Module 1: Introduction and Asymptotic Analysis

CS 240 – Data Structures and Data Management

A. Hunt and O. Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science,
University of Waterloo

Winter 2023

Outline

- CS240 overview
  - course objectives
  - course topics
- Introduction and Asymptotic Analysis
  - algorithm design
  - pseudocode
  - measuring efficiency
  - asymptotic analysis
  - analysis of algorithms
  - analysis of recursive algorithms
  - helpful formulas

Outline

- CS240 overview
    - course objectives
    - course topics
- Introduction and Asymptotic Analysis
    - algorithm design
    - pseudocode
    - measuring efficiency
    - asymptotic analysis
    - analysis of algorithms
    - analysis of recursive algorithms
    - helpful formulas

Course Objectives

- When first learn to program, emphasize *correctness*
  - does program output the expected results?
- This course is also concerned with *efficiency*
  - does program use computer resources efficiently?
    - processor time, memory space
- Strong emphasis on mathematical analysis of efficiency
- Will study efficient methods of *storing*, *accessing*, and performing *operations* on large collections of data

Course Objectives

- New abstract data types (ADTs)
  - how to implement ADT efficiently using appropriate data structures
    - typical operations in data structures
      - *inserting* new data items
      - *deleting* data items
      - *searching* for specific data items
- Algorithms
  - presented in pseudocode
  - analyzed using order notation (big-Oh, etc.)

Course Topics

▪ asymptotic (big-Oh) analysis	mathematical tool for efficiency
▪ priority queues and heaps ▪ sorting, selection ▪ binary search trees, AVL trees, B-trees	twists on data structures and algorithms you already know
▪ skip lists ▪ hashing	makes efficient dictionaries in practice
▪ quadtrees, kd-trees ▪ range search	searching data in multiple dimensions
▪ tries	special dictionary for strings
▪ string matching ▪ data compression	useful for unstructured data

CS Background

- Topics covered in previous courses with relevant sections [Sedgewick]
  - arrays, linked lists (Sec. 3.2–3.4)
  - strings (Sec. 3.6)
  - stacks, queues (Sec. 4.2–4.6)
  - abstract data types (Sec. 4-intro, 4.1, 4.8–4.9)
  - recursive algorithms (5.1)
  - binary trees (5.4–5.7)
  - sorting (6.1–6.4)
  - binary search (12.4)
  - binary search trees (12.5)
  - probability and expectation (Goodrich & Tamassia, Section 1.3.4)

Outline

- CS240 overview
    - Course objectives
    - Course topics
- Introduction and Asymptotic Analysis
    - algorithm design
    - pseudocode
    - measuring efficiency
    - asymptotic analysis
    - analysis of algorithms
    - analysis of recursive algorithms
    - helpful formulas

Algorithm Design Terminology

- **Problem:** given a problem instance, carry out a particular computational task
    - sort an input array A
- **Problem Instance:** *input* for the specified problem
    - A = [5, 2, 1, 8, 2]
- **Problem Solution:** *output* (correct answer) for the specified problem instance
    - A = [1, 2, 2, 5, 8]
- **Size of a Problem Instance** size(*I*)
    - a positive integer measuring size of instance *I*
    - size(A) = 5
    - often use n to denote instance size
        - often input is array, and instance size is array size

Algorithm Design Terminology

- **Algorithm:** *step-by-step process* (usually described in pseudocode) for carrying out a series of computations, given an arbitrary problem instance *I*
- **Algorithm solving a problem:** algorithm ***A*** *solves* problem ***Π*** if for every instance *I* of ***Π***, ***A*** computes a valid solution in finite time
- **Program:** *implementation* of an algorithm using a specified computer language
- In this course, the emphasis is on algorithms
  - as opposed to programs or programming

Algorithms in Practice

- For a problem ***Π***, can have many algorithms
- Given a problem ***Π***
  1. **Algorithm Design:** design algorithm **A** that solves ***Π***
  2. **Algorithm Analysis:** assess *correctness* and *efficiency* of ***A***
  3. **Implementation**: if acceptable (correct and efficient), implement ***A***
     - many possible programs implementing ***A***

Outline

- CS240 overview
    - Course objectives
    - Course topics
- Introduction and Asymptotic Analysis
    - algorithm design
    - pseudocode
    - measuring efficiency
    - asymptotic analysis
    - analysis of algorithms
    - analysis of recursive algorithms
    - helpful formulas

Pseudocode

- Pseudocode is a method of communicating algorithm to a human
  - whereas program (implementation) is a method of communicating algorithm to a computer

Test3(A, n)
A: array of size n

1. **for** $i \leftarrow 1$ **to** $n-1$ **do**
2. $\quad j \leftarrow i$
3. $\quad$ **while** $j > 0$ and $A[j] > A[j-1]$ **do**
4. $\quad\quad$ swap $A[j]$ and $A[j-1]$
5. $\quad\quad j \leftarrow j - 1$

- Pseudocode
  - preferred language for describing algorithms
  - omits obvious details, e.g. variable declarations
  - sometimes uses English descriptions
  - has limited if any error detection
  - sometimes uses mathematical notation

Pseudocode Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - indentation replaces braces
- Expressions
  - $\leftarrow$ assignment
  - $==$ equality testing
  - $\boldsymbol{n}^2$ superscripts and other mathematical formatting allowed
- Method declaration
  - **Algorithm** ***method*** (***arg, arg***…)
  - Input …
  - Output …

Algorithm ***arrayMax***(***A***, ***n***)

Input: array A of n integers

Output: maximum element of A

$currentMax \leftarrow \boldsymbol{A}[0]$

for $\boldsymbol{i} \leftarrow 1$ **to** $\boldsymbol{n} - 1$ **do**

  if $A[i] > currentMax$ **then**

    $currentMax \leftarrow A[i]$

return $currentMax$

Outline

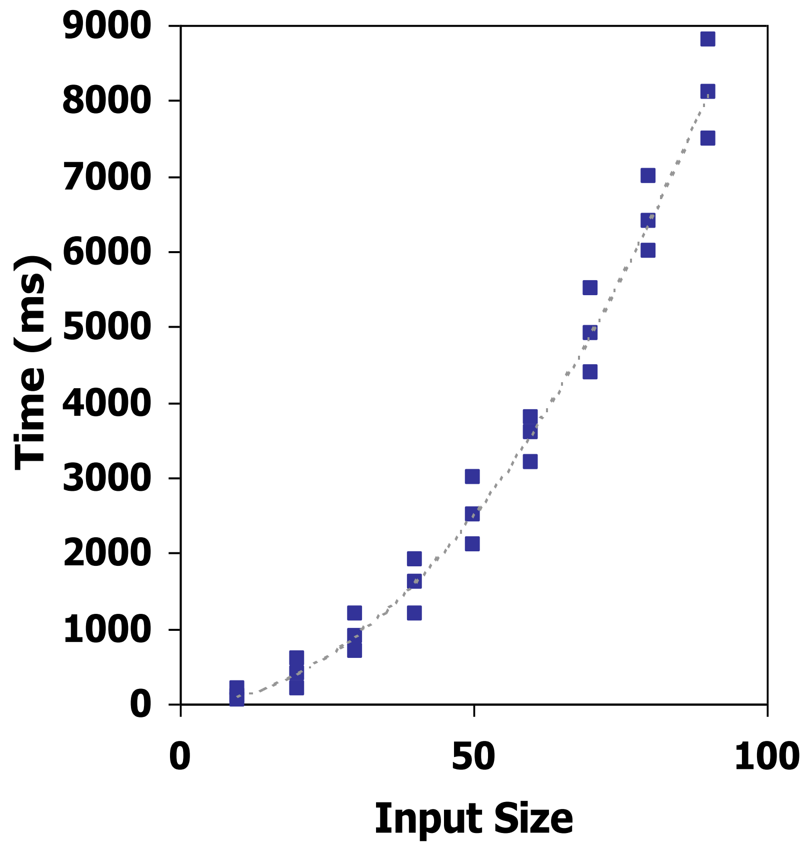
- CS240 overview
  - Course objectives
  - Course topics
- Introduction and Asymptotic Analysis
  - algorithm design
  - pseudocode
  - measuring efficiency
  - asymptotic analysis
  - analysis of algorithms
  - analysis of recursive algorithms
  - helpful formulas

Efficiency of Algorithms/Programs

- How decide which algorithm or program is the most efficient for a given problem?
- Efficiency
  - **time:** *amount of time* program takes to run
    - also called time complexity
  - **space:** *amount of memory* program requires
    - also called space complexity
- Efficiency depends on *size(I)*, size of a given problem instance *I*
  - efficiency is a function of input size
- Primarily concerned with time efficiency in this course

Running Time of Algorithms/Programs

- One option: *experimental studies*
  - write program implementing the algorithm
  - run program with inputs of *varying size* and *composition*
  - can use clock() from time.h, to measure running time
  - plot/compare results



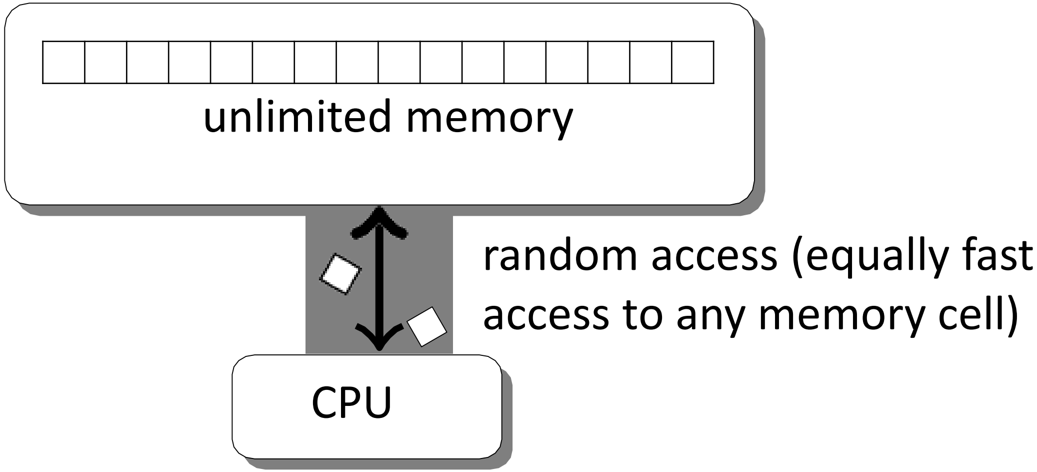
Running Time of Algorithms/Programs

- Shortcomings of experimental studies
  - implementation may be complicated/costly
  - timings are affected by many factors
    - *hardware* (processor, memory)
    - *software environment* (OS, compiler, programming language)
    - *human factors* (programmer)
  - cannot test all inputs, hard to select good *sample inputs*
  - thus cannot easily compare two algorithms/programs
- Want framework that
  - does not require implementing the algorithm
  - independent of hardware/software environment
  - takes into account all possible input instances

Theoretical Framework For Algorithm Analysis

- To overcome dependency on hardware/software
    - write algorithms in pseudo-code
        - language independent
    - “run” algorithms on idealized computer model
        - allows to reason about efficiency

