Algorithm Analysis I (cont); Algorithm Analysis II: Amortization

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

gradescope.com/courses/256241

- Consider **f(*n*)** = n^3 and **g(*n*)** = $4n^2 + 3n + 4$. Is **g(*n*)** in O(**f(*n*)**)?

- Bonus question: choose a `c` and n_0 to support your answer. You do not need to submit a proof, just two values.

Announcements

- Substitute lecturer for Wednesday; TBD for Friday
  - Canceling my Tuesday afternoon OH
- Delayed upload of Friday (L3) materials; Gradescope duedate for "participation" given an extra 24h
  - Ie, due Tuesday at 12:20
- Project 1's Checkpoint is a Gradescope survey that released on Thursday
  - Stays open for 2d

Lecture Outline

- Algorithm Analysis I: Asymptotics Wrapup
  - **Review: Big-O, Formally**
  - Big-Omega and Big-Theta

- Algorithm Analysis II: Amortization
  - Amortized Bounds
  - Where We've Come / Where We're Going

Computational Model for a Single Algorithm

- Running benchmarks is noisy and not predictive

- In our model, we abstract away the computer by counting:
  1. Constant-space elements (space complexity)
  2. Constant-time operations (time complexity)

- We can analyze multiple cases, but typically focus on worst
  - So we typically analyze the slower branch

Asymptotic Analysis to Compare Algorithms

- Even with a simplified model to derive expressions, we still don't know how to compare functions
  - What's faster: `8n + 2` or $0.2n^2$?
  - Depends on ***specific case***, ***constant factors***, and ***size of n*** !

- We pick n→∞ to establish a shared point of reference
  - As n→∞ , constant factors don't contribute meaningfully to runtime
  - Intuitively, we begin to compare *curve shapes*

- Asymptotic analysis compares functions
  - Eg Big-O, but also big-Ω et al.

Big-Oh Relates Functions

- We use O on a function f(n) (for example n^2) to mean *the set of functions with asymptotic behavior* ***less than or equal to*** f(n)

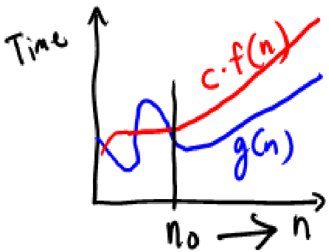
- So ($3n^2+17$) **is in** $O(n^2)$
  - $3n^2+17$ and n^2 have the same **asymptotic behavior**

- Formally,

Definition: **g(*n*)** is in O(**f(*n*)**) iff there exist positive constants ***c*** and $\boldsymbol{n_0}$ such that

g(*n*) ≤ ***c*** **f(*n*)** for all $\boldsymbol{n} \geq \boldsymbol{n_0}$

$n_0 \geq 1$ and a natural number; $c > 0$



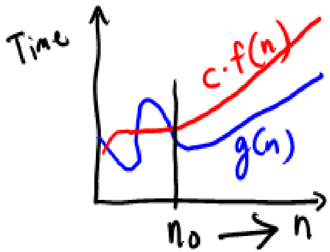
gradescope

gradescope.com/courses/256241

- True or false?
  - $4+3n$ is $O(n)$
  - $n+2\log n$ is $O(\log n)$
  - $\log n+2$ is $O(1)$
  - n^{50} is $O(1.1^n)$

- Notes:
  - Do NOT ignore constants that are not multipliers:
    - n^3 is $O(n^2)$: FALSE
    - 3^n is $O(2^n)$: FALSE
  - When in doubt, refer to the definition

Big-Oh, Formally



Definition: g(n) is in O(f(n)) iff there exist positive constants c and n_0 such that

g(n) ≤ c f(n) for all $n \geq n_0$

$n_0 \geq 1$ and a natural number; $c > 0$

- To show g(n) is in O(f(n)), pick
  - a c large enough to "cover the constant factors"
  - an n_0 large enough to "cover the lower-order terms"

- Example: Let g(n) = 3n + 4 and f(n) = n
  - https://www.desmos.com/calculator/zmsgznyrnu
- Example: Let g(n) = 3n + 4 and f(n) = n^5
  - https://www.desmos.com/calculator/b5tg7wy6dk
- Example: Let g(n) = 3n + 4 and f(n) = 2^n
  - https://www.desmos.com/calculator/n0nzmjxanh

What's with the c?

