function / program
Solution: Output / return value

When you write a program, you are implementing an algorithm.

The algorithm is an abstract representation how to solve the problem, the function is the concrete implementation of those steps.
Analyzing an Algorithm

There are many ways we can assess an algorithm. (See the list in Module 1)

We’re sticking with efficiency: How well the algorithm makes use of resources.

We’re also sticking with the Big-O notation used in CS116*/136.

*Unless you took it a while ago…
Efficiency

Typically efficiency is measured in terms of Time or Space

Time
• How many operations are needed

Space
• How much additional memory is required
An Algorithm’s Efficiency

While you can analyze an algorithm’s efficiency, it’s usually better to analyze an implementation.

Why?

Your algorithm will be abstract, and use abstract views of the data types involved.

Depending on the implementation, the operations will be different!
Does every operation count?

If we have a list \( L \) where \( \text{len}(L) == 10 \).
Algorithm A takes 37 operations
Algorithm B takes 50 operations. A is faster, but since computers perform \text{BILLIONS} of operations per second, who gives a…darn?

Let’s make \( \text{len}(L) == 1000000 \)

Algorithm A takes 200,000,000,017 operations
Algorithm B takes 5,000,000 operations.

B is now the clear winner.
Big-O

We’re interested in how algorithms (and their implementations) perform for “sufficiently large input size”

We’re also more concerned with proportionality than we are with exact counts.
Big-O With Minimal Math

You want to work out exactly how many operations are in your algorithm:

\[ x = x + 3 \]

There are two operations right there. Do that for the whole function and you get an equation like

\[ T(n) = 2n^2 - 3n +1 \]

The TL;DR of Big-O is “take the highest order term, ignore the coefficients”
Ignore the what now?

\[ T(n) = 2n^2 - 3n + 1 \]

"Number of primitive operations as a function of the input size n"

"Coefficient"  "Highest Order Term"

\[ T(n) \text{ is in } O(n^2) \]
Big-O, the Set

Math Light Version

$O(n^2)$ is the set of all mathematical functions where the highest order term is some constant $c$ times $n^2$

Math Heavy Version

$O(n^2) = \{ f \mid \exists c, n_0 : n \geq n_0 \Rightarrow f(n) \leq cn^2 \}$
Big-O, in plot form

\[ T(n) = 2n^2 + 32n + 1 \]

\[ 3n^2 \]

\[ n_0 = 32.03 \]
Forget the Math

Since all we care about is “proportionality” and “sufficiently large $n$” we can ignore plenty of annoying numbers.

This is why all of the Big-O arithmetic you learned in CS116 / CS136 works!

Note to those reading along at home: See the CS136 efficiency slides for details! (I.e. This is the slide where Dan takes the adage “Don’t reinvent the wheel” to heart)
Runtime vs Complexity

If you have determined that the number of operations performed by Algorithm A is in $O(n)$, we say Algorithm A has “linear time complexity”.

People tend to also say that’s its runtime. Strictly speaking, runtime is how long it takes the program to run. It’s a number.

Time complexity is how the runtime grows as the input size is increased. E.g. “linear” means “double the size, double the time” and “$2^n$” means “add 1 to the size, double the time”.
Is Big-O all we need?

Who’s “we”?

It’s all you need as CS234 students.

It’s not the ultimate in algorithm analysis. It’s the **starting point**.

Algorithm A has $T(n) = 1000n^2$
Algorithm B has $T(n) = 1/2n^2$

Both belong to $O(n^2)$ but you B will **always** be much faster!
More shortcomings

Algorithm A has \( T(n) = 2n \log_2 n \)
Algorithm B has \( T(n) = 64n \)

A belongs to \( O(n \log n) \), B belongs to \( O(n) \)

That means for sufficiently large \( n \), B is faster. However, the crossover point is \( 2^{32} \), which is around 4 billion. Is that a reasonable size for your problem?

(It might be, but you should ask the question!)
The Big Question

Algorithm A has a runtime in $O(n^3)$, and so does algorithm B.

What do you use?

A: Not enough information. They both sound like OK choices if there isn’t anything lower order.
The Follow-up Question

Algorithm A has a runtime in $O(n^3)$, algorithm B has a runtime in $O(n^2)$.

Which one is faster if the input size is 10,000?