Idealized Computer Model



- **Random Access Machine (RAM)** Model
  - has a set of memory cells, each of which stores one data item
    - memory cells are big enough to hold stored items
  - any *access to a memory location* takes constant time
  - run *primitive operations* on this machine
    - primitive operation takes constant time
- Simplified model
  - most of these assumptions are not valid for a real computer

Theoretical Framework For Algorithm Analysis

- To overcome dependency on hardware/software
  - write algorithms in pseudo-code
    - language independent
  - “run” algorithms on idealized computer model
    - allows to reason about efficiency
  - instead of time, count number of *primitive operations*
    - assume all primitive operations take the same time
  - measure time efficiency of an algorithm in terms of growth rate
    - avoids complicated functions and isolates the factor that effects the efficiency the most for large inputs
- This framework makes many simplifying assumptions
  - makes analysis of algorithms easier

Theoretical Analysis of Running time

- Pseudocode is a sequence of *primitive operations*
- A primitive operation is
  - independent of input size
- Examples of Primitive Operations
  - addition, subtraction, etc.
    - $x \cdot n$ is a primitive operation
    - x^n is not a primitive operation, runtime depends on input size n
  - assigning a value to a variable
  - indexing into an array
  - returning from a method
  - exact definition not important
    - will see why later

Algorithm ***arrayMax***(***A***, ***n***)
  Input: array *A* of *n* integers
  Output: maximum element of *A*
  currentMax ← ***A***[0]
  for ***i*** ← 1 **to** ***n*** − 1 **do**
    if *A*[*i*] > *currentMax* **then**
      currentMax ← *A*[*i*]
  return *currentMax*

- To find running time, count the number of primitive operations
  - as a function of input size $\boldsymbol{n}$

Theoretical Analysis of Running time

- To find running time, count the number of primitive operations $T(\boldsymbol{n})$
  - function of input size $\boldsymbol{n}$

Algorithm ***arrayMax***(***A***, ***n***)	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	
if $A[i] > currentMax$ **then**	
$currentMax \leftarrow A[i]$	
{ increment counter ***i*** }	
return $currentMax$	

Theoretical Analysis of Running time

- To find running time, count the number of primitive operations $T(\boldsymbol{n})$
  - function of input size $\boldsymbol{n}$

Algorithm ***arrayMax***(***A***, ***n***) — # operations

currentMax $\leftarrow$ *A*[0] — 2

for *i* $\leftarrow$ 1 **to** *n* – 1 **do**

if *A*[*i*] > *curr*

curr

{ increment coun

return *currentMa*

$i \leftarrow 1$

$n - 1$

$i = 1$, check $i < n - 1$ (enter inside loop)

$i = 2$, check $i < n - 1$ (enter inside loop)

…

$i = n - 1$, check $i < n - 1$(enter inside loop)

$i = n$, check $i < n - 1$(do not enter inside loop)

Total: 2+n

Theoretical Analysis of Running time

- To find running time, count the number of primitive operations $T(\boldsymbol{n})$
  - function of input size $\boldsymbol{n}$

Algorithm ***arrayMax***(***A***, ***n***)	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2 + \boldsymbol{n}$
if $A[i] > currentMax$ **then**	$2(\boldsymbol{n} - 1)$
$currentMax \leftarrow A[i]$	$2(\boldsymbol{n} - 1)$
{ increment counter ***i*** }	$2(\boldsymbol{n} - 1)$
return *currentMax*	1
	Total: $7\boldsymbol{n} - 1$

Theoretical Analysis of Running time: Multiplicative factors

- Algorithm ***arrayMax*** executes $\boldsymbol{T(n)} = 7\boldsymbol{n} - 1$ primitive operations
- Let a = time taken by fastest primitive operation

  b = time taken by slowest primitive operation
- $\boldsymbol{T(n)}$ is bounded by two linear functions

$$a\,(7\boldsymbol{n} - 1) \le \boldsymbol{T(n)} \le b(7\boldsymbol{n} - 1)$$

- Changing hardware/software environment affects $\boldsymbol{T(n)}$ by a multiplicative constant factor
- $\boldsymbol{T(n)} = const \cdot n$ [ignoring the subtracted constant]
  - $const$ will change depending on software/hardware environment
- Want to say $\boldsymbol{T(n)} = 7\boldsymbol{n} - 1$ is essentially $\boldsymbol{n}$
- **Want to ignore constant multiplicative factors**

Theoretical Analysis of Running time: Lower Order Terms

- Running time on small inputs hardly ever matters
  - consider behaviour of algorithms for large input sizes
  - further simplifies running time analysis
- Consider $\boldsymbol{T}(\boldsymbol{n}) = \boldsymbol{n}^2 + \boldsymbol{n}$
- For large $\boldsymbol{n}$, only the fastest growing factor is important

$$\boldsymbol{T}(100{,}000) = 10{,}000{,}000{,}000 + 100{,}000$$

- **Want to ignore slower growing terms**

Theoretical Analysis of Running time

- Thus we want to ignore
  - multiplicative constant factors
  - lower-order (slower growing) terms
- This means focusing on the *growth rate* of the function
  - $10\boldsymbol{n}^2 + 100\boldsymbol{n}$ has growth rate of $\boldsymbol{n^2}$
  - $10\boldsymbol{n} + 10$ has growth rate of $\boldsymbol{n}$
- Asymptotic analysis (i.e. order notation) gives tools to formally focus on the growth rate

Outline

- CS240 overview
  - Course objectives
  - Course topics
- Introduction and Asymptotic Analysis
  - algorithm design
  - pseudocode
  - measuring efficiency
  - asymptotic analysis
  - analysis of algorithms
  - analysis of recursive algorithms
  - helpful formulas

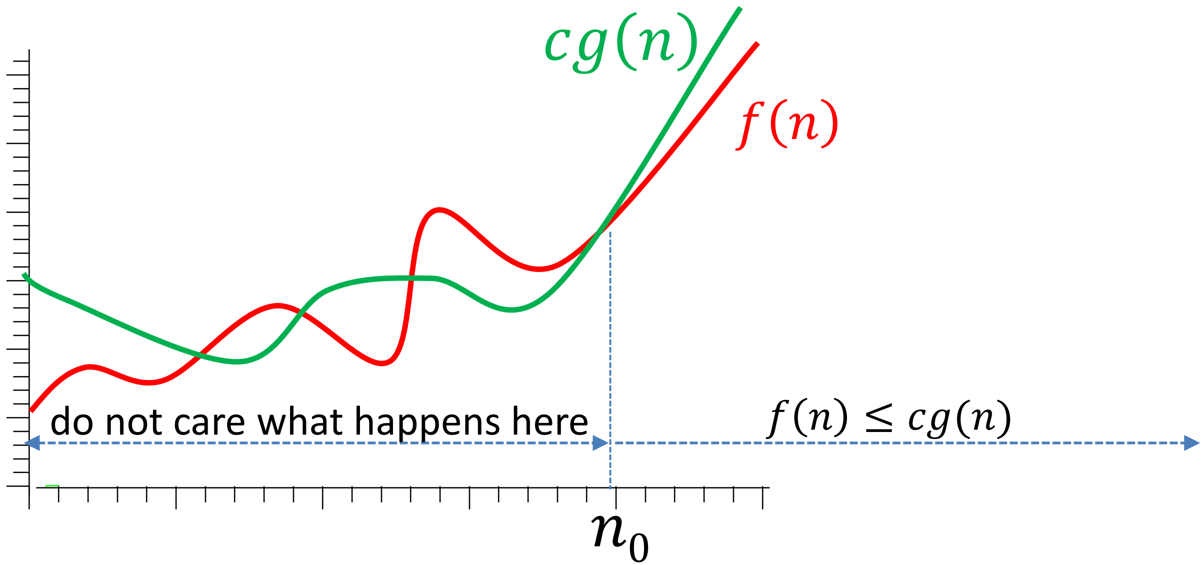
Order Notation: big-Oh

- Bound from above by function expressing "growth rate"

$f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t.

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

a set of functions



- Need c to "get rid" of multiplicative constant in the growth rate
  - cannot say $5n^2 \leq n^2$, but can say $5n^2 \leq cn^2$ for some constant c
- Absolute value signs are not relevant for analysis of run-time or space, but useful in other applications of asymptotic notation

big-Oh Example

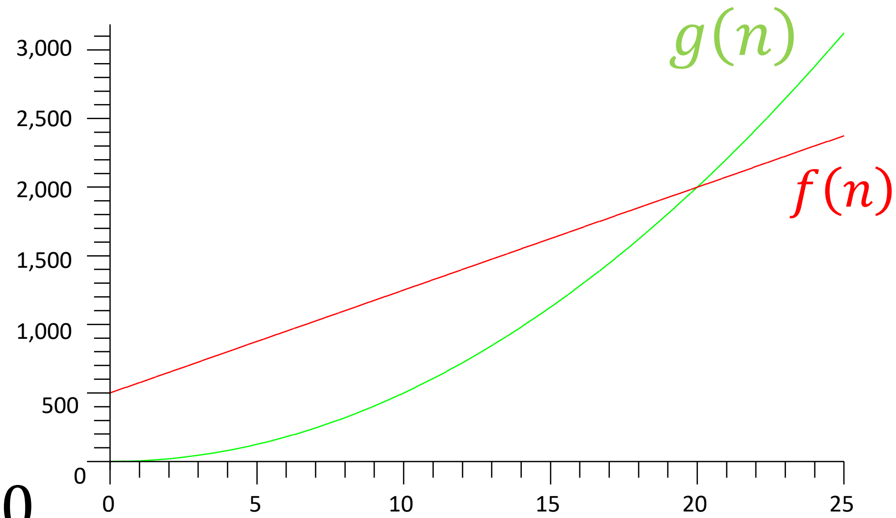
O-notation

$f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t.