- To capture this notion of "similar asymptotic behavior", we allow a constant multiplier called **c**. Consider:

  g(n) = 3n+4

  f(n) = n

- These have the same asymptotic behavior (linear), even though **g(n)** is always larger
  - ie, there is **no** positive n_0 such that $g(n) \leq f(n)$ for all $n \geq n_0$

- The '**c**' allows us to show their asymptotic relationship:

  $$g(n) \leq c\, f(n) \qquad \text{for all } n \geq n_0$$

- To show **g(n)** is in O(**f(n)**), let $c = 12$, $n_0 = 1$
  - https://www.desmos.com/calculator/zmsgznyrnu

Example: Using the Definition of Big-Oh

To show **g(*n*)** is in O(**f(*n*)**), pick a ***c*** large enough to "cover the constant factors" and n_0 large enough to "cover the lower-order terms"

- Example: Let **g(*n*)** = $4n^2 + 3n + 4$ and **f(*n*)** = n^3

Choose c=4:

$$g(n) = 4n^2 + 3n + 4 \overset{?}{\leq} 4n^3 = cf(n)$$

Choose $n_0 = 3$

$$4 \cdot 9 + 3 \cdot 3 + 4 \overset{?}{\leq} 4 \cdot 27$$

$$49 \leq 108$$

$$\therefore g(n) \leq 4f(n) \ \forall \ n \geq 3$$

gradescope

gradescope.com/courses/256241

- For g(n) = 4n and f(n) = n^2, show g(n) is in O(f(n))
  - A valid proof is to find valid **c** & **n_0**
  - When n=4, g(n) =16 & f(n) =16; this is the crossing over point
  - So we can choose **n_0** = 4, and **c** = 1

  - Note: There are many possible choices:
    ex: **n_0** = 78, and **c** = 42 works fine

g(n) is in O(f(n)) iff there exist *positive* constants c and n_0 such that

$$g(n) \leq c\, f(n) \text{ for all } n \geq n_0.$$

Example 3: Using the Definition of Big-Oh

- For **g(n)** = n^4 and **f(n)** = 2^n, show **g(n)** is in O(**f(n)**)
  - A valid proof is to find valid **c** & $\mathbf{n_0}$
  - One possible answer: $\mathbf{n_0}$ = 20, and **c** = 1

g(n) is in O(f(n)) iff there exist *positive* constants c and n_0 such that
$$g(n) \le c\, f(n) \text{ for all } n \ge n_0.$$

Reviewing the Big-O Rules

- Eliminate coefficients because we don't have units anyway
  - $3n^2$ versus $5n^2$ doesn't mean anything because our computational model assumes "constant" operations

- Eliminate low-order terms because they have vanishingly small impact as *n* grows

- Do NOT ignore constants that are not multipliers
  - n^3 is not $O(n^2)$
  - 3^n is not $O(2^n)$

(These all follow from the formal definition)

Common Complexity Classes

O(1) **(O(k) for any k)*	Constant
O(log log n)	
O(log n)	Logarithmic
$O(\log^k n)$ **(for any k>1)*	
O(n)	Linear
O(n log n)	Loglinear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^k)$ **(for any k>1)*	Polynomial
$O(k^n)$ **(for any k>1)*	Exponential

Note: "exponential" does not mean "grows really fast"; it means "grows at rate proportional to k^n for some $k>1$"

Lecture Outline

- Algorithm Analysis: Asymptotics Wrapup
  - Review: Big-O, Formally
  - **Big-Omega and Big-Theta**
- Algorithm Analysis II: Amortization
  - Amortized Bounds
  - Where We’ve Come / Where We’re Going

Big-O: Intuition

- Big-O can be thought of as something like “less-than or equals”

