Module 1: Introduction and Asymptotic Analysis

CS 240 – Data Structures and Data Management

O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2025

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: What is this course about?

- Computer Science is mostly about problem solving
    - write program that converts given input to expected output
- 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
- Study efficient methods of *storing*, *accessing*, and *organizing* large collections of data
    - typical operations: *inserting* new data items, *deleting* data items, *searching* for specific data items, *sorting*

Course Objectives: What is this course about?

- New abstract data types (ADTs)
  - how to implement ADT efficiently using appropriate data structures
- New algorithms solving problems in data management
  - sorting, pattern matching, compression
- 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
- skip lists
- tries
- hashing
- quadtrees, kd-trees, range search
- string matching
- data compression
- external memory

Data Structures and Algorithms

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)
  - basic 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:** description of input and required output
  - for example, given an input array, rearrange elements in non-decreasing order
- **Problem Instance:** one possible input for specified problem
  - I = [5, 2, 1, 8, 2]
- **Size of a problem instance** $\text{size}(I)$
  - non-negative integer measuring size of instance I
  - $\text{size}([5, 2, 1, 8, 2]) = 5$
  - $\text{size}([]) = 0$
- Often input is array, and instance size is usually array size

Algorithm Design Terminology

- **Algorithm:** step-by-step process (can be described in finite length) for carrying out a series of computations, given an arbitrary instance I
- **Solving a problem:** algorithm $\boldsymbol{A}$ *solves* problem $\boldsymbol{\Pi}$ if for every instance I of $\boldsymbol{\Pi}$, $\boldsymbol{A}$ computes a valid output for instance I 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 and Programs

- From problem Π to program that solves it
  1. **Algorithm Design:** design algorithm(s) that solves Π
  2. **Algorithm Analysis:** assess *correctness* and *efficiency* of algorithm(s)
  3. **Implementation**: if acceptable (correct and efficient), implement algorithms(s)
     - for each algorithm, multiple implementations are possible
  4. If multiple acceptable algorithms/implementations, run experiments to determine a better solution
- CS240 focuses on the first two steps
  - the main point is to avoid implementing obviously bad 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

Pseudocode

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

```
insertion-sort(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
```

- preferred language for describing algorithms
- omits obvious details, e.g. variable declarations
- sometimes uses English descriptions (swap)
- has limited if any error detection, e.g. assumes A is initialized
- sometimes uses mathematical notation
- should use good variable names

Pseudocode Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - indentation replaces braces
- Expressions
  - $\leftarrow$ assignment
  - == equality testing
  - 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 $\leftarrow$ ***A***[0]
for $i \leftarrow 1$ **to** $n - 1$ **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

- Efficiency
  - **Running Time:** *amount of time* program takes to run
  - **Auxiliary Space:** *amount of additional memory* program requires
    - additional to the memory needed for the input instance
- Primarily concerned with time efficiency in this course
  - but also look at space efficiency sometimes
    - same techniques as for time apply to space efficiency
- When we say efficiency, assume time efficiency
  - unless we explicitly say space efficiency
- Running time is sometimes called **time complexity**
- Auxiliary space sometimes is called **space complexity**

Efficiency is a Function of Input

- The amount of time and/or memory required by a program usually depends on given instance (instance size and sometimes elements instance stores)

```
Algorithm hasNegative(A, n)
    Input: array A of n integers
    for i ← 0 to n − 1 do
        if A[i] < 0
            return True
    return False
```