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

- Example:

$$f(n) = 75n + 500$$
$$g(n) = 5n^2$$



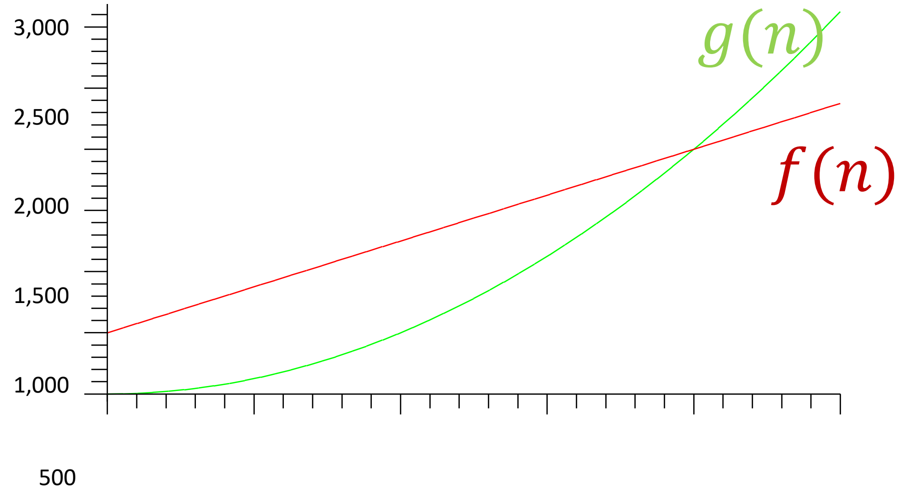
- Take $c = 1, n_0 = 20$
- Can also take $c = 10, n_0 = 30$

Order Notation: big-Oh

$f(n) \in O(g(n))$

if there exist constants $c > 0$ and $n_0 \geq 0$

s.t. $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$



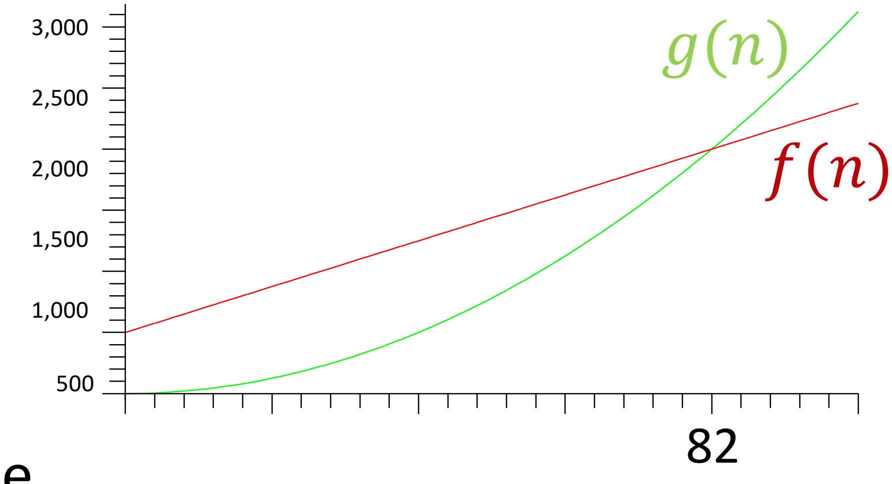
- Big-O gives asymptotic *upper bound*
  - $f(n) \in O(g(n))$ means function $f(n)$ is "bounded" above by function $g(n)$
    1. eventually, for large enough n
    2. ignoring multiplicative constant
  - Growth rate of $f(n)$ is slower or the same as growth rate of $g(n)$
- Use big-O to bound the growth rate of algorithm
  - $f(n)$ for running time
  - $g(n)$ for growth rate
    - should choose $g(n)$ as simple as possible
- Saying $f(n)$ is $O(g(n))$ is equivalent to saying $f(n) \in O(g(n))$

Order Notation: big-Oh

$f(n) \in O(g(n))$

if there exist constants $c > 0$ and $n_0 \geq 0$

s.t. $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$



- Choose $g(n)$ as simple as possible
- Previous example: $f(n) = 75n + 500,\ g(n) = 5n^2$
- Simpler function for growth rate: $g(n) = n^2$
- Can show $f(n) \in O(g(n))$ as follows
  - set $f(n) = g(n)$ and solve the resulting quadratic equation
  - intersection point is $n = 82$
  - take $c = 1, n_0 = 82$

Order Notation: big-Oh

$f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$
s.t. $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$

- Do not have to solve quadratic equation
- $f(n) = 75n + 500,\ g(n) = n^2$
- Show $f(n) \in O(g(n))$

$$75n + 500 \leq 75n^2 + 500n^2 = 575n^2$$

for all $n \geq 1$

- take $c = 575, n_0 = 1$

Order Notation: big-Oh

$$f(n) \in O\big(g(n)\big) \text{ if there exist constants } c > 0 \text{ and } n_0 \geq 0$$
$$\text{s.t. } |f(n)| \leq c|g(n)| \text{ for all } n \geq n_0$$

- Better (i.e. “tighter”) bound on growth
  - can bound $f(n) = 75n + 500$ by a function growing slower than $g(n) = n^2$
- $f(n) = 75n + 500, \; g(n) = n$
- Show $f(n) \in O\big(g(n)\big)$

$$75n + 500 \leq 75n + 500n = 575n$$
$$\text{for all } n \geq 1$$

- take $c = 575, n_0 = 1$

More big-O Examples

- Prove that

$$2n^2 + 3n + 11 \in O(n^2)$$

- Need to find $c > 0$ and $n_0 \geq 0$ s.t.

$$2n^2 + 3n + 11 \leq cn^2 \text{ for all } n \geq n_0$$

$$2n^2 + 3n + 11 \leq 2n^2 + 3n^2 + 11n^2 = 16n^2$$

for all $n \geq 1$

- Take $c = 16, n_0 = 1$

More big-O Examples

- Prove that

$$2n^2 - 3n + 11 \in O(n^2)$$

- Need to find $c > 0$ and $n_0 \geq 0$ s.t.

$$2n^2 - 3n + 11 \leq cn^2 \text{ for all } n \geq n_0$$

$$2n^2 - 3n + 11 \leq 2n^2 + 0 + 11n^2 = 13n^2$$

for all $n \geq 1$

- Take $c = 13, n_0 = 1$

More big-O Examples

- Have to be careful with logs
- Prove that

$$2n^2 \log n + 3n \in O(n^2 \log n)$$

- Need to find $c > 0$ and $n_0 \geq 0$ s.t.

$$2n^2 \log n + 3n \leq cn^2 \log n \quad \text{for all } n \geq n_0$$

$$2n^2 \log n + 3n \leq 2n^2 \log n + 3n^2 \log n \leq 5n^2 \log n$$

$\cancel{\text{for all } n \geq 1}$

$\text{for all } n \geq 2$

- Take $c = 5, n_0 = 2$

Theoretical Analysis of Running time

- To find running time, count the number of primitive operations $T(\boldsymbol{n})$
  - function of input size $\boldsymbol{n}$
- Last step: express the running time using asymptotic notation

Algorithm ***arrayMax***(***A***, ***n***)	# operations
currentMax ← *A*[0]	c_1
for *i* ← 1 **to** *n* − 1 **do**	
if *A*[*i*] > *currentMax* **then**	$c_2 n$
currentMax ← *A*[*i*]	
{ increment counter ***i*** }	
return *currentMax*	c_3

Total: $c_1 + c_3 + c_2 n$ which is $O(n)$

Theoretical Analysis of Running time

- To find running time, count the number of primitive operations $T(\boldsymbol{n})$
  - function of input size $\boldsymbol{n}$
- Last step: express the running time using asymptotic notation

Algorithm ***arrayMax***(***A***, ***n***) # operations

currentMax ← *A*[0] c

for *i* ← 1 **to** *n* − 1 **do**

  if *A*[*i*] > *currentMax* **then**

    currentMax ← *A*[*i*]

{ increment counter ***i*** } cn

return *currentMax* c

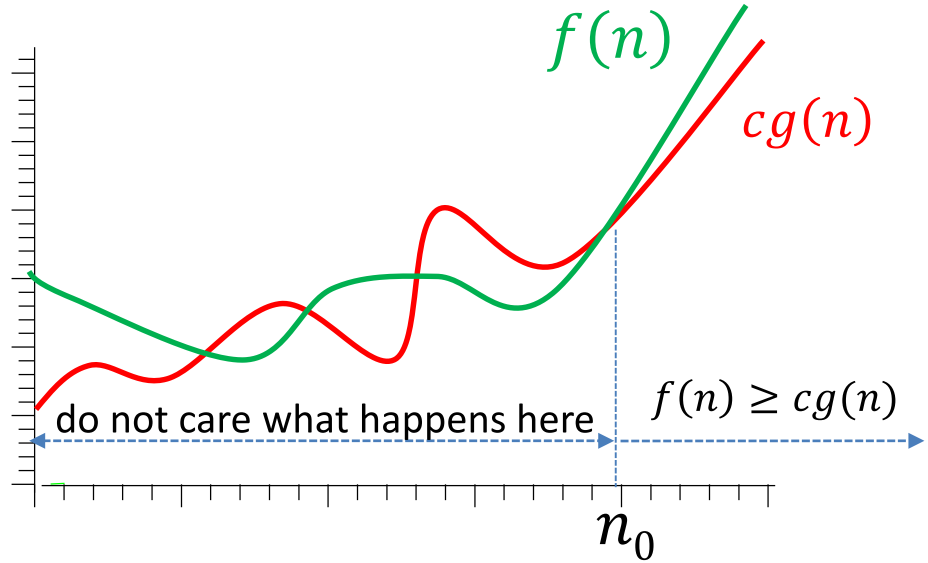
Total: $c + cn$ which is $O(n)$

Need for Asymptotic Tight bound

- $2n^2 + 3n + 11 \in O(n^2)$
- But also $2n^2 + 3n + 11 \in O(n^{10})$
  - this is a true but hardly a useful statement
  - analogy: if I say I have less than a million $ in my pocket, it is true, but useless statement
  - i.e. this statement does not give a tight upper bound
  - a bound is tight if it uses the slowest grown function possible
- Want an asymptotic notation that guarantees a tight bound
- On our way to tight bound, we first need

  an asymptotic *lower bound*

Aymptotic Lower Bound



- Ω-notation (asymptotic lower bound)

$$f(n) \in \Omega(g(n)) \text{ if there exist constants } c > 0 \text{ and } n_0 \geq 0$$
$$\text{s.t. } |f(n)| \geq c|g(n)| \text{ for all } n \geq n_0$$

- $f(n) \in \Omega\big(g(n)\big)$ means function $f(n)$ is asymptotically bounded below by function $g(n)$
  1. eventually, for large enough n
  2. ignoring multiplicative constant
- Growth rate of $f(n)$ is larger or the same as growth rate of $g(n)$

Asymptotic Lower Bound

$f(n) \in \Omega(g(n))$ if $\exists$ constants $c > 0,\ n_0 \geq 0$ s.t. $|f(n)| \geq c|g(n)|$ for $n \geq n_0$

- Prove that $2n^2 + 3n + 11 \in \Omega(n^2)$
- Find $c > 0$ and $n_0 \geq 0$ s.t. $2n^2 + 3n + 11 \geq cn^2$ for all $n \geq n_0$

  $2n^2 + 3n + 11 \geq 2n^2$ for all $n \geq 1$

- Take $c = 2, n_0 = 1$

Asymptotic Lower Bound

$f(n) \in \Omega(g(n))$ if $\exists$ constants $c > 0,\ n_0 \geq 0$ s.t. $|f(n)| \geq c|g(n)|$ for $n \geq n_0$

- Prove that $\frac{1}{2}n^2 - 5n \in \Omega(n^2)$
  - $\frac{1}{2}n^2 - 5n < 0$ for $0 < n < 10$
  - since we ignore absolute value in the derivation, we need to ensure $f(n)$ is actually positive
  - for positivity of $f(n)$, make sure to take $n_0 \geq 10$
- Need to find c and n_0 s.t. $\frac{1}{2}n^2 - 5n \geq cn^2$ for all $n \geq n_0$
- Unlike before, cannot 'drop' lower growing term, as $\frac{1}{2}n^2 - 5n \leq \frac{1}{2}n^2$
- Need $\frac{1}{2}n^2 - 5n \geq cn^2$

$$an^2 \quad bn^2$$

positive for large enough n

for large enough n

$$\frac{1}{2}n^2 - 5n \geq an^2 + (bn^2 - 5n) \geq an^2$$

Asymptotic Lower Bound

$f(n) \in \Omega(g(n))$ if $\exists$ constants $c > 0,\ n_0 \geq 0$ s.t. $|f(n)| \geq c|g(n)|$ for $n \geq n_0$