Function	Big-O
$N^3 + 3N^4$	$O(N^4)$
$(1 / N) + N^3$	$O(N^3)$
$Ne^N + N$	$O(Ne^N)$
$40 \sin(N) + 4N^2$	$O(N^2)$

Also Big-O
$O(N^5)$
$O(N^{423421531542})$
$O(N*3^N)$
$O(N^{2.1})$

$g(n)$ is in $O(\ f(n)\ )$ iff there exist positive constants c and n_0 such that

$$g(n) \leq c\ f(n) \quad \text{for all } n \geq n_0$$

Big-Omega: Intuition

- Big-Omega can be thought of as something like "greater-than or equals"

Function	Big-O	Big-Omega	Also Big-Omega
$N^3 + 3N^4$	$O(N^4)$	$\Omega(N^4)$	$\Omega(N^2)$
$(1 / N) + N^3$	$O(N^3)$	$\Omega(N^3)$	$\Omega(1)$
$Ne^N + N$	$O(Ne^N)$	$\Omega(Ne^N)$	$\Omega(N)$
$40 \sin(N) + 4N^2$	$O(N^2)$	$\Omega(N^2)$	$\Omega(N)$

$g(n)$ is in $\Omega(\, f(n) \,)$ iff there exist positive constants c and n_0 such that

$$g(n) \geq c\, f(n) \quad \text{for all } n \geq n_0$$

Big-Theta: Intuition

- Big-Theta more closely resembles “equals”

Function	Big-O	Big-Omega	Big-Theta
$N^3 + 3N^4$	$O(N^4)$	$\Omega(N^4)$	$\Theta(N^4)$
$(1 / N) + N^3$	$O(N^3)$	$\Omega(N^3)$	$\Theta(N^3)$
$Ne^N + N$	$O(Ne^N)$	$\Omega(Ne^N)$	$\Theta(Ne^N)$
$40 \sin(N) + 4N^2$	$O(N^2)$	$\Omega(N^2)$	$\Theta(N^2)$

$g(n)$ is in $\Theta(f(n))$ iff there exist positive constants c and n_0 such that

$c_1 f(n) \leq g(n) \leq c_2 f(n)$ for all $n \geq n_0$

Big-O, Big-Theta, Big-Omega Relationship

- If a function f is in Big-Theta, what does it mean for its membership in Big-O and Big-Omega? Vice versa?

Function	Big-O	Big-Theta	Big-Omega
$N^3 + 3N^4$	$O(N^4)$	$\Theta(N^4)$	$\Omega(N^4)$
$(1 / N) + N^3$		$\Theta(N^3)$	
$Ne^N + N$		$\Theta(Ne^N)$	
$40 \sin(N) + 4N^2$		$\Theta(N^2)$	

In Other Words …

- **Upper bound**: O(**f(n)**) is the set of all functions asymptotically *less than or equal* to **f(n)**
  - g(n) is in O(**f(n)**) if there exist constants c and n_0 such that

    $$g(n) \leq c\, f(n) \text{ for all } n \geq n_0$$

- **Lower bound**: Ω(**f(n)**) is the set of all functions asymptotically *greater than or equal* to **f(n)**
  - g(n) is in Ω(**f(n)**) if there exist constants c and n_0 such that

    $$g(n) \geq c\, f(n) \text{ for all } n \geq n_0$$

- **Tight bound**: θ(**f(n)**) is the set of all functions asymptotically *equal* to **f(n)**
  - Intersection of O(**f(n)**) and Ω(**f(n)**) (can use *different* c values)

A Warning about Terminology

- A common error is to say $O(f(n))$ when you mean $\theta(f(n))$
  - People often say O() to mean a tight bound
    - Say we have f(n)=n; we could say f(n) is in O(n), which is true, but only conveys the upper-bound
    - Since f(n)=n is *also* $O(n^5)$, it's tempting to say "this algorithm is *exactly* $O(n)$"
    - It's better to say it is $\theta(n)$
      - That means that it is not, for example $O(\log n)$