$T([3,\ 4]) < T([3, 1, 4, 7, 0])$

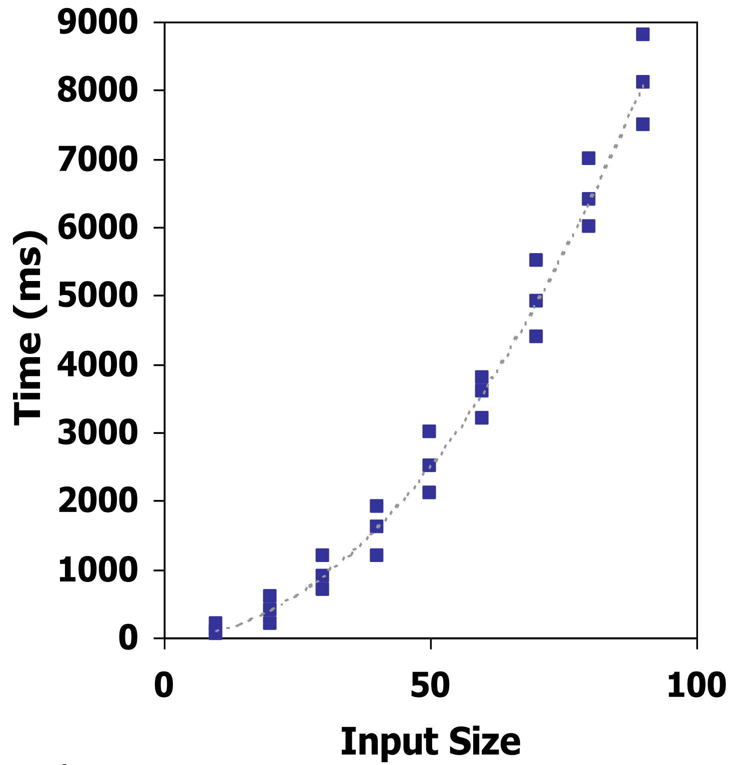
$T([3, -\mathbf{1}, 4, 7, 10]) < T([3, 1, 4])$

- So we express time or memory efficiency as a function of instances, i.e. $T(I)$
- Deriving $T(I)$ for each specific instance I is impractical
- Usually running time is longer for larger instances
- Group all instances of size n into set $I_n = \{ I \mid size(I) = n \}$
    - I_4 is all arrays of size 4
- **Measure efficiency over the set** I_n: $T(n)$ = "time for instances in I_n"
    - average over I_n?
    - smallest time instance in I_n ?
    - largest time instance in I_n ?

Running Time, Option 1: Experimental Studies

- Write program implementing the algorithm
- Run program with inputs of *varying size* and *composition*

Algorithm ***hasNegative***(***A***, ***n***)
    Input: array *A* of *n* integers
    for *i* ← 0 **to** ***n*** − 1 **do**
        if *A*[*i*] < 0
            return *True*
    return *False*

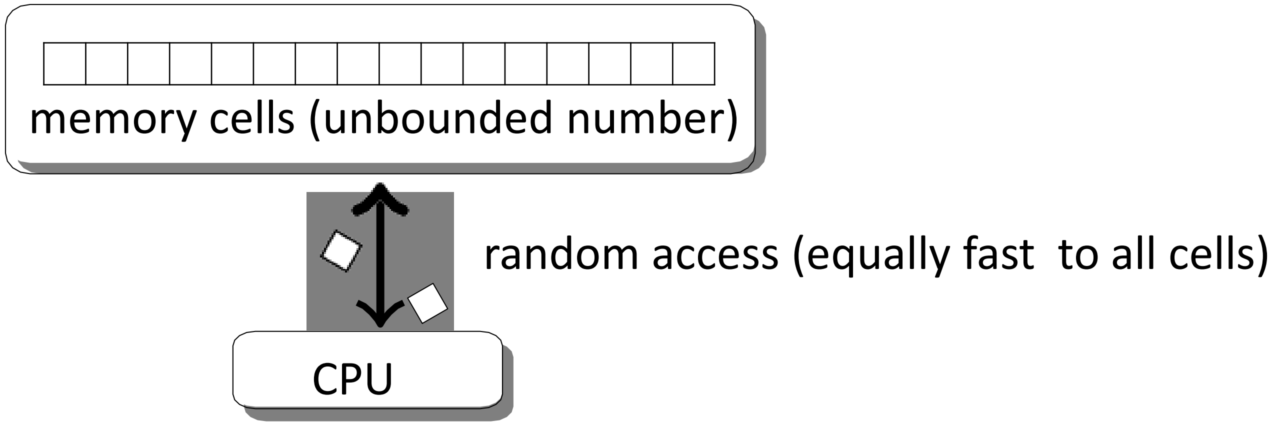


- Shortcomings
  - implementation may be complicated/costly
  - timings are affected by many factors
    - *hardware* (processor, memory)
    - *software environment* (OS, compiler, programming language)
    - *human factors* (programmer)
  - cannot test all inputs, hard to select good *sample inputs*

