Algorithm Analysis I (cont); Priority Queue ADT

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Announcements

- Project 1's Checkpoint is *due tomorrow!
  - *actually due, not airquotes "due"
  - Checkpoint is a Gradescope-administered survey
  - You should have contacted your P1 partner by now!
- Three exercises released today; all *due on Monday
  - Ex 2: code modeling
  - Exs 3 & 4: heap
- Office hours finally (😳😳😳) posted!
- Upvote questions at pollev.com/cse332

Learning Objectives

- Be able to prove a O, θ, or Ω bound for any iterative algorithm
- Choose between the following ADTs when given a specific scenario: List, Stack, Queue, Dictionary, Set, Priority Queue

Lecture Outline

- Algorithm Analysis Intro: Wrapup
  - **Review: Big-O, Formally**
  - Big-Omega and Big-Theta
- Priority Queue ADT

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**

- Confusingly, we also say/write:
  - $(3n^2+17)$ **is** $O(n^2)$
  - $(3n^2+17)$ **∈** $O(n^2)$
  - $(3n^2+17)$ **=** $O(n^2)$ ← *least ideal*

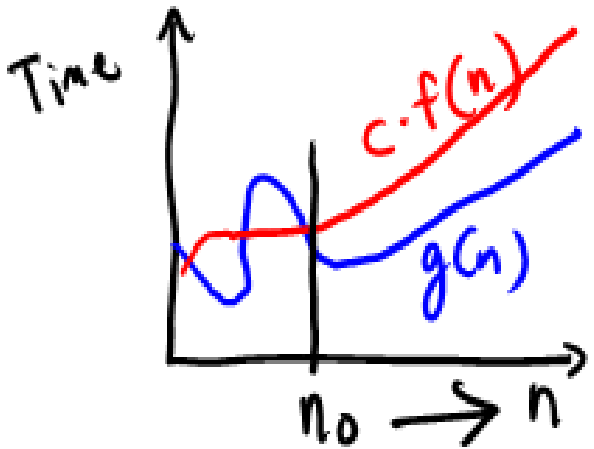
- But we would never say $O(n^2) = (3n^2+17)$

Big-Oh, Formally (1 of 3)

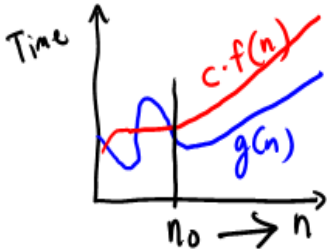
Definition: **g(*n*)** is in O(**f(*n*)**) iff there exist positive constants ***c*** and n_0 such that

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

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$



Big-Oh, Formally (2 of 3)



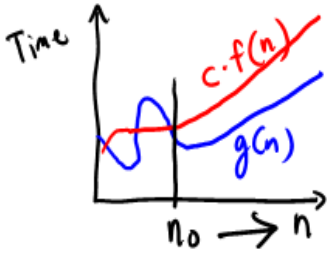
Definition: $g(n)$ is in $O(f(n))$ iff there exist positive constants c and n_0 such that

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

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$

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"

Big-Oh, Formally (3 of 3)



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

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

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$

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), so **g(n)** is in O(**f(n)**) even though **g(n)** is always larger

- There is **no** positive n_0 such that **g(n)** ≤ **f(n)** for all n ≥ n_0
  - The '**c**' in the definition allows for that:

    g(*n*) ≤ ***c*** **f(*n*)** for all $n \geq n_0$
  - To show **g(n)** is in O(**f(n)**), let **c** = 12, n_0 = 1

An Example

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 $n_0 = 3$

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

$$49 \leq 108$$

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

Your Turn!

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

- 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

Reviewing the Big-O Rules

- Eliminate coefficients because we don't have units anyway
  - $3n^2$ versus $5n^2$ doesn't mean anything when we cannot count operations very accurately
- 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 Intro: Wrapup
  - Review: Big-O, Formally
  - **Big-Omega and Big-Theta**
- Priority Queue ADT

Big-O: Intuition

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

Function	Big-O	Also Big-O
$N^3 + 3N^4$	$O(N^4)$	$O(N^5)$
$(1 / N) + N^3$	$O(N^3)$	$O(N^{423421531542})$
$Ne^N + N$	$O(Ne^N)$	$O(N*3^N)$
$40 \sin(N) + 4N^2$	$O(N^2)$	$O(N^{2.1})$

tight bound

loose bounds

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

$g(n) \leq c f(n)$ 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
$N^3 + 3N^4$	$O(N^4)$	$\Omega(N^4)$	$\Omega(N^2)$
$(1 / N) + N^3$	$O(N^3)$	$\Omega(N^3)$	$\Omega(1)$
$Ne^N + N$	$O(Ne^N)$	$\Omega(Ne^N)$	$\Omega(N)$
$40 \sin(N) + 4N^2$	$O(N^2)$	$\Omega(N^2)$	$\Omega(N)$

← tight bounds →

loose bound

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

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

Big-Theta: Intuition

- Big-Theta more closely resembles "equals"