- Less common notation:
  - "little-oh": like "big-Oh" but strictly less than
    - Example: f(n) is $o(n^2)$ but not $o(n)$
  - "little-omega": like "big-Omega" but strictly greater than
    - Example: f(n) is $\omega(\log n)$ but not $\omega(n)$

What We are Analyzing

- The most common thing to do is give an *O* or θ **bound** to the **worst-case** running **time** of an **algorithm**

- Reminder that Case Analysis != Asymptotic Analysis
  - Cases describe *a specific path through your algorithm*
  - Big-O/Big-Omega/Big-Theta bounds describe *curve shapes for large values*

- When comparing two algorithms, you must pick all of these:
  - A case (eg, best, worst, amortized, etc)
  - A metric (eg, time, space)
  - A bound type (eg, big-O, big-Theta, little-omega, etc)

What We are Analyzing: Examples

- True statements about binary-search algorithm:
  - Common: θ(`log` *n*) running-time in the worst-case
  - Less common: $\theta(1)$ in the best-case
    - item is in the middle
  - Less common: Ω(`log log` *n*) in the worst-case
    - it is not really, really, really fast asymptotically
  - Less common (but very good to know): the find-in-sorted-array ***problem*** is Ω(`log` *n*) in the worst-case
    - *No* algorithm can do better (without parallelism)
    - A ***problem*** cannot be $O(f(n))$ since you can always find a slower algorithm, but can mean ***there exists*** an algorithm

Lecture Outline

- Algorithm Analysis I: Asymptotics Wrapup
  - Review: Big-O, Formally
  - Big-Omega and Big-Theta
- Algorithm Analysis II: Amortization
  - **Amortized Bounds**
  - Where We've Come / Where We're Going

Linear Search: Best vs Worst Case

- Find an integer in a *sorted* array

2	3	5	16	37	50	73	75	126

```
// Requires arr to be sorted
// Returns whether k is in array
boolean findSorted(int[] arr, int k) {
  for(int i=0; i < arr.length; ++i) {
    if(arr[i] == k)
      return true;
    else if(arr[i] > k)
      return false;
  }
  return false;
}
```

Best k:

Worst k:

Complexity Cases

- We started with two cases:
  - **Worst-case complexity**: *maximum* number of steps algorithm takes on "most challenging" input of size N
  - **Best-case complexity**: *minimum* number of steps algorithm takes on "easiest" input of size N

- We punted on one case: **Average-case complexity**
  - Sometimes: relies on distribution of inputs
    - Eg, binary heap's O(1) insert
    - See CSE312 and STAT391
  - Sometimes: uses randomization in the algorithm
    - Will see an example with sorting; also see CSE312

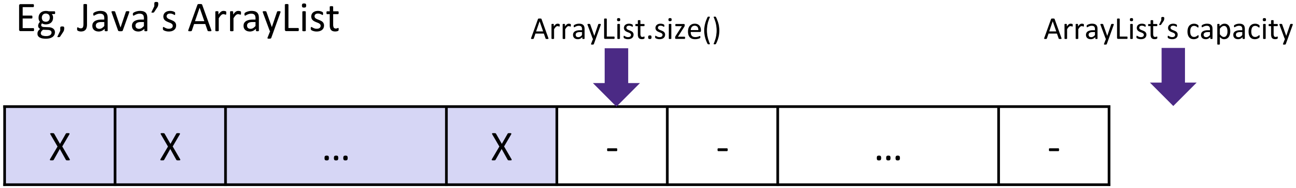
- We've mentioned, but not defined, one *category* of cases:
  - **Amortized-case complexity**