- For positivity of $f(n)$, make sure to take $n_0 \geq 10$
- Need to find c and n_0 s.t. $\frac{1}{2}n^2 - 5n \geq cn^2$ for all $n \geq n_0$
- Rewrite

$$\frac{1}{2}n^2 - 5n = \frac{1}{4}n^2 + \frac{1}{4}n^2 - 5n = \frac{1}{4}n^2 + \underbrace{\left(\frac{1}{4}n^2 - 5n\right)}_{\geq 0,\text{ if } n \geq 20} \geq \frac{1}{4}n^2$$

so take $n_0 \geq 20$

- Take $c = \frac{1}{4}, n_0 = 20$
  - n_0 happened to be bigger than 10, as needed, automatically

Tight Asymptotic Bound

- Θ-notation

  $f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0, n_0 \geq 0$ s.t.
  $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$

- Easy to prove that

  $$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))$$

- Therefore, to show that $f(n) \in \Theta(g(n))$, it is enough to show
  1. $f(n) \in O(g(n))$
  2. $f(n) \in \Omega(g(n))$
     - that's why we said that for tight bound, we also need lower bound
- $f(n) \in \Theta\big(g(n)\big)$ means $f(n)$, $g(n)$ have equal growth rates

Tight Asymptotic Bound

- Proved previously that
  - $2n^2 + 3n + 11 \in O(n^2)$
  - $2n^2 + 3n + 11 \in \Omega(n^2)$
- Thus $2n^2+3n + 11 \in \Theta(n^2)$
- Ideally, should use Θ to determine growth rate of algorithm
  - $f(n)$ for running time
  - $g(n)$ for growth rate
- Sometimes determining tight bound is hard, so big-O is used

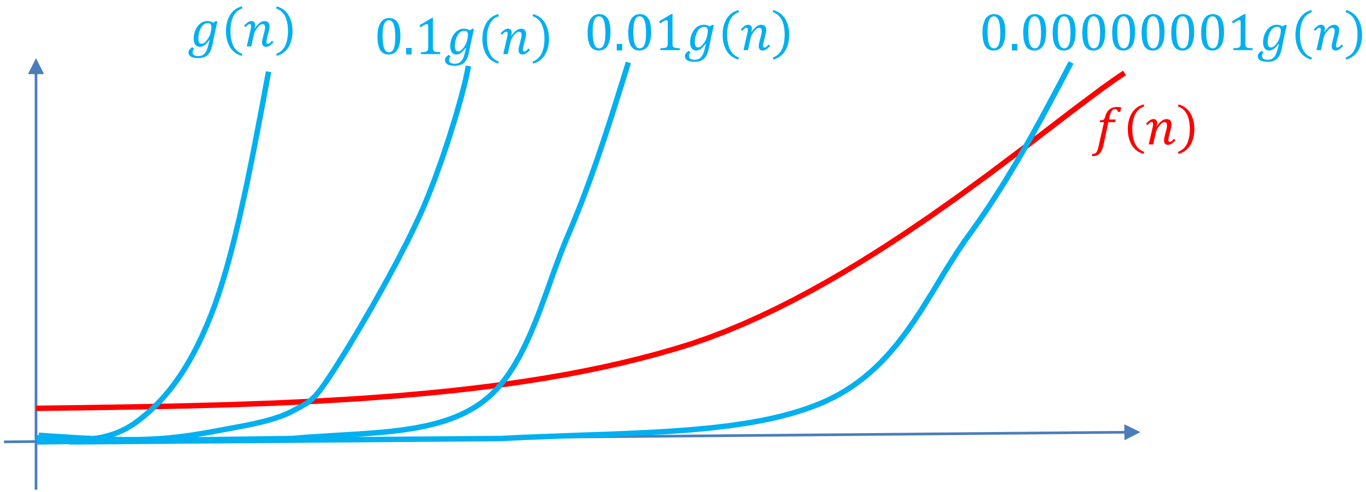
Tight Asymptotic Bound

Prove that $\log_b n \in \Theta(\log n)$ for $b > 1$

- Find $c_1, c_2 > 0, n_0 \geq 0$ s.t. $c_1 \log n \leq \log_b n \leq c_2 \log n$ for all $n \geq n_0$
- $\log_b n = \frac{1}{\log b} \log n$
- $\frac{1}{\log b} \log n \leq \log_b n \leq \frac{1}{\log b} \log n$
- Since $b > 1, \log b > 0$
- Take $c_1 = c_2 = \frac{1}{\log b}$ and $n_0 = 1$

Strictly Smaller Asymptotic Bound

- $f(n) = 2n^2 + 3n + 11 \in \Theta(n^2)$
- How to say $f(n)$ is asymptotically strictly smaller than $g(n) = n^3$?



- *o*-notation

  $f(n) \in o(g(n))$ if **for any constant** $c > 0$, there exists a constant $n_0 \geq 0$ s.t. $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$
- Meaning: f grows much slower than g

Strictly Larger Asymptotic Bound

- ω-notation

  $f(n) \in \omega(g(n))$ if **for any constant** $c > 0$, there exists a constant $n_0 \geq 0$ s.t. $|f(n)| \geq c|g(n)|$ for all $n \geq n_0$

- Meaning: f grows much faster than g

Strictly Smaller Proof Example

$f(n) \in o(g(n))$ if **for any** $c > 0$, there exists $n_0 \geq 0$ s.t. $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$

Prove that $5n \in o(n^2)$

- Given $c > 0$ need to find n_0 s.t. $5n \leq cn^2$ for all $n \geq n_0$
- Dividing both sides by n , this is equivalent to the statement below
- Given $c > 0$ need to find n_0 s.t. $5 \leq cn$ for all $n \geq n_0$
    - holds for for $n \geq \frac{5}{c}$
- Therefore, $5n \leq cn^2$ for $n \geq \frac{5}{c}$
- Take $n_0 = \frac{5}{c}$

Limit Theorem for Order Notation

- So far had proofs for order notation from the *first principles*
    - i.e. from the definition
- There is a useful limit theorem for order notation
- Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$
- Suppose that $\mathrm{L} = \lim_{n \to \infty} \frac{f(n)}{g(n)}$
- Then $f(n) \in \begin{cases} o\big(g(n)\big) & if\ L = 0 \\ \Theta\big(g(n)\big) & if\ 0 < L < \infty \\ \omega\big(g(n)\big) & if\ L = \infty \end{cases}$
- The required limit can often be computed using l'Hopital's rule
- Theorem gives sufficient but not necessary conditions

Example 1

Let $f(n)$ be a polynomial of degree $d \geq 0$ with $c_d > 0$

$$f(n) = c_d n^d + c_{d-1} n^{d-1} + \cdots + c_1 n + c_0$$

Then $f(n) \in \Theta\left(n^d\right)$

Proof:

$$\lim_{n \to \infty} \frac{f(n)}{n^d} = \lim_{n \to \infty} \left(\frac{c_d n^d}{n^d} + \frac{c_{d-1} n^{d-1}}{n^d} + \cdots + \frac{c_0}{n^d} \right)$$

$$= \underbrace{\lim_{n \to \infty} \left(\frac{c_d n^d}{n^d} \right)}_{= c_d} + \underbrace{\lim_{n \to \infty} \left(\frac{c_{d-1} n^{d-1}}{n^d} \right)}_{= 0} + \cdots + \underbrace{\lim_{n \to \infty} \left(\frac{c_0}{n^d} \right)}_{= 0}$$

$$= c_d \quad > 0$$

Example 2

- Compare growth rates of $\log n$ and n

$$\lim_{n\to\infty}\frac{\log n}{n} = \lim_{n\to\infty}\frac{\frac{\ln n}{\ln 2}}{n} = \lim_{n\to\infty}\frac{\frac{1}{\ln 2 \cdot \mathrm{n}}}{1} = \lim_{n\to\infty}\frac{1}{\mathrm{n}\cdot\ln 2} = 0$$

L'Hopital rule

- $\log n \in o(n)$

Example 3

- Prove $(\log n)^a \in o(n^d)$, for any (big) $a > 0$, (small) $d > 0$

1) Prove (by induction):

$$\lim_{n\to\infty} \frac{\ln^k n}{n} = 0 \text{ for any integer } k$$

- Base case $k = 1$ is proven on previous slide
- Inductive step: suppose true for $k - 1$
- $$\lim_{n\to\infty} \frac{\ln^k n}{n} \underset{\text{L'Hopital rule}}{=} \lim_{n\to\infty} \frac{\frac{1}{n} k\, ln^{k-1} n}{1} = k \lim_{n\to\infty} \frac{ln^{k-1} n}{n} = 0$$

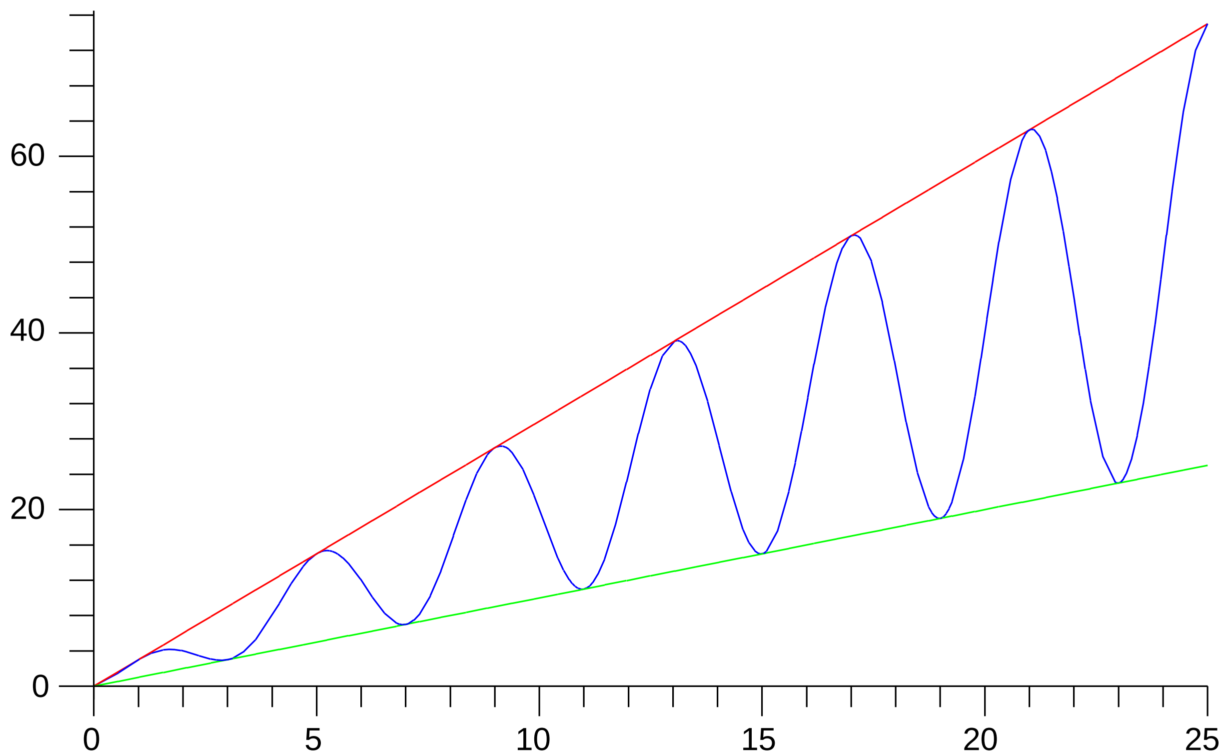
2) Prove $\lim_{n\to\infty} \frac{\ln^a n}{n^d} = 0$