Running Time, Option 2: Theoretical Analysis

- Does not require implementing the algorithm
- Independent of hardware/software environment
- Takes into account all possible input instances
- [Side note: experimental studies are still useful]
  - especially when theoretical analysis yields no useful results for deciding between multiple algorithms
- For theoretical analysis, need an idealized computer model
  - “run” algorithms on idealized computer model
    - allows to understand how to compute running time and space theoretically
    - i.e. states explicitly all the assumptions we make when computing efficiency

Random Access Machine (RAM) Idealized Computer Model



- Has a set of memory cells, each of which stores one data item
    - number, character, reference
    - memory cells are big enough to hold stored items
- Any *access to a memory location* takes the same constant time
    - constant time means that time is *independent of the input size* n
- Memory access is an example of a *primitive operations*
- Can run other primitive operations on this machine (arithmetic, etc., more on this later)
    - **primitive operations take the same constant time**
- These assumptions may be invalid for a real computer

Theoretical Framework For Algorithm Analysis

- Write algorithms in pseudo-code
- Run algorithms on idealized computer model
- Time efficiency: count # *primitive operations*
  - **as a function of problem size n**
  - running time is proportional to number of primitive operations
    - since all primitive operations take the same constant time
  - can get complicated functions like $99n^3 + 8n^2 + 43421$
    - measure time efficiency in terms of *growth rate*
      - behaviour of the algorithm as the input gets larger
    - avoids complicated functions and isolates the factor that effects the efficiency the most for large inputs
- Space efficiency: count maximum # of *memory cells* ever in use
- 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
  - arithmetic: -, +, %, *, mod, round