A: Not enough information. “Sufficiently large n” is potentially a big number.
So Big-O is worthless? Damnit grandpa why are you telling me this stuff???

Nah, it’s a good rule of thumb. E.g. for \( n^3 \) vs \( n \log n \), it’s very likely that the cubic algorithm is much slower than the log linear one for most / all inputs.

For two \( O(n \log n) \) algorithms you have to go deeper
A new Twist

You have a dictionary that maps Int -> Str

Data Structure A: A sorted list of pairs
Data Structure B: An unsorted list of pairs

<table>
<thead>
<tr>
<th>Operation</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>O(n)</td>
<td>O(1) amortized</td>
</tr>
<tr>
<td>Lookup</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Remove</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

What do?
Deciding on a Data Structure

Not enough information! I know I keep saying this. We always need more, though.

When implementing an ADT, you need to assume a “typical usage pattern”.

You might want to just implement it *twice* or *thrice*. Probably not 4 times though, that would be ridiculous!
Time Complexity and Documentation

Time Complexity should be documented for all ADT operations. Or, at least, all operations where it is not obviously* O(1)

A2-4 will be clear about what they’re looking for.

*if you aren’t sure if it’s obvious, it’s obviously not obvious
Analyzing Python’s List

As we saw, Python’s list is a **dynamic array** that uses a **doubling** strategy for resizes.

The cost of a resize is $O(n)$, but in CS116 we kept saying “Append is $O(1)$”
Amortized Analysis

This is mostly true, if you do something called “amortization” (it means “averaging”).

**THIS IS NOT AN AVERAGE CASE ANALYSIS**

In an amortized analysis, you’re still thinking worst case. You ask “If my function does n operations, what is complexity of the function?”

If the total cost is in $O(n)$, then the **amortized** cost of those operations is $O(n)/n = O(1)$.
Amortized Resizing

Let’s say you have an empty list, and you .append until its length is $n$.

Resizing is $O(n)$ where $n$ is the new length.

So, the final resize is $O(n)$, the penultimate resize is $O(n/2)$ the antepenultimate resize is $O(n/4)$ etc.

$$n(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{n}) = 2n - 1 \in O(n)$$
Insert / Remove

If you shrink the array when the list is too empty, you can keep both append(v) and pop() ∈ O(1) amortized.

The analysis is a bit tricky if you allow the n operations to be append or pop in any order (i.e. worst case)
Insert Algorithm

```
mylist.insert(2, 0)
```
Bullet Points for Previous Slide

Even if we don’t need to resize, insert is still $O(n)$
- We still need to move everything out of the way
  - If we’re inserting to a position that is $O(n)$ away from the end

Is the position part of “worst case” or not?
- Strong Maybe
**List.Remove**

Let’s delete that 0 from the list!

```python
mylist.pop(2) or mylist.remove(0)
```

O(n) again (even if we don’t need to resize)
Use Case

You are a giant grocery store chain. You have 10,000,000 customers in the loyalty program, and post around 3,000,000 transactions per day. You also get around 1000 new signups per day.

Do you go with A (the sorted list) or B (the unsorted one)?

<note to class: insert meaningful discussion here>
Let’s Try!

For my next trick, I shall analyze the Set!

We saw two ways to do it already, I might as well tell you two more!

• An AVL Tree (we’ll get to those)
• A Hash Table (we’ll get to those)
# Set Implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>List</th>
<th>List (sorted)</th>
<th>AVL Tree*</th>
<th>Hash Table*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>O(1)**</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Remove</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Contains</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Union***</td>
<td>O(nm)</td>
<td>O(n+m)</td>
<td>O(m log ((\frac{n}{m}+1))))</td>
<td>O(nm)</td>
</tr>
</tbody>
</table>

* - Don’t worry about it, we’ll see this data structure soon
** - Amortized
*** - Where the two sets have size n and m, m ≤ n

Wah? Why the heck does Python use Hash Tables?
Hash Table Trailer

Hash Tables suck in theory. But they’re great in practice!

Their **average case** performance is $O(1)$
Again, this is not the same as **amortized**

The quicksort algorithm is in a similar boat. We’ll see this soon, too!

Its worst case complexity is $O(n^2)$ but its average case is $O(n \log n)$