Amortized Analyses = Multiple Executions

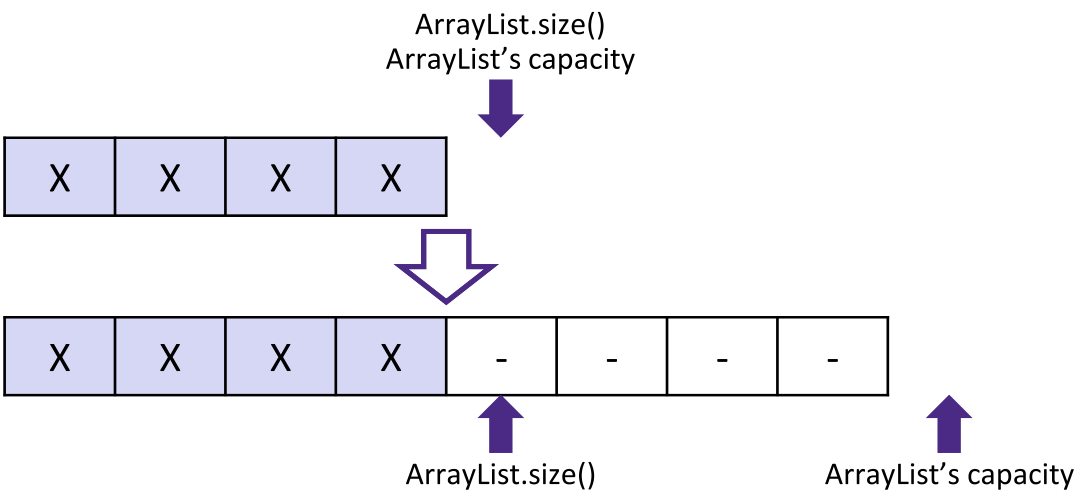
Single Execution	**Multiple Executions**
Worst Case	Amortized Worst Case
Best Case	Amortized Best Case
Average Case	*Amortized Average Case*

Amortized Analysis: `ArrayList.add()`

- Consider adding an element to an array-backed structure
  - Eg, Java's ArrayList



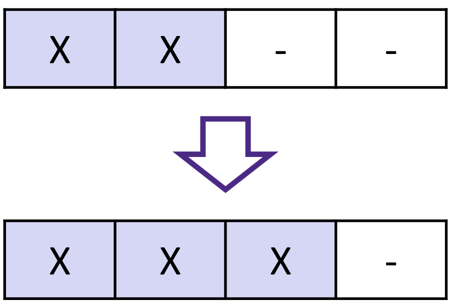
- When the underlying array fills, we allocate and copy contents



`ArrayList.add()` Runtime (1 of 2)

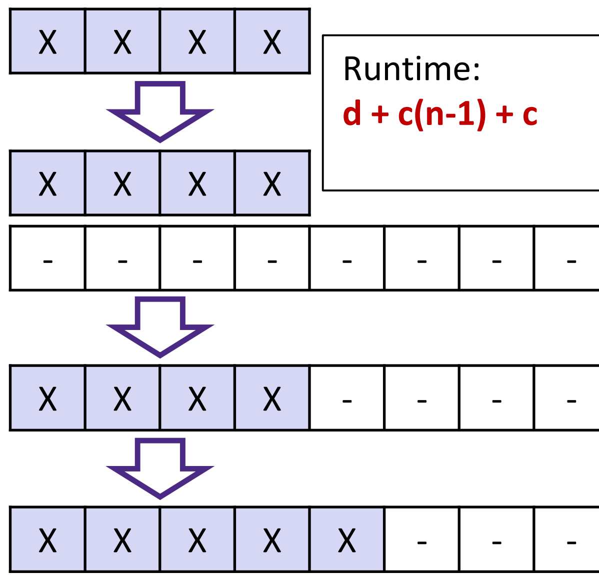
- We know that copying a single element and allocating arrays are both constant-time operations
  - Let's call their runtimes 'c' and 'd', respectively