  - assigning a value to a variable
  - indexing into an array
  - returning from a method
  - comparisons, calling subroutine, entering a loop, breaking, etc.

```
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 n

Primitive Operation Exercise

- n is the input size
- x^n is a primitive operation
  a) True
  b) False

Primitive Operation Exercise

- n is the input size
- $x^{100000000000}$ is a primitive operation
  a) True
  b) False

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 ***arraySum***(***A***, ***n***)	# operations
$sum \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	
$sum \leftarrow sum + A[i]$	
{ increment counter ***i*** }	
return *sum*	

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 ***arraySum***(***A***, ***n***) | # operations

sum ← *A*[0] | 2

for *i* ← 1 **to** *n* – 1 **do**

sum ← *sum* + *A*[*i*]

{ increment counter ***i*** }

return *sum*

i ← 1
$n-1$
$i=1$, check $i \leq n-1$ (go inside loop)
$i=2$, check $i \leq n-1$ (go inside loop)
…
$i=n-1$, check $i \leq n-1$(go inside loop)
$i=n$, check $i \leq n-1$ (do not go 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 ***arraySum*(*A*, *n*)**	# operations
sum ← *A*[0]	2
for *i* ← 1 **to** *n* – 1 **do**	2 + ***n***
sum ← *sum* + *A*[*i*]	3(***n*** – 1)
{ increment counter ***i*** }	2(***n*** – 1)
return *sum*	1

Total: 6***n***

Theoretical Analysis of Running time: Multiplicative factors

- Algorithm ***arraySum*** executes $\boldsymbol{T}(\boldsymbol{n}) = 6\boldsymbol{n}$ primitive operations
- On a real computer, primitive operations will have different runtimes
- Let a = time taken by fastest primitive operation

  b = time taken by slowest primitive operation
- Actual runtime is bounded by two linear functions

$$a\,(6\boldsymbol{n}) \le \text{actual runtime}(\boldsymbol{n}) \le b(6\boldsymbol{n})$$

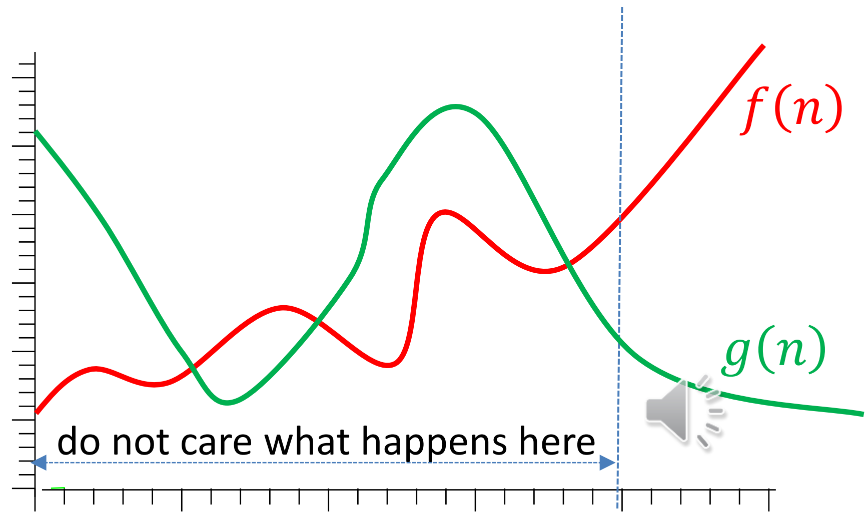
- Changing hardware/software affects runtime by a multiplicative factor
  - a and will b change, but the runtime is always bounded by $const \cdot n$
