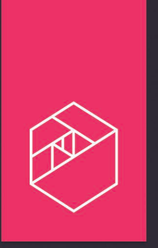
Computational Complexity

What’s important about Computational Complexity?

Why complexity?



- We have to decide between algorithm A and algorithm B
  - We have a dataset for benchmarking algorithms with about 1,000 rows
- What should we do?

	Algo A	Algo B
Time on lab data	5s	30s

Why complexity?

- We have to decide between algorithm A and algorithm B
  - We have a dataset for benchmarking algorithms with about 1,000 rows
- What should we do?
  - This depends on what kinds of data the algorithms will be given *in production*
  - What if we'll be running it on 100k rows?

	Algo A	Algo B
Time on lab data	5s	30s

Why complexity?

- We have to decide between algorithm A and algorithm B
  - We have a dataset for benchmarking algorithms with about 1,000 rows
- What should we do?
  - This depends on what kinds of data the algorithms will be given *in production*
  - What if we'll be running it on 100k rows?
- What if we had a formula that told us the runtime for different size inputs?

	Algo A	Algo B
Time on lab data	5s	30s

Why complexity?

- We have to decide between algorithm A and algorithm B
- What should we do?
- What if we had a formula that told us the runtime for different size inputs?
  - $t_a(n) = 5 * \left(\frac{n}{1000}\right)^2$
  - $t_b(n) = 29 + \left(\frac{n}{1000}\right)$

	Algo A	Algo B
1k	5s	30s

Why complexity?

- We have to decide between algorithm A and algorithm B
- What should we do?
- What if we had a formula that told us the runtime for different size inputs?
  - $t_a(n) = 5 * \left(\frac{n}{1000}\right)^2$
  - $t_b(n) = 29 + \left(\frac{n}{1000}\right)$
  - Now we can know the runtime for any size input

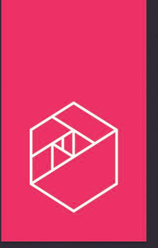
	Algo A	Algo B
1k	5s	30s
10k	500s	39s

Why complexity?

- We have to decide between algorithm A and algorithm B
- What should we do?
- What if we had a formula that told us the runtime for different size inputs?
  - $t_a(n) = 5 * \left(\frac{n}{1000}\right)^2$
  - $t_b(n) = 29 + \left(\frac{n}{1000}\right)$
  - Now we can know the runtime for any size input

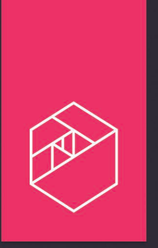
	Algo A	Algo B
1k	5s	30s
10k	500s	39s
100k	13.9 hours	49s

Complexity Defined



- Complexity: An algorithm's complexity is a measure of the resources it uses
- Most common is time complexity
  - In a few weeks, we will discuss space complexity (memory use)

Complexity Defined



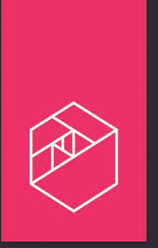
- Complexity: An algorithm's complexity is a measure of the resources it uses
- Most common is time complexity
  - In a few weeks, we will discuss space complexity (memory use)
- Time complexity can be measured in several ways
  - Wall clock measures amount of time that passes while the algorithm runs
    - Imminently useful
    - However, sensitive to details about hardware and implementation like processor core speed and parallelization

Complexity Defined

- Complexity: An algorithm's complexity is a measure of the resources it uses
- Most common is time complexity
  - In a few weeks, we will discuss space complexity (memory use)
- Time complexity can be measured in several ways
  - Wall clock measures amount of time that passes while the algorithm runs.
    - Imminently useful
    - However, sensitive to details about hardware and implementation like processor core speed and parallelization
  - Compute time measures the computations made during the algorithm's execution
    - ex: number of comparisons, processor cycles
    - Relevant across different hardware setups and implementations

Introducing Big-O Notation

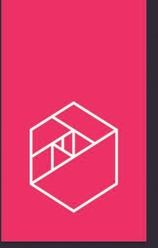
Big-O



Problem:

- We care about complexity but equations are difficult to compare
- Ex:
  - $f_a(n) = n^2 + 200n(n + \log(n))$ vs
  - $f_b(n) = .01n^3$
- Do we care about which is faster at $n = 10$ or $n = 1{,}000$?