- $$\lim_{n\to\infty} \frac{\ln^a n}{n^d} = \left(\lim_{n\to\infty} \frac{\ln^{a/d} n}{n}\right)^d \leq \left(\lim_{n\to\infty} \frac{\ln^{\lceil a/d \rceil} n}{n}\right)^d = 0$$

3) Finally $\lim_{n\to\infty} \frac{(\log n)^a}{n^d} = \lim_{n\to\infty} \frac{\left(\frac{\ln n}{ln 2}\right)^a}{n^d} = \left(\frac{1}{ln 2}\right)^a \lim_{n\to\infty} \frac{(\ln n)^a}{n^d} = 0$

Example 4

- Sometimes limit does not exist, but can prove from first principles
- Let $f(n) = n(2 + \sin n\pi/2)$
- Prove that $f(n)$ is $\Theta(n)$



Example 4

- Let $f(n) = n(2 + \sin n\pi/2)$, prove that $f(n)$ is $\Theta(n)$
- Proof:

$$-1 \leq sin(\text{any number}) \leq 1$$

$$f(n) \leq n(2 + 1) = 3n \quad \text{for all } n \geq 1$$

$$n = n(2 - 1) \leq f(n) \quad \text{for all } n \geq 1$$

$$n \leq f(n) \leq 3n \quad \text{for all } n \geq 1$$

Use $c_1 = 1, c_2 = 3, n_0 = 1$

Order notation Summary

- $f(n) \in \Theta(g(n))$: growth rates of f and g are the same
- $f(n) \in o(g(n))$: growth rate of f is less than growth rate of g
- $f(n) \in \omega\left(g(n)\right)$: growth rate of f is greater than growth rate of g
- $f(n) \in O\left(g(n)\right)$: growth rate of f is the same or less than growth rate of g
- $f(n) \in \Omega(g(n))$: growth rate of f is the same or greater than growth rate of g

Relationship between Order Notations

One can prove the following relationships

- $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$
- $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
- $f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \notin \Omega(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \notin O(g(n))$

Algebra of Order Notations

- The following rules are easy to prove

1. **Identity rule:** $f(n) \in \Theta\big(f(n)\big)$
2. **Transitivity**
   - if $f(n) \in O\big(g(n)\big)$ and $g(n) \in O\big(h(n)\big)$ then $f(n) \in O\big(h(n)\big)$
   - if $f(n) \in \Omega\big(g(n)\big)$ and $g(n) \in \Omega\big(h(n)\big)$ then $f(n) \in \Omega\big(h(n)\big)$
3. **Maximum rules**

Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$, then

a) $f(n) + g(n) \in \Omega(max\{f(n), g(n)\})$

b) $f(n) + g(n) \in O(max\{f(n), g(n)\})$

Proof:

a) $max\{f(n), g(n)\} =$ either $f(n)$ or $g(n) \leq f(n) + g(n)$

b)
$$
\begin{aligned}
f(n) + g(n) &= max\{f(n), g(n)\} + min\{f(n), g(n)\} \\
&\leq max\{f(n), g(n)\} + max\{f(n), g(n)\} \\
&= 2max\{f(n), g(n)\}
\end{aligned}
$$

Abuse of Notation

- Normally, we say $f(n) \in \Theta(g(n))$ because $\Theta(g(n))$ is a set
- Sometimes convenient to abuse of notation, i.e.
  - $f(n) = n^2 + \Theta(n)$
    - $f(n)$ is a quadratic function plus a linear term
  - $f(n) = n^2 + O(n)$
    - $f(n)$ is a quadratic function plus a term that grows slower or at the same rate as a linear function
  - $f(n) = n^2 + O(1)$
    - $f(n)$ is a quadratic function plus a constant
  - $f(n) = n^2 + o(1)$
    - $f(n)$ is a quadratic function plus a term that goes to 0

Common Growth Rates

- Commonly encountered growth rates in increasing order of growth
  - $\Theta(1)$ *constant complexity*
  - $\Theta(\log n)$ *logarithmic complexity*
  - $\Theta(n)$ *linear complexity*
  - $\Theta(n \log n)$ *linearithmic*
  - $\Theta(n \log^k n)$ *quasi-linear* (k is constant, i.e. independent of the problem size)
  - $\Theta(n^2)$ *quadratic complexity*
  - $\Theta(n^3)$ *cubic complexity*
  - $\Theta(2^n)$ *exponential complexity*

How Growth Rates Affect Running Time

- How running time affected when problem size doubles ($n \rightarrow 2n$)
  - constant complexity: $T(n) = c$ $\quad T(2n) = c$
  - logarithmic complexity: $T(n) = c \log n$ $\quad T(2n) = T(n) + c$
  - linear complexity: $T(n) = cn$ $\quad T(2n) = 2T(n)$
  - linearithmic: $T(n) = cn \log n$ $\quad T(2n) = 2T(n) + 2cn$
  - quadratic complexity: $T(n) = cn^2$ $\quad T(2n) = 4T(n)$
  - cubic complexity: $T(n) = cn^3$ $\quad T(2n) = 8T(n)$
  - exponential complexity: $T(n) = c2^n$ $\quad T(2n) = \frac{1}{c} T^2(n)$

Comparison of Growth Rates

n	$\log(n)$	n	$n\log(n)$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4096	65536
32	5	32	160	1024	32768	4.3×10^{9}
64	6	64	384	4096	262144	1.8×10^{19}
128	7	128	896	16384	2097152	3.4×10^{38}
256	8	256	2048	65536	16777218	1.2×10^{77}

Outline

- CS240 overview
    - Course objectives
    - Course topics
- Introduction and Asymptotic Analysis
    - algorithm design
    - pseudocode
    - measuring efficiency
    - analysis of algorithms
    - analysis of recursive algorithms
    - helpful formulas

Techniques for Algorithm Analysis

- Goal: Use asymptotic notation to simplify run-time analysis
- Running time of an algorithm depends on the *input size* n

```
Test1(n)
1.  sum ← 0
2.  for i ← 1 to n do
3.      for j ← i to n do
4.          sum ← sum + (i − j)^2
5.  return sum
```

- Identify *primitive operations* that require $\Theta(1)$ time
- Loop complexity expressed as *sum* of complexities of each iteration
- Nested loops: start with the innermost loop and proceed outwards
- This gives *nested summations*

Techniques for Algorithm Analysis

- Goal: Use asymptotic notation to simplify run-time analysis
- Running time of an algorithm depends on the *input size* n

Test1(n)

1. $sum \leftarrow 0$
2. **for** $i \leftarrow 1$ **to** n **do**
3.   **for** $j \leftarrow i$ **to** n **do**
4. $\boxed{sum \leftarrow sum + (i-j)^2}$ c
5. **return** sum

- Identify *primitive operations* that require $\Theta(1)$ time
- Loop complexity expressed as *sum* of complexities of each iteration
- Nested loops: start with the innermost loop and proceed outwards
- This gives *nested summations*

Techniques for Algorithm Analysis

- Goal: Use asymptotic notation to simplify run-time analysis
- Running time of an algorithm depends on the *input size* n

Test1(n)

```
sum ← 0
for i ← 1 to n do
    for j ← i to n do
        sum ← sum + (i − j)^2
return sum
```

$$\sum_{j=i}^{n} c$$

- Identify *primitive operations* that require constant, i.e. $\Theta(1)$ time
- Loop complexity expressed as *sum* of complexities of each iteration
- Nested loops: start with the innermost loop and proceed outwards
- This gives *nested summations*

Techniques for Algorithm Analysis

- Goal: Use asymptotic notation to simplify run-time analysis
- Running time of an algorithm depends on the *input size* n

Test1(n)

```
1.  sum ← 0
2.  for i ← 1 to n do
3.      for j ← i to n do
4.          sum ← sum + (i − j)^2
5.  return sum
```

$$\sum_{i=1}^{n} \sum_{j=i}^{n} c$$

- Identify *primitive operations* that require $\Theta(1)$ time
- Loop complexity expressed as *sum* of complexities of each iteration
- Nested loops: start with the innermost loop and proceed outwards
- This gives *nested summations*

Techniques for Algorithm Analysis

- Goal: Use asymptotic notation to simplify run-time analysis
- Running time of an algorithm depends on the *input size* n

Test1(n)

```
1. sum ← 0
2. for i ← 1 to n do
3.     for j ← i to n do
4.         sum ← sum + (i − j)^2
5. return sum
```

$$\sum_{i=1}^{n}\sum_{j=i}^{n} c \;\; + c$$

- Identify *primitive operations* that require $\Theta(1)$ time
- Loop complexity expressed as *sum* of complexities of each iteration
- Nested loops: start with the innermost loop and proceed outwards
- This gives *nested summations*

Techniques for Algorithm Analysis

Test1(n)

1. $sum \leftarrow 0$
2. **for** $i \leftarrow 1$ **to** n **do**
3. **for** $j \leftarrow i$ **to** n **do**
4. $sum \leftarrow sum + (i - j)^2$
5. **return** sum

- Derived complexity as $$c + \sum_{i=1}^{n} \sum_{j=i}^{n} c$$
- Some textbooks will write this as $$c_1 + \sum_{i=1}^{n} \sum_{j=i}^{n} c_2$$
- Or as $$1 + \sum_{i=1}^{n} \sum_{j=i}^{n} 1$$
- Now need to work out the sum

Sums: Review

$$S = \sum_{i=1}^{n} i = \underset{i=1}{1} + \underset{i=2}{2} + \underset{i=3}{3} \underset{\cdots}{\cdots} + \underset{i=n}{n}$$

$$n+1 \quad n+1 \quad n+1 \quad n+1$$

$$\begin{array}{rrcccccccc} & S = & 1 & + 2 & + 3 & \cdots & + n \\ + & S = & n & +(n-1) & +(n-2) & \cdots & + 1 \\ \hline \end{array}$$

$$2S = (n+1)n$$

$$S = \sum_{i=1}^{n} i = \frac{1}{2}(n+1)n$$

Sums: Review

$$S = \sum_{i=a}^{b} i = \quad a \quad + (a+1) \quad \ldots \quad + b$$

$i = 1 \quad i = 2 \quad \ldots \quad i = n$

$a + b \quad a + b \quad a + b$

$$S = a + (a+1) \quad \ldots \quad + b$$

$$+$$

$$S = b \quad + (b-1) \quad \ldots \quad + a$$

$$2S = (a+b)(b-a+1)$$

$$S = \sum_{i=a}^{b} i = \frac{1}{2}(a+b)(b-a+1)$$

Sums: Review

$$\sum_{j=i}^{n} 1 = \quad 1 \quad +1 \quad +1 \quad \ldots \quad +1 \quad = n - i + 1$$