  - therefore, multiplicative constants are not essential
- Want to **ignore constant multiplicative** factors and say $\boldsymbol{T}(\boldsymbol{n}) = \boldsymbol{6n}$ is essentially $\boldsymbol{n}$
  - in a theoretically justified way

Theoretical Analysis of Running time: Lower Order Terms

- Interested in runtime for large inputs (large n)
  - datasets keep increasing in size
- Consider $\boldsymbol{T}(\boldsymbol{n}) = \boldsymbol{n}^2 + \boldsymbol{n}$
- For large $\boldsymbol{n}$, fastest growing factor contributes the most

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

- Want to ignore lower order terms in a theoretically justified way
- Perform analysis for large n (or 'eventual' behaviour)
  - this further simplifies analysis and comparing algorithms



Theoretical Analysis of Running time

- We want
  1) ignore multiplicative constant factors
  2) focus on behaviour for large n (i.e. ignore lower order terms)
- This means focusing on the *growth rate* of the function

- Want to say

$\boldsymbol{f(n)} = 10\boldsymbol{n}^2 + 100\boldsymbol{n}$ has growth rate of $\boldsymbol{g(n) = n^2}$

$\boldsymbol{f(n)} = 10\boldsymbol{n} + 10$ has growth rate of $\boldsymbol{g(n) = n}$

- Asymptotic analysis gives tools to formally focus on growth rate
- To say that function $\boldsymbol{f(n)}$ has growth rate expressed by $\boldsymbol{g(n)}$
  1) upper bound: asymptotically bound $\boldsymbol{f(n)}$ from above by $\boldsymbol{g(n)}$
  2) lower bound: asymptotically bound $\boldsymbol{f(n)}$ from below by $\boldsymbol{g(n)}$
     - asymptotically means: for large enough n, ignoring constant multiplicative factors

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

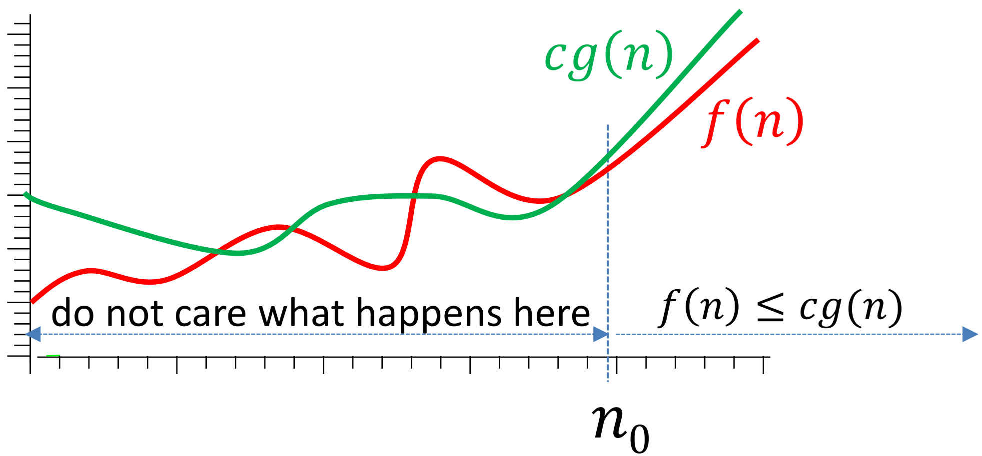
Order Notation: big-Oh

- Upper bound: asymptotically bound $\boldsymbol{f(n)}$ from above by $\boldsymbol{g(n)}$
  - $\boldsymbol{f(n)}$ is running time, is function expressing growth rate $\boldsymbol{g(n)}$

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