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

← tight bounds →

- same n_0
- two different c_1 and c_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
$N^3 + 3N^4$	$O(N^4)$	$\Theta(N^4)$	$\Omega(N^4)$
$(1 / N) + N^3$		$\Theta(N^3)$	
$Ne^N + N$		$\Theta(Ne^N)$	
$40 \sin(N) + 4N^2$		$\Theta(N^2)$	

$f(n) \in O(g(n))$ and also $f(n) \in \Omega(g(n))$ iff $f(n) \in \Theta(g(n))$

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

    $g(n) \leq c\ f(n)$ 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

    $g(n) \geq c\ f(n)$ 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 $O(\texttt{log}\ n)$

- Less common notation:
  - "little-oh": like "big-Oh" but strictly less than
    - Example: f(n) is $o(n^2)$ but not $o(n)$ $<$ (compare to $\leq$)
  - "little-omega": like "big-Omega" but strictly greater than
    - Example: f(n) is $\omega(\texttt{log}\ n)$ but not $\omega(n)$ $>$ (compare to $\geq$)

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: $\theta(\log n)$ running-time in the worst-case
  - Less common: $\theta(1)$ in the best-case
    - item is in the middle
  - Less common: $\Omega(\log \log n)$ 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 $\Omega(\log n)$ 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 Intro: Wrapup
  - Review: Big-O, Formally
  - Big-Omega and Big-Theta

- **Priority Queue ADT**

ADTs So Far (1 of 3)

Set ADT. A collection of values.

- A set has a size defined as the number of elements in the set
- You can add and remove values, but the contained values are unique
- Each value is accessible via a "get" operation

Dictionary ADT. A collection of keys, each associated with a value.

- A dictionary has a size defined as the number of elements in the dictionary
- You can add and remove (key, value) pairs, but the keys are unique
- Each value is accessible by its key via a "find" or "contains" operation

ADTs So Far (2 of 3)

List ADT. A collection storing an ordered sequence of elements.

- Each element is accessible by a zero-based index
- A list has a size defined as the number of elements in the list
- Elements can be added to the front, back, *or any index in the list*
- Optionally, elements can be removed from the front, back, *or any index in the list*

ADTs So Far (3 of 3)

Stack ADT. A collection storing an ordered sequence of elements.

- A stack has a size defined as the number of elements in the stack
- Elements can only be added and removed from the top ("LIFO")

Queue ADT. A collection storing an ordered sequence of elements.

- A queue has a size defined as the number of elements in the queue
- Elements can only be added to one end and removed from the other ("FIFO")

A Scenario

- What is the difference between waiting for service at a pharmacy versus an ER?
  - Pharmacies usually follow the rule "First Come, First Served"

Queue ADT

  - Emergency Rooms assign priorities based on each individual's need

Priority Queue ADT

A New ADT: Priority Queue

- See Weiss Chapter 6

- A **priority queue** holds *compare-able data*
  - Unlike lists, stacks, and queues, we need to *compare items*
    - Given *x* and *y:* is *x* less than, equal to, or greater than *y?*
    - Much of this course will require comparable items: e.g. sorting
  - Typically two fields: the *priority* and the *data*

- Aside: we will use integers as priority and data
  - For simplicity in lecture, we'll suppose data are **`int`**s *and* that same **`int`** value is also the priority
    - **`int`** priorities are common, but really just need `Comparable`
  - Not having "other data" is very rare
    - Example: print job has a priority *and* the file to print

Priority Queue ADT

Priority Queue ADT. A collection storing a set of elements and their priority.

- A PQ has a size defined as the number of elements in the set
- You can add elements (and their priorities)
- You cannot access or remove arbitrary elements, only the element with the min priority

Primary Operations:

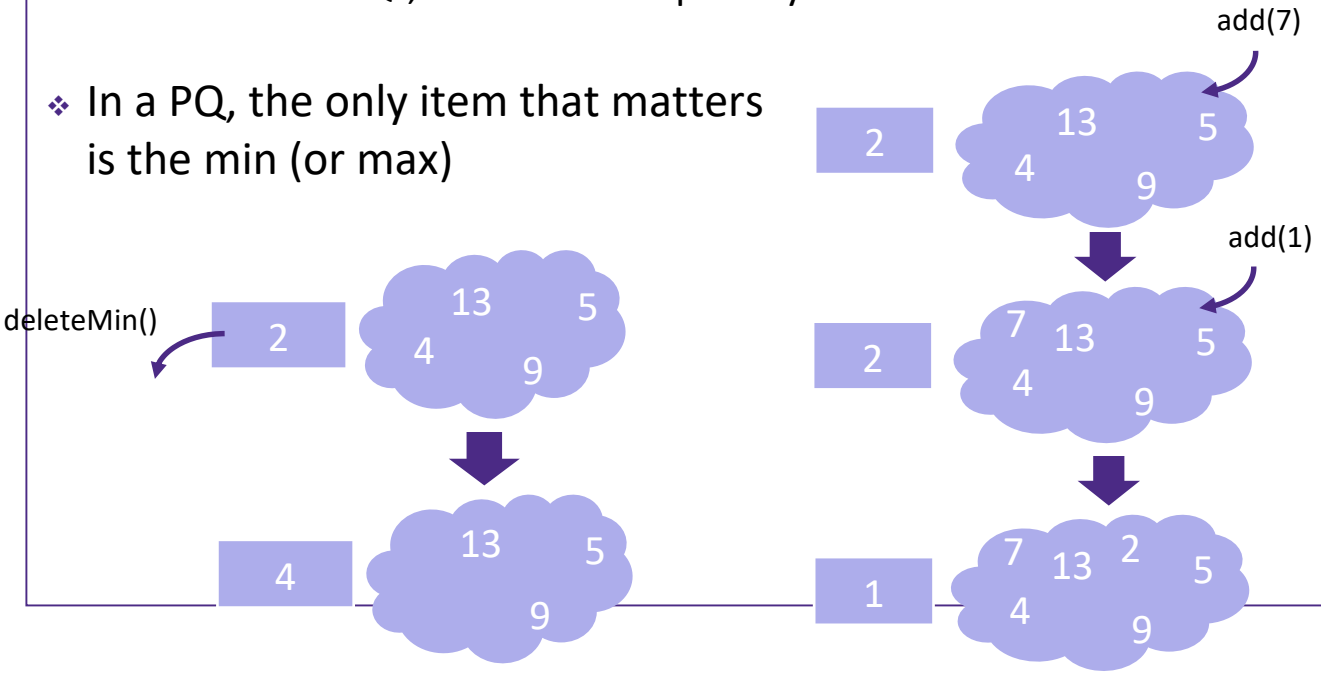
- **`add`**
- **`deleteMin`**

Key property:

- **`deleteMin`** removes and returns the "most important" item (lowest priority value)
- Can resolve ties arbitrarily

Priority Queues

- In lecture, we will study **min priority queues** but you may also see **max priority queues**
  - Same as minPQs, but invert the priority
- In a PQ, the only item that matters is the min (or max)



Priority Queue: Example

add *a* with priority *5*
add *b* with priority *3*
add *c* with priority *4*
w = **deleteMin**
x = **deleteMin**
add *d* with priority *2*
add *e* with priority *6*
y = **deleteMin**
z = **deleteMin**

```
  after execution:
6->e

w = b
x = c
y = d
z = a
```

Analogy: `add` is like `enqueue`, and `deleteMin` is like `dequeue`

Unlike queues, priority queues use *priorities* instead of *time-of-insertion* to order its elements

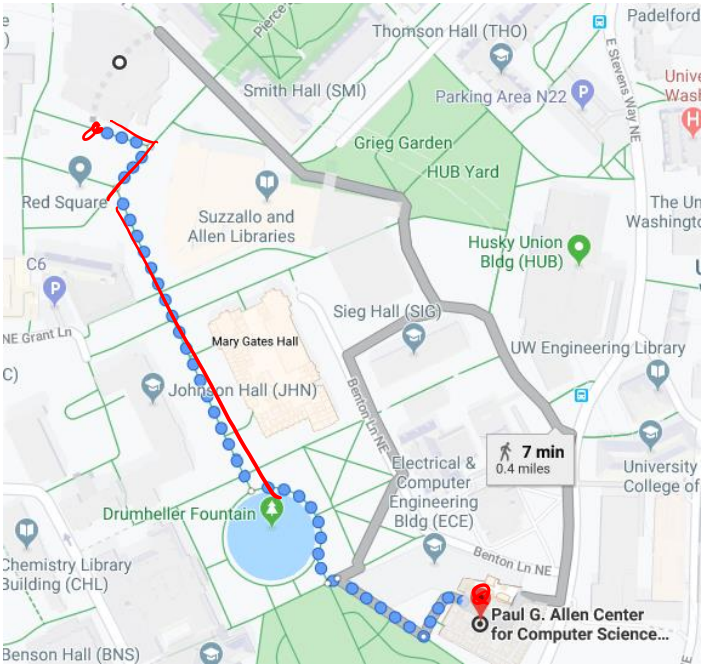
Priority Queue: Applications

- Run multiple programs in the operating system
  - "critical" before "interactive" before "compute-intensive"
- Triage (or treat) hospital patients in order of severity
- Order print jobs in order of decreasing length?
- Forward network packets by order of urgency
- Identify most frequently-used symbols for data compression
- Sorting!
  - **`add`** all elements, then repeatedly **`deleteMin`**

Priority Queue: More Applications

- Used heavily in **greedy algorithms**, where each phase of the algorithm picks the locally optimum solution

- Example: route finding
  - Represent a map as a series of *segments*
  - At each intersection, ask which segment gets you closest to the destination (ie, has max priority or min distance)



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
- The Priority Queue ADT is really cool!
  - We'll discuss data structures that implement it next lecture