Most of the time



Runtime:
c

Worst case



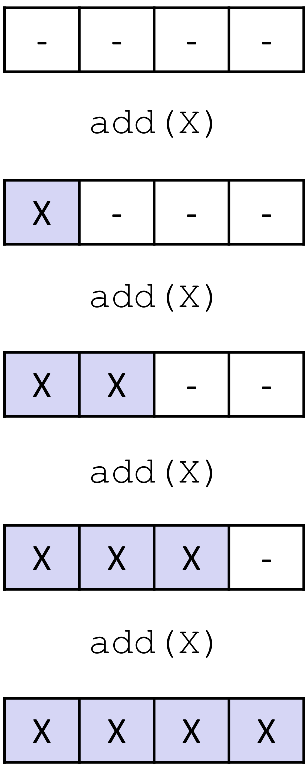
Runtime:
d + c(n-1) + c

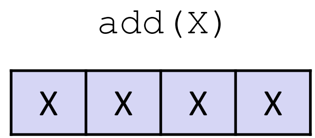
`ArrayList.add()` Runtime (2 of 2)

Single Execution	Multiple Executions	
Worst Case: **Θ(N)**	Aggregate Worst: **Θ(N)**	Amortized Worst: **??**
Best Case: **Θ(1)**	Aggregate Best: **Θ(1)**	Amortized Best: **??**

- Some applications *cannot tolerate* the "occasional O(n) behavior"
- Other applications *can tolerate* "occasional O(n) behavior" if we can show that it's "not too bad" / "not too common"

`ArrayList.add()`: Best-Case Aggregate Runtime

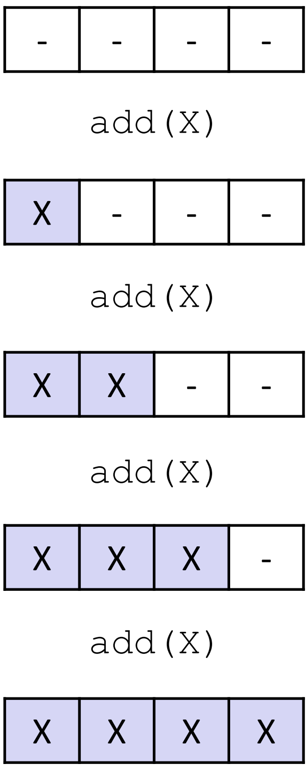


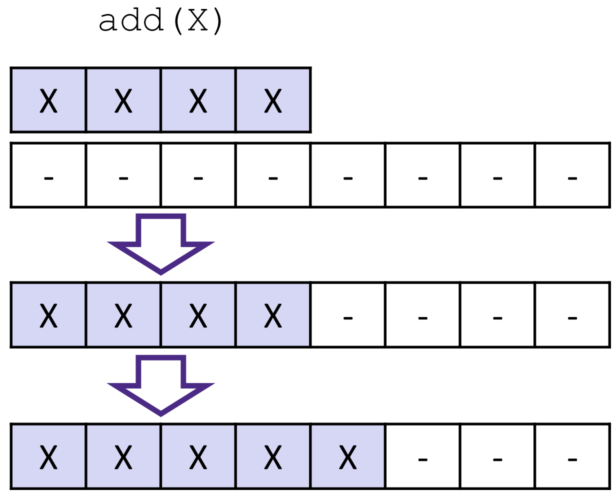


Best-case Aggregate Runtime:

gradescope

gradescope.com/courses/256241





Worst-case Aggregate Runtime:

Amortized Analysis Intuition

- See Weiss, ch 11, for formal methods

- But the intuition is: if our client is willing to tolerate it, we will "smooth" the *aggregate cost of n operations* over n itself

Single Execution	Multiple Executions	
Worst Case: **Θ(N)**	Aggregate Worst: **Θ(N)**	Amortized Worst: **Θ(1)**
Best Case: **Θ(1)**	Aggregate Best: **Θ(1)**	Amortized Best: **Θ(1)**

- Note: we increased our array size by a factor of n (eg, 2n, 3n, etc). What if we increased it by a constant factor (eg, 1, 100, 1000) instead?

Summary

- Asymptotic analysis gives us a common "frame of reference" with which to compare algorithms
  - Most common comparisons are Big-O, Big-Omega, and Big-Theta
  - But also little-o and little-omega
- Case Analysis != Asymptotic Analysis
- We combine asymptotic analysis and case analysis to compare the behavior of data structures and algorithms