$j = i \quad j = i + 1 \quad j = i + 2 \quad \ldots \quad j = n$

$j = i + (n - i)$

$$\sum_{j=i}^{n} (n - e^{x}) = n - e^{x} \quad + n - e^{x} \quad \ldots \quad + n - e^{x} = (n - i + 1)(n - e^{x})$$

$j = i \quad j = i + 1 \quad \ldots \quad j = n$

Techniques for Algorithm Analysis

```
Test1(n)
1.    sum ← 0
2.    for i ← 1 to n do
3.        for j ← i to n do
4.            sum ← sum + (i − j)^2
5.    return sum
```

$$c + \sum_{i=1}^{n} \sum_{j=i}^{n} c \quad = c + \sum_{i=1}^{n} c(n - i + 1)$$

$$= c \ + c\sum_{i=1}^{n} n - c\sum_{i=1}^{n} i + c\sum_{i=1}^{n} 1$$

$$= c \ + cn^2 - c\frac{(n+1)n}{2} + cn = c\frac{n^2}{2} + c\frac{n}{2} + c$$

- Complexity of algorithm Test1 is $\Theta(n^2)$

Techniques for Algorithm Analysis

- Two general strategies
  1. Use Θ-bounds *throughout the analysis* and obtain Θ-bound for the complexity of the algorithm
  2. Prove a *O*-bound and a *matching* Ω-bound *separately*
     - use upper bounds (for *O*-bounds) and lower bounds (for Ω-bound) early and frequently
     - easier because upper/lower bounds are easier to sum

Techniques for Algorithm Analysis

```
Test2(A, n)
    max ← 0
    for i ← 1 to n do
        for j ← i to n do
            sum ← 0
            for k ← i to j do
                sum ← A[k]
    return max
```

- First strategy

$$\sum_{j=i}^{n} \left(c + \sum_{k=i}^{j} c\right)$$

- Will write instead

$$\sum_{j=i}^{n} \sum_{k=i}^{j} c$$

- This omits lower order term that does not effect Θ-bound

Techniques for Algorithm Analysis

```
Test2(A, n)
    max ← 0
    for i ← 1 to n do
        for j ← i to n do
            sum ← 0
            for k ← i to j do
                sum ← A[k]
    return max
```

- First strategy

$$\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} c =$$

$$c\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 1 = c\sum_{i=1}^{n} \overbrace{\sum_{j=i}^{n} (j-i+1)}^{(1+2+\cdots+(n-i+1))}$$

$$= c\sum_{i=1}^{n} \frac{(n-i+1)(n-i+2)}{2} = \frac{c}{2}\sum_{i=1}^{n} (n^2 - (2n+3)i + i^2 + 3n + 2)$$

$$= \frac{c}{2}\left(n^3 - (2n+3)\frac{(n+1)n}{2} + \frac{(2n+1)(n+1)n}{6} + 3n^2 + 2n\right)$$

- Test2 is $\Theta(n^3)$

Techniques for Algorithm Analysis

```
Test2(A, n)
    max ← 0
    for i ← 1 to n do
        for j ← i to n do
            sum ← 0
            for k ← i to j do
                sum ← A[k]
    return max
```

- Second strategy: upper bound
- Make the number of summands in each sum equal to n
    - more iterations of both inner loops

$$c\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j}1 \le c\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}1 = c\sum_{i=1}^{n}\sum_{j=1}^{n}n$$

$$= c\sum_{i=1}^{n}n^2$$

$$= cn^3$$

- Test2 is $O(n^3)$

Techniques for Algorithm Analysis

- Second strategy: lower bound

$$c\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j}1 \geq \ ?$$

- Cannot make number of summands in each sum equal to n
- Can we make number of summands in each sum equal to $frac \cdot n$?
    - for any $0 < frac < 1$
    - sufficient for a cubic bound

Techniques for Algorithm Analysis

- Let innermost bound loop start with an and end with bn, where $0 < a < b < 1$

$$\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 1 \geq \sum \sum \sum_{k=\boldsymbol{an}}^{\boldsymbol{bn}} 1 = \sum \sum (b-a)n$$

- Inequality valid if the inner loop makes less than from $k = i$ to j summations
  - $i \leq an$
  - $j \geq bn$
  - in concrete numbers

$$\sum_{k=10}^{100} 1 \geq \sum_{k=\mathbf{20}}^{\mathbf{80}} 1$$

Techniques for Algorithm Analysis

- Let innermost bound loop start with an and end with bn, where $0 < a < b < 1$

$$\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 1 \geq \sum \sum \sum_{k=\boldsymbol{an}}^{\boldsymbol{bn}} 1 = \sum \sum (b-a)n$$

- Inequality valid if the inner loop makes less than from $k = i$ to j summations
  - $i \leq an$
  - $j \geq bn$
- Therefore

$$\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 1 \geq \sum_{i=1}^{\boldsymbol{an}} \sum_{j=\boldsymbol{bn}}^{n} \sum_{k=\boldsymbol{an}}^{\boldsymbol{bn}} 1$$

- Lets plug in $a = 1/3, b = 2/3$ (but any $0 < a < b < 1$ works)

$$\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 1 \geq \sum_{i=1}^{\boldsymbol{n/3}} \sum_{j=\boldsymbol{2n/3}}^{n} \sum_{k=\boldsymbol{n/3}}^{\boldsymbol{2n/3}} 1 = \sum_{i=1}^{\boldsymbol{n/3}} \sum_{j=\boldsymbol{2n/3}}^{n} \frac{n}{3} = \frac{n^3}{27}$$

- Test2 is $\Omega(n^3)$
- Combined with upper bound, Test2 is $\Theta(n^3)$

Worst Case Time Complexity

- Can have different running times on two instances of equal size

```
Test3(A, n)
A: array of size n
    for i ← 1 to n − 1 do
        j ← i
        while j > 0 and A[j] > A[j − 1] do
            swap A[j] and A[j − 1]
            j ← j − 1
```

- Let $T_A(I)$ be running time of an algorithm *A* on instance *I*
- **Worst-case complexity of an algorithm**: take the worst *I*
- Formal definition: the worst-case running time of algorithm *A* is a function $f : Z^+ \rightarrow R$ mapping n (the input size) to the *longest* running time for any input instance of size n

$$T_A(n) = max\{T_A(I) : Size(I) = n\}$$

Worst Case Time Complexity

- Can have different running times on two instances of equal size

```
Test3(A, n)
A: array of size n
for i ← 1 to n − 1 do
    j ← i
    while j > 0 and A[j] > A[j − 1] do
        swap A[j] and A[j − 1]
        j ← j − 1
```

$$\sum_{i=0}^{n-1} \sum_{j=1}^{i} c = c \sum_{i=0}^{n-1} i$$
$$= c(n-1)n/2$$

- **Worst-case complexity of an algorithm**: take worst instance I
- $T_{worst}(n) = c(n-1)n/2$
  - this is primitive operation count as a function of input size n
  - once we know primitive operation count, apply asymptotic analysis
    - $\Theta(n^2)$ or $O(n^2)$ or $\Omega(n^2)$ are all valid statements about the worst case time complexity
- For any instance I of size n, it holds $T_{worst}(n) \geq T(I) \in \Omega(T(I))$

Best Case Time Complexity

```
Test3(A, n)
A: array of size n
    for i ← 1 to n − 1 do
        j ← i
        while j > 0 and A[j] > A[j − 1] do
            swap A[j] and A[j − 1]
            j ← j − 1
```

$$\sum_{i=1}^{n-1} c = c(n-1)$$

- **Best-case complexity of an algorithm**: take the best instance *I*
- Formal definition: the best-case running time of an algorithm *A* is a function *f* : Z⁺ → R mapping n (the input size) to the *smallest* running time for any input instance of size n

$$T_A(n) = min\{T_A(I): Size(I) = n\}$$

- $T_{best}(n) = c(n-1)$
    - this is primitive operation count as a function of input size n
    - once we know primitive operation count, apply asymptotic analysis
        - $\Theta(n)$ or $O(n)$ or $\Omega(n)$ are all valid about best case time complexity
- For any instance I of size n, it holds $T_{best}(n) \leq T(I) \in O(T(I))$

Best Case Time Complexity

- Note that best-case complexity is a function of input size n
- Have to think of the best instance of size n
  - for Test3, best instance is sorted (non-increasing) array *A* of size n
  - best instance is not an array of size 1
- For ***hasNegative***, best instance is array *A* of size n where $A[0] < 0$
- Best-case complexity is $\Theta(1)$

```
Test3(A, n)
A: array of size n
1.    for i ← 1 to n − 1 do
2.        j ← i
3.        while j > 0 and A[j] > A[j − 1] do
4.            swap A[j] and A[j − 1]
5.            j ← j − 1
```

```
Algorithm hasNegative(A, n)
    Input: array A of n integers
    found ← false
    i ← 0
    while i < n − 1 and found == false
        if A[i] < 0 then
            found ← true
        i ← i + 1
    return found
```

Average Case Time Complexity

Average-case complexity of an algorithm: The average-case running time of an algorithm *A* is function $f : Z^+ \rightarrow R$ mapping n (input size) to the *average* running time of *A* over all instances of size n

$$T_A^{avg}(n) = \frac{1}{|\{I: Size(I) = n\}|} \sum_{I:Size(I)=n} T_A(I)$$

Average vs. Worst vs. Best Case Time Complexity

- Sometimes, best, worst, average time complexities are the same
- If there is a difference, then best time complexity could be overly pessimistic, worst time complexity could be overly pessimistic, and average time complexity is most useful
- However, average case time complexity is usually hard to compute
- Therefore, most often, use worst time complexity
    - worst time complexity is useful as it gives bound on the maximum amount of time one will have to wait for the algorithm to complete
    - default in this course
        - unless stated otherwise, whenever we mention time complexity, assume we mean worst case time complexity
- Suppose A has worst and best case complexities $\Theta(n^2)$ and $\Theta(n)$
    - can say complexity of A is $O(n^2)$, implying that A takes at most $O(n^2)$ time, but can have better time, depending on input