Big-O



Problem:

- We care about complexity but equations are difficult to compare
- Ex:
  - $f_a(n) = n^2 + 200n(n + \log(n))$ vs
  - $f_b(n) = .01n^3$
- Do we care about which is faster at $n = 10$ or $n = 1{,}000$?

Solution

- Big-O notation simplifies comparisons of computational resources

Big-O

Idea

- At scale, the most important comparison is the overall growth behavior
- For an equation like $2^n + 4n^2$, which part is going to matter as $n \to \infty$?

Big-O

Idea

- At scale, the most important comparison is the overall growth behavior
- For an equation like $2^n + 4n^2$, which part is going to matter as $n \to \infty$?

Complexity equations	
$2n!$	
$2^n + 4n^2$	
$n^{100} + 3$	
$n^2 + n$	

Big-O

Idea

Definition

- **If there exists a k and $g(n)$ such that $\lim_{n\to\infty} f(n) < kg(n)$**
- **Then $f(n) = O(g(n))$**

Complexity equations	What we care about
$2n!$	$2n!$
$2^n + 4n^2$	$2^n + 4n^2$
$n^{100} + 3$	$n^{100} + 3$
$n^2 + n$	$n^2 + n$

Big-O

Idea

Definition

- **If there exists a k and $g(n)$ such that $\lim_{n\to\infty} f(n) < kg(n)$**
- **Then $f(n) = O(g(n))$**

Complexity equations	What we care about	Big O
$2n!$	$2n!$	$O(n!)$
$2^n + 4n^2$	$2^n + 4n^2$	$O(2^n)$
$n^{100} + 3$	$n^{100} + 3$	$O(n^{100})$
$n^2 + n$	$n^2 + n$	$O(n^2)$

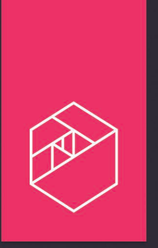
Finding Big-O

Using heuristics to find Big-O

Heuristic workflow

1. Find expression
   - (more on how to find the expression in the notebook section)
2. Simplify expression
3. Only keep highest order term
4. Discard constants

Heuristics Cheat Sheet

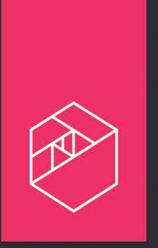


- Coefficients: Drop any constants / coefficients
  - O(3N) is reduced to O(N)
- Simple statements: O(1)
  - print()
  - dict[‘key’] = value)
- Loops: O(inside loop) * number of iterations
- Consecutive statements: Most complex statement on same indent
- Conditional statements: O(condition + longest branch)
- Leading Terms: Only keep the leading term
  - O(N2 + N) is reduced to O(N2)
- Keep the furthest left term: 2n >> n5 >> n3 >> n1 >> n lg(n) >> lg(n) >> 1

Finding Big-O (you try it)

Find the big-O complexity for:
$f(n) = n(2n + \sin(n))$

Finding Big-O



Find the big-O complexity for:

$f(n) = n(2n + \sin(n))$

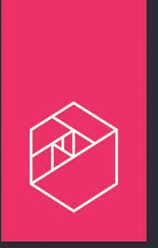
$\rightarrow 2n^2 + n\sin(n)$ **[simplify]**

$\rightarrow 2n^2$ **[keep highest order term]**

$\rightarrow n^2$ **[discard coefficients]**

$f(n) = O(n^2)$

Takeaways



- Computational Complexity allows us to evaluate an algorithm's resource use
- Big-O notation is a tool for summarizing complexity in a useful way
- The Big-O complexity for an algorithm can be solved formally using proofs or informally using heuristics


John Westrock

Extras

Method 2: Proof

Proof

- Show that there exists a k_0 and an n_0 such that
  - For all $n > n_0$, $k_0 g(n) > f(n)$
  - Then $f(n) = O(g(n))$

Method 2 (you try it)

Use a proof to find the big-O complexity of:

$$f(n) = \frac{1}{2}(2^n) - 3n$$

Let $g(n) = 2^n$

$$k2^n > \frac{1}{2}(2^n) - 3n$$

$$k > \frac{1}{2} - \frac{3n}{2^n}$$

True for $k = .5, n > 3$**, therefore** $f(n) = O(2^n)$ ■