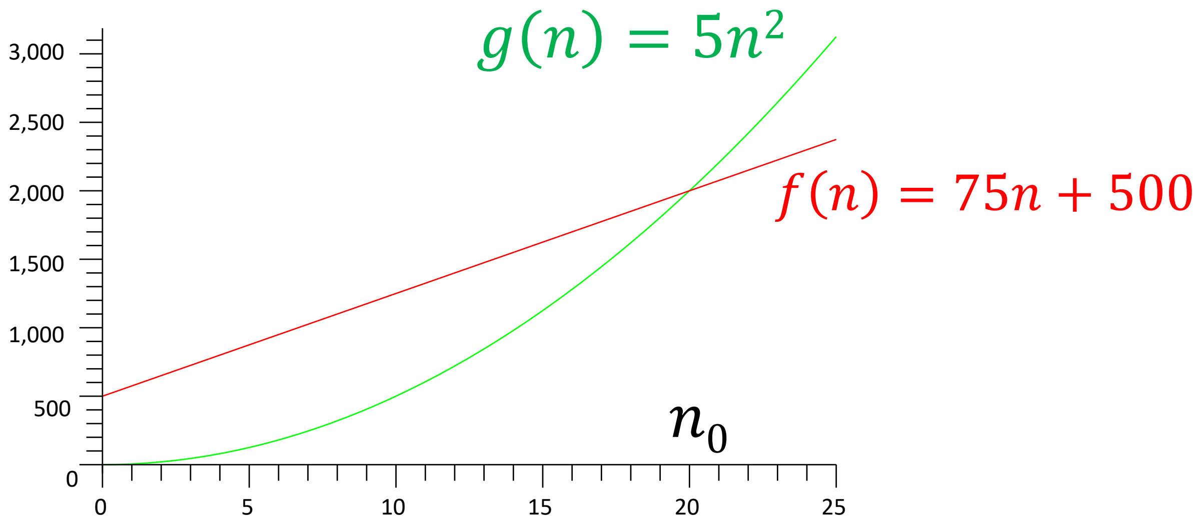


- Need c to get rid of multiplicative constant in growth rate
  - cannot say $5n^2 \leq n^2$, but can say $5n^2 \leq cn^2$ for some constant c
- Absolute value not relevant for run-time, but useful in other applications
- Unless say otherwise, assume n (and n_0) are real numbers

big-Oh Example

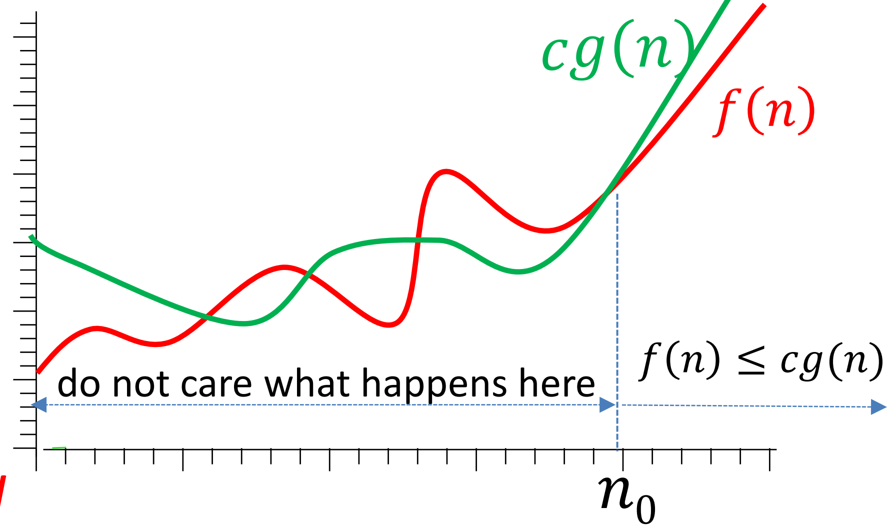
$f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t.

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



- Take $c = 1, n_0 = 20$
  - many other choices work, such as $c = 10, n_0 = 30$
- Conclusion: $f(n)$ has same or slower growth rate as $g(n)$

Order Notation: big-Oh



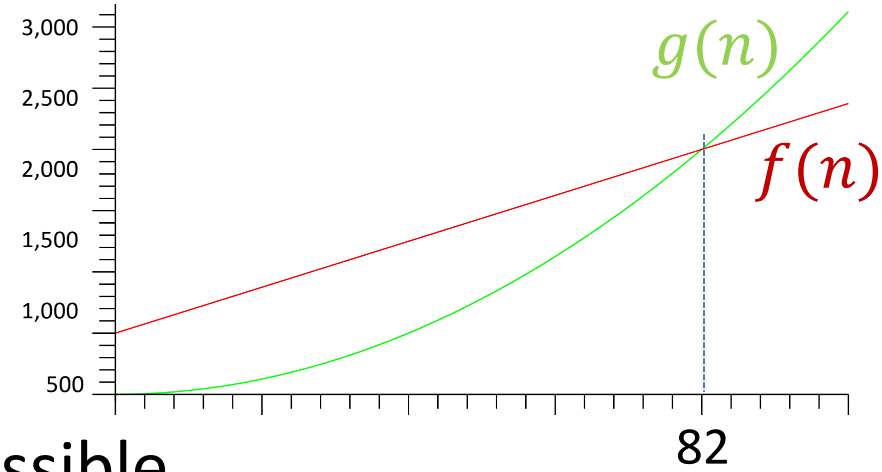
- Big-O gives asymptotic *upper bound*
  - $f(n) \in O(g(n))$ means function $f(n)$ is "bounded" above by function $g(n)$
    1. eventually, for large enough n
    2. ignoring multiplicative constant
  - Growth rate of $f(n)$ is slower or the same as growth rate of $g(n)$
- Use big-O to upper bound the 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 $f(n) \in O(g(n))$
  - $O(g(n))$ is a set of functions with the same or larger growth rate as $g(n)$

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 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 equations
- $f(n) = 75n + 500, \; g(n) = n^2$
- For all $n \geq 1$

$$75n \leq 75n \cdot n = 75n^2$$

$$500 \leq 500 \cdot n \cdot n = 500n^2$$

- Therefore, for all $n \geq 1$

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

- So take $c = 575, n_0 = 1$

Side note: for $0 < n < 1$
$75n > 75n \cdot n = 75n^2$

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 $n \geq n_0$

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

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

for all $n \geq 1$

- So 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.

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

$$2n^2 - 3n + 11 \leq 2n^2 + 0 + 11n^2 = 13n^2$$
$$\text{for all } n \geq 1$$

- Take $c = 13, n_0 = 1$

More big-O Examples