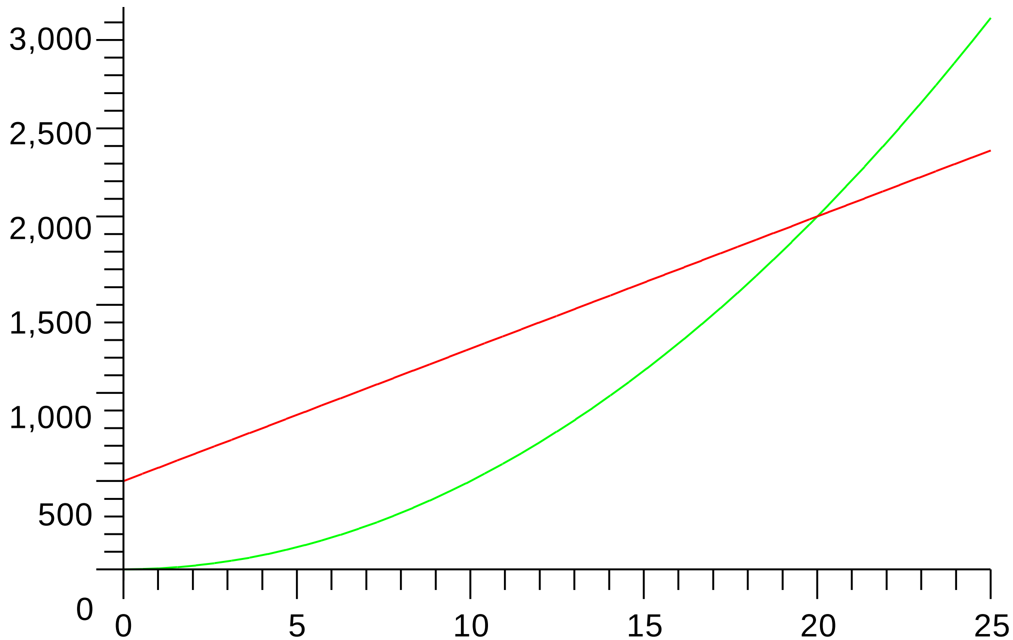
O-notation and Running Time of Algorithms

- It is important not to try make *comparisons* between algorithms using O-notation
- Suppose algorithm ***A*** and ***B*** both solve the same problem
  - ***A*** has worst-case runtime $O(n^3)$
  - ***B*** has worst-case runtime $O(n^2)$
- Cannot conclude that ***B*** is more efficient that ***A*** for all inputs
  1. the worst case runtime may only be achieved on some instances
  2. more importantly, O-notation is only an upper bound, ***A*** could have worst case runtime $O(n)$
- To compare algorithms, should use Θ notation

Running Time: Theory and Practice, Multiplicative Constants

- Algorithm ***A*** has runtime $T(n) = 10000n^2$
- Algorithm ***B*** has runtime $T(n) = 10n^2$
- Theoretical efficiency of ***A*** and ***B*** is the same, $\Theta(n^2)$
- In practice, algorithm ***B*** will run faster (for most implementations)
  - multiplicative constants matter in practice, given two algorithms with the same growth rate
  - but we will not talk about this issue more in this course

Running Time: Theory and Practice, Small Inputs

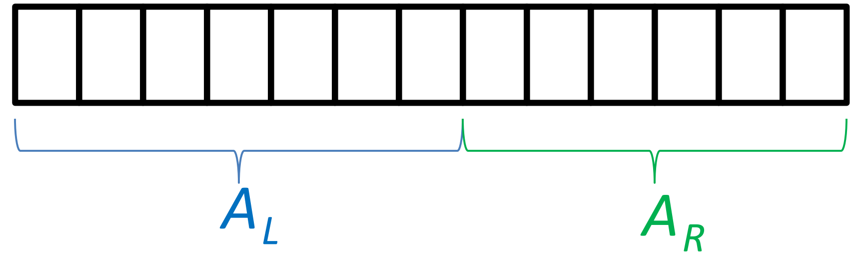


- Algorithm $\boldsymbol{A}$ running time $T(n) = 75n + 500$
- Algorithm $\boldsymbol{B}$ running time $T(n) = 5n^2$
- Then $\boldsymbol{B}$ is faster for $n \leq 20$
  - will use this fact when talking about practical implementation of recursive sorting algorithms

Outline

- CS240 overview
    - Course objectives
    - Course topics
- Introduction and Asymptotic Analysis
    - algorithm design
    - pseudocode
    - measuring efficiency
    - asymptotic analysis
    - analysis of algorithms
    - analysis of recursive algorithms
    - helpful formulas

Design of MergeSort



Input: Array A of n integers

Step 1: split A into two subarrays

- A_L consists of the first $\left\lceil \frac{n}{2} \right\rceil$ elements
- A_R consists of the last $\left\lfloor \frac{n}{2} \right\rfloor$ elements

Step 2: *Recursively* run *MergeSort* on A_L and A_R

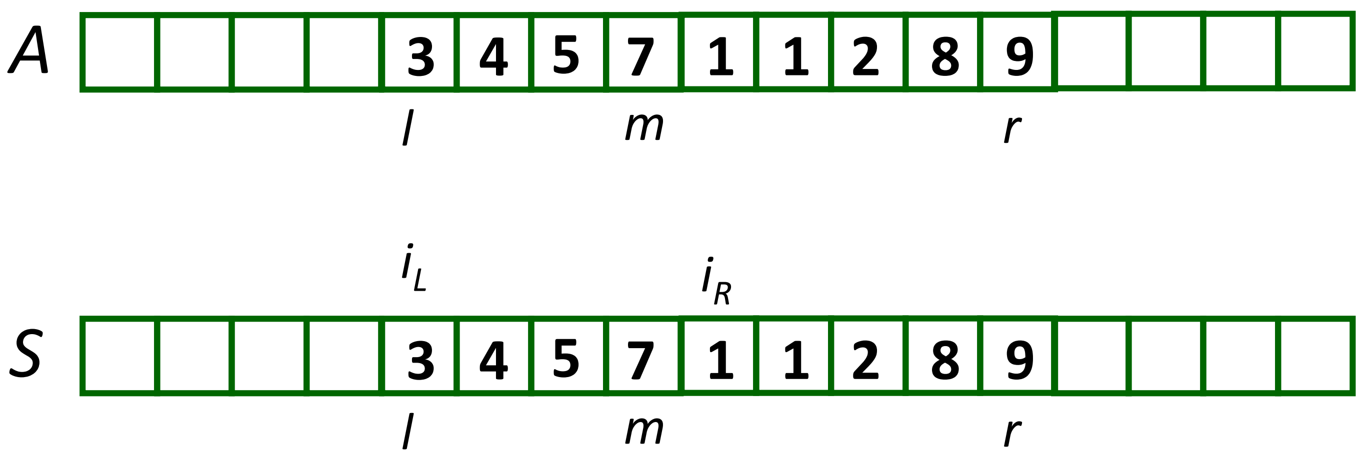
Step 3: Use function *Merge* to merge now sorted A_L and A_R into a single sorted array

MergeSort

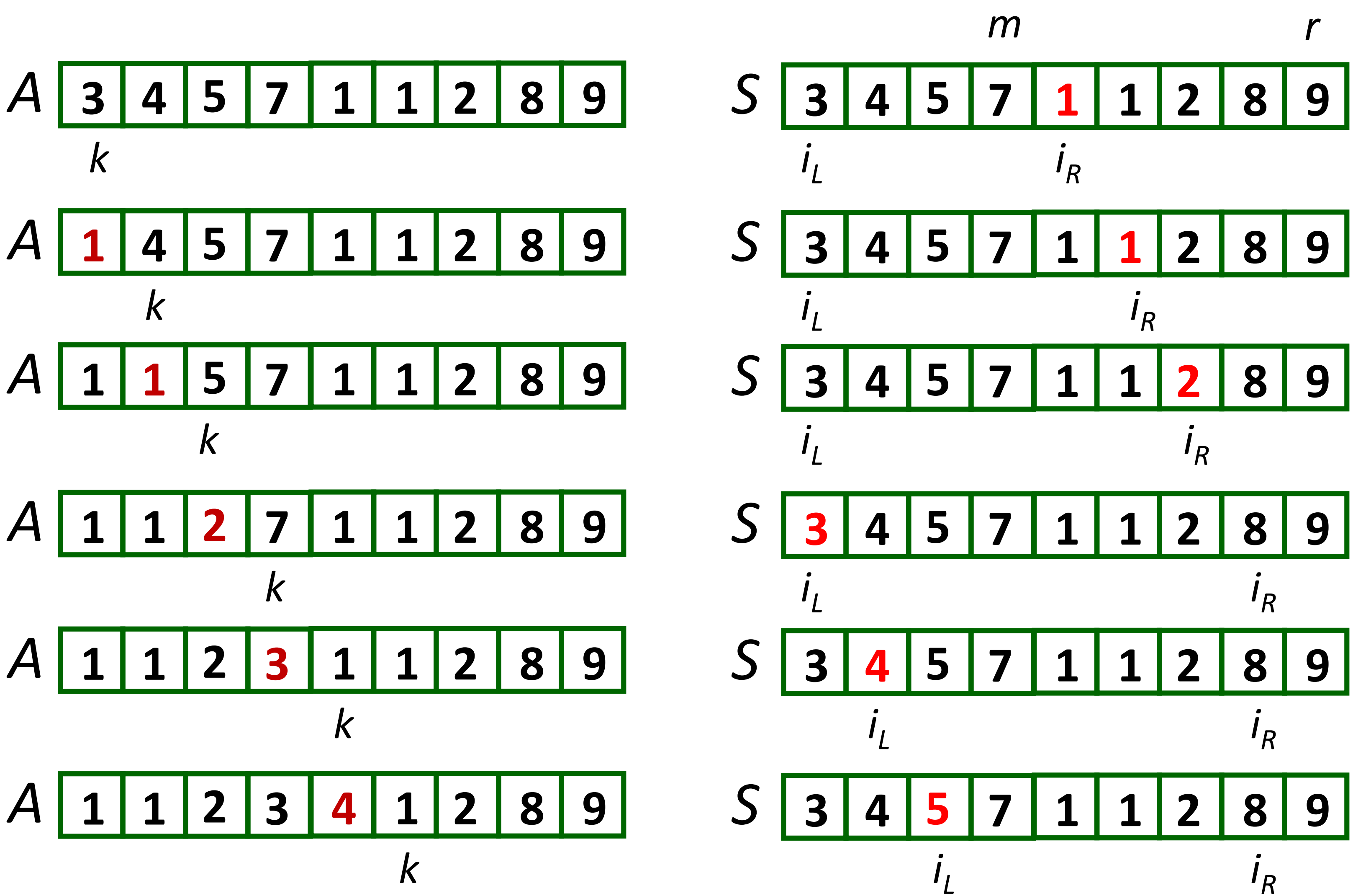
```
MergeSort(A, ℓ ← 0, r ← n − 1, S ← NIL)
A: array of size n, 0 ≤ ℓ ≤ r ≤ n − 1
1.   if S is NIL   initialize it as array S[0..n − 1]
2.   if (r ≤ ℓ) then
3.        return
4.   else
5.        m = (r + ℓ)/2
6.        MergeSort(A, ℓ, m, S)
7.        MergeSort(A, m + 1, r, S)
8.        Merge(A, ℓ, m, r, S)
```

- Two tricks to avoid copying/initializing too many arrays
  - recursion uses parameters that indicate the range of the array that needs to be sorted
  - array S used for merging is passed along as parameter

Merging Two Sorted Subarrays: Initialization



Merging Two Sorted Subarrays: Merging Starts



Merging Two Sorted Subarrays: Merging Cont.

A	1	1	2	3	**4**	1	2	8	9	
						k				

				m					r
S	3	4	**5**	7	1	1	2	8	9
			i_L					i_R	

A	1	1	2	3	4	**5**	2	8	9	
							k			

S	3	4	5	**7**	1	1	2	8	9
				i_L				i_R	

A	1	1	2	3	4	5	**7**	8	9	
								k		

S	~~3~~	~~4~~	~~5~~	~~7~~	1	1	2	8	9
					i_L			i_R	

$i_L > m$, done with the first subarray

A	1	1	2	3	4	5	7	**8**	9	
									k	

S	3	4	5	7	1	1	2	**8**	9
					i_L				i_R

A	1	1	2	3	4	5	7	8	**9**	
										k

S	3	4	5	7	1	1	2	8	**9**
					i_L				i_R

Merge

Merge(A, ℓ, m, r, S)

$A[0..n-1]$ is an array, $A[\ell..m]$ is sorted, $A[m+1..r]$ is sorted

$S[0..n-1]$ is an array

copy $A[\ell..r]$ into $S[\ell..r]$

$(i_L, i_R) \leftarrow (\ell, m+1)$;

for $(k \leftarrow \ell; k \leq r; k{+}{+})$ **do**

    if $(i_L > m)$ $A[k] \leftarrow S[i_R{+}{+}]$

    else if $(i_R > r)$ $A[k] \leftarrow S[i_L{+}{+}]$

    else if $(S[i_L] \leq S[i_R])$ $A[k] \leftarrow S[i_L{+}{+}]$

    else $A[k] \leftarrow S[i_R{+}{+}]$

- Merge takes $\Theta(\, l - r + 1\,)$ time
  - this is $\Theta(n)$ time for merging n elements

Analysis of MergeSort

- Let $T(n)$ be time to run *MergeSort* on an array of length n
  - Steps 5 takes $T\left(\left\lceil \frac{n}{2} \right\rceil\right)$
  - Steps 6 takes $T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$
  - Step 7 takes $\Theta(n)$
- The recurrence relation for *MergeSort*

$$T(n) = \begin{cases} T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c\,n & if\ \ n > 1 \\ c & if\ \ n = 1 \end{cases}$$

MergeSort$(A, \ell \leftarrow 0, r \leftarrow n-1)$
A: array of size n, $0 \leq \ell \leq r \leq n-1$

1. **if** $(r \leq \ell)$ **then**
2. return
3. **else**
4. $m = (r+\ell)/2$
5. *MergeSort*(A, ℓ, m)
6. *MergeSort*$(A, m+1, r)$
7. *Merge*(A, ℓ, m, r)

Analysis of MergeSort

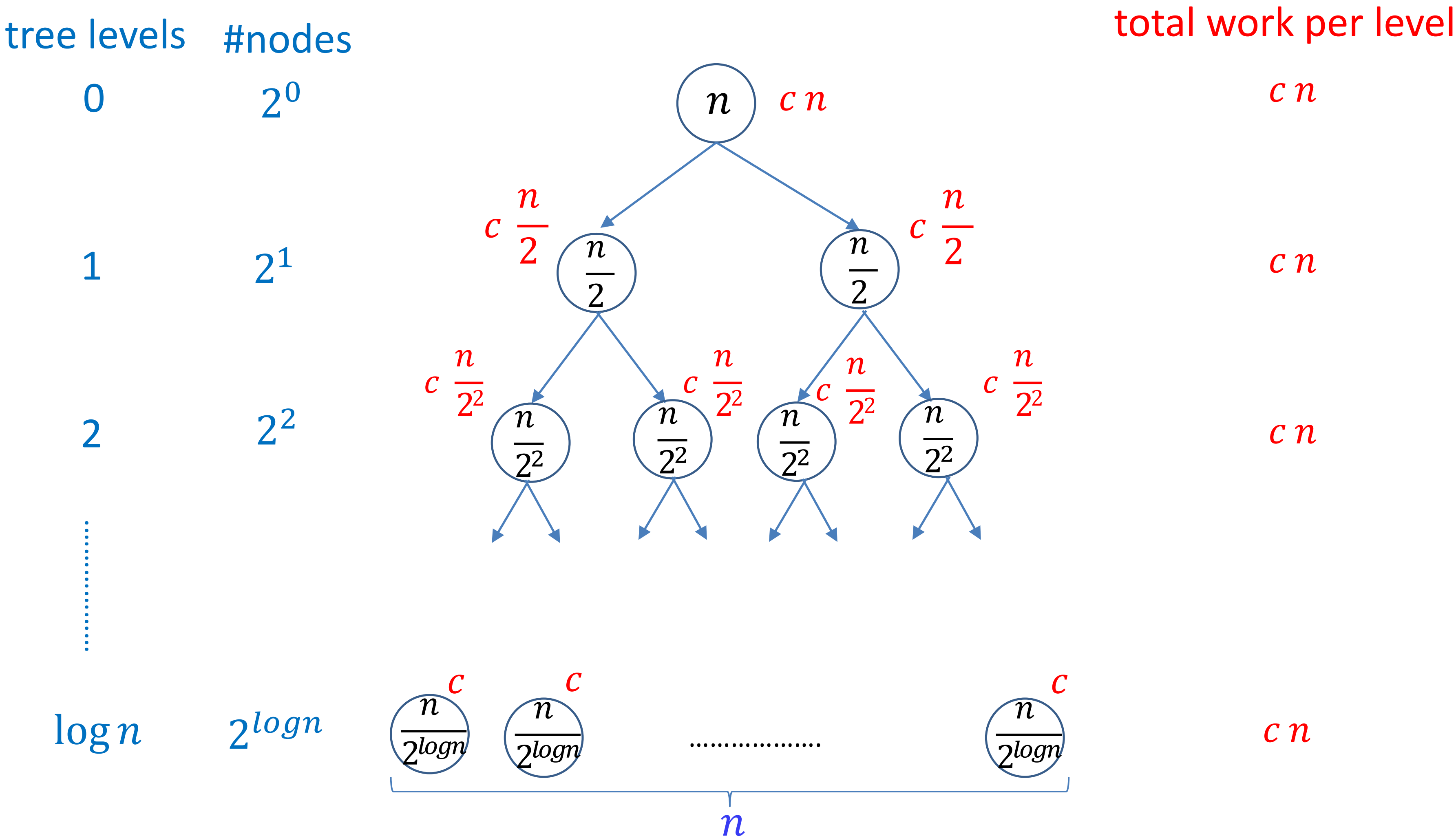
- *Sloppy recurrence* with floors and ceilings removed

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \\ c & \text{if } n = 1 \end{cases}$$

- Exact and sloppy recurrences are *identical* when n is a power of 2
- Recurrence easily solved when $n = 2^j$

Visual proof via Recursion Tree

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + c\,n & \text{if } n > 1 \\ c & \text{if } n = 1 \end{cases}$$



- cn operations on each tree level, $\log n$ levels, total work is $cn \log n \in \Theta(n \log n)$

Analysis of MergeSort

- Can show $T(n) \in \Theta(n \log n)$ for all n by analyzing exact recurrence
  - for smallest m s.t. $2^{m-1} \leq n$
    - $T(2^{m-1}) \leq T(n) \leq T(2^m)$
    - $T(2^{m-1}), T(2^m) \in \Theta(n \log n)$

Some Recurrence Relations

Recursion	resolves to	example
$T(n) = T(n/2) + \Theta(1)$	$T(n) \in \Theta(\log n)$	Binary search
$T(n) = 2T(n/2) + \Theta(n)$	$T(n) \in \Theta(n \log n)$	Mergesort
$T(n) = 2T(n/2) + \Theta(\log n)$	$T(n) \in \Theta(n)$	Heapify ($\rightarrow$ later)
$T(n) = T(cn) + \Theta(n)$ for some $0 < c < 1$	$T(n) \in \Theta(n)$	Selection ($\rightarrow$ later)
$T(n) = 2T(n/4) + \Theta(1)$	$T(n) \in \Theta(\sqrt{n})$	Range Search ($\rightarrow$ later)
$T(n) = T(\sqrt{n}) + \Theta(1)$	$T(n) \in \Theta(\log \log n)$	Interpolation Search ($\rightarrow$ later)

- Once you know the result, it is (usually) easy to prove by induction
- You can use these facts without a proof, unless asked otherwise
- Many more recursions, and some methods to solve, in cs341

Outline

- CS240 overview
    - Course objectives
    - Course topics
- Introduction and Asymptotic Analysis
    - algorithm design
    - pseudocode
    - measuring efficiency
    - asymptotic analysis
    - analysis of algorithms
    - analysis of recursive algorithms
    - helpful formulas

Order Notation Summary

- *O*-notation $f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. $|f(n)| \leq c\,|g(n)|$ for all $n \geq n_0$

- Ω-notation $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. $c\,|g(n)| \leq |f(n)|$ for all $n \geq n_0$
- Θ-notation $f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$

- *o*-notation

  $f(n) \in o(g(n))$ if **for all constants** $c > 0$, there exists a constant $n_0 \geq 0$ s.t. $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$
- ω-notation

  $f(n) \in \omega(g(n))$ if for all constants $c > 0$, there exists a constant $n_0 \geq 0$ s.t. $0 \leq c\,|g(n)| \leq |f(n)|$ for all $n \geq n_0$

Useful Sums

- Arithmetic $\sum_{i=0}^{n-1}(a+di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2)$

- Geometric $\sum_{i=0}^{n-1} ar^i = \begin{cases} a\frac{r^n-1}{r-1} \in \Theta(r^{n-1}) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a\frac{1-r^n}{1-r} \in \Theta(1) & \text{if } 0 < r < 1 \end{cases}$

- Harmonic $\sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + o(1) \in \Theta(\log n)$

- A few more $\sum_{i=1}^{n} \frac{1}{i^2} \in \Theta(1)$ $\quad \sum_{i=1}^{n} i^k \in \Theta\left(n^{k+1}\right)$ for $k \geq 0$

  $\sum_{i=0}^{\infty} ip(1-p)^{i-1} = \frac{1}{p}$ for $0 < p < 1$

- You can use these facts without a proof, unless asked otherwise

Useful Math Facts

- **Logarithms:**
  - $c = \log_b(a)$ means $b^c = a$. E.g. $n = 2^{\log n}$.
  - $\log(a)$ (in this course) means $\log_2(a)$
  - $\log(a \cdot c) = \log(a) + \log(c)$, $\log(a^c) = c\log(a)$,
  - $\log_b(a) = \frac{\log_c a}{\log_c b} = \frac{1}{\log_a(b)}$.
  - $a^{\log_b c} = c^{\log_b a}$
  - $\ln(x) = \text{natural log} = \log_e(x)$, $\frac{d}{dx}\ln x = \frac{1}{x}$
- **Factorial:**
  - $n! := n(n-1)(n-2)\cdots 2 \cdot 1 = \#$ ways to permute n elements
  - $\log(n!) = \log n + \log(n-1) + \cdots + \log 2 + \log 1 \in \Theta(n\log n)$
- **Probability and moments:**
  - $E[aX] = aE[X]$, $E[X+Y] = E[X] + E[Y]$ (linearity of expectation)