- 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.
  $2n^2 \log n + 3n \leq cn^2 \log n$ for all $n \geq n_0$

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

~~for all $n \geq 1$~~

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 ***arraySum***(***A***, ***n***) — # operations

sum ← *A*[0]	c_1
for *i* ← 1 **to** *n* – 1 **do**	
sum ← *sum* + *A*[*i*]	$c_2 n$
{ increment counter ***i*** }	
return *sum*	c_3

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

Theoretical Analysis of Running time

- Distinguishing between c_1 c_2 c_3 has no influence on asymptotic running time
  - can just use on constant c throughout

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

sum ← *A*[0] → c

for *i* ← 1 **to** *n* − 1 **do**

  sum ← *sum* + *A*[*i*]

{ increment counter ***i*** } → cn

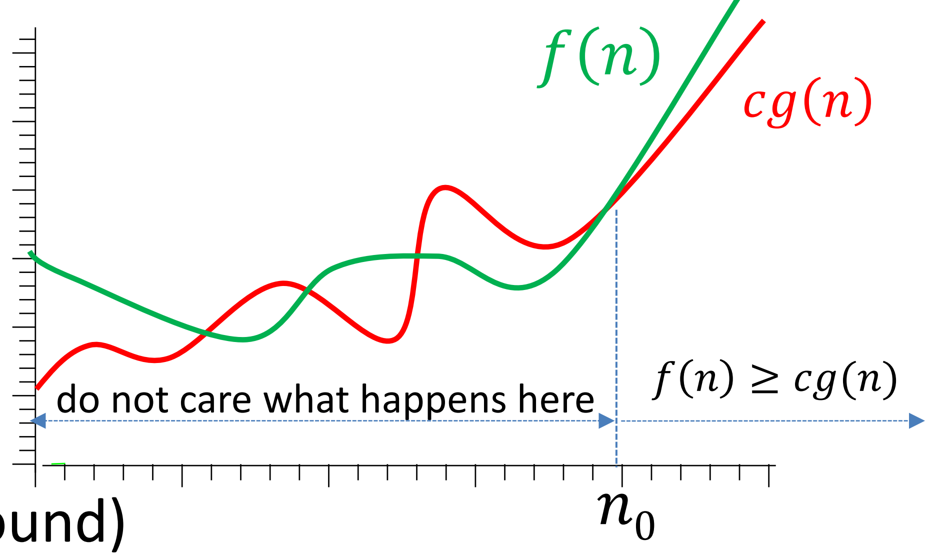
return *sum* → c

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
    - if I say I have less than a million $ in my pocket, it is a true, but useless statement
    - i.e. this statement does not give a tight upper bound
    - upper bound is *tight* if it uses the slowest growing function possible
- Want an asymptotic notation that guarantees a *tight* upper bound
- For tight bound, also need asymptotic *lower bound*

Aymptotic Lower Bound



- Ω-notation (asymptotic lower bound)

  $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$

  s.t. $|f(n)| \geq c|g(n)|$ 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 $g(n)$
- $f(n) \in O\big(g(n)\big), f(n) \in \Omega(g(n)) \Rightarrow f(n)$ has same growth as $g(n)$

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.

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

$$2n^2 + 3n + 11 \geq 2n^2 \quad \text{for all } n \geq 0$$

- Take $c = 2, n_0 = 0$

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 $\frac{1}{2}n^2 - 5n \in \Omega(n^2)$
  - to handle absolute value correctly, need to insure $f(n) \geq 0$ for $n \geq n_0$
- 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 just drop lower growing term, as $\frac{1}{2}n^2 - 5n \leq \frac{1}{2}n^2$

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

- Take $c = \frac{1}{4}, n_0 = 20$
  - $f(n) \geq \frac{1}{4}n^2$ for $n \geq 20 \Rightarrow f(n) \geq 0$ for $n \geq 20$
    - as needed to handle absolute value correctly

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.

$$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)| \text{ for all } n \geq n_0$$

- $f(n) \in \Theta\big(g(n)\big)$ means $f(n)$, $g(n)$ have equal growth rates
  - typically $f(n)$ is complicated, and $g(n)$ is chosen to be simple
- 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. $f(n) \in \Omega(g(n))$

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 $n \geq n_0$
- $\log_b n = \frac{\log n}{\log b} = \frac{1}{\log b} \log n$
- $\frac{1}{\log b} \log n \leq \log_b n \leq \frac{1}{\log b} \log n$
- Since $b > 1, \log b > 0$
- Take $c_1 = c_2 = \frac{1}{\log b}$ and $n_0 = 1$
  - rarely $c_1 = c_2$, normally $c_1 < c_2$

Common Growth Rates

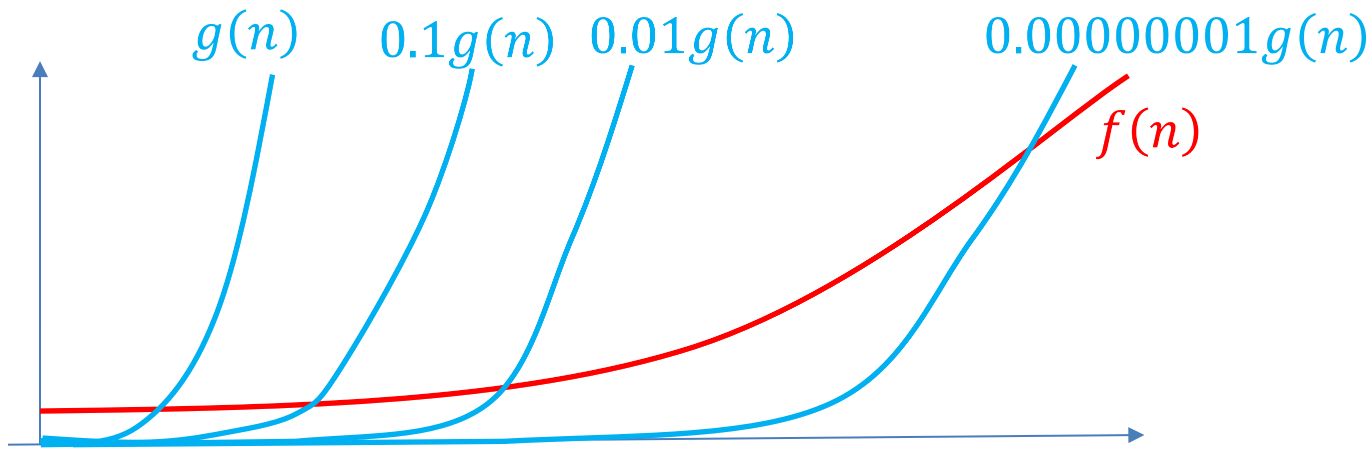
- $\Theta(1)$ *constant*
  - 1 stands for function $f(n) = 1$
- $\Theta(\log n)$ *logarithmic*
- $\Theta(n)$ *linear*
- $\Theta(n \log n)$ *linearithmic*
- $\Theta(n \log^k n)$ *quasi-linear*
  - k is constant, i.e. independent of the problem size
- $\Theta(n^2)$ *quadratic*
- $\Theta(n^3)$ *cubic*
- $\Theta(2^n)$ *exponential*
- These are listed in increasing order of growth
  - how to determine which function has a larger order of growth?

How Growth Rates Affect Running Time

- How running time affected when problem size doubles ($n \rightarrow 2n$)
  - $T(n) = c$ $\quad$ $T(2n) = c$
  - $T(n) = c\log n$ $\quad$ $T(2n) = T(n) + c$
  - $T(n) = cn$ $\quad$ $T(2n) = 2T(n)$
  - $T(n) = cn\log n$ $\quad$ $T(2n) = 2T(n) + 2cn$
  - $T(n) = cn^2$ $\quad$ $T(2n) = 4T(n)$
  - $T(n) = cn^3$ $\quad$ $T(2n) = 8T(n)$
  - $T(n) = c2^n$ $\quad$ $T(2n) = \frac{1}{c}T^2(n)$

Strictly Smaller Asymptotic Bound

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



- *o*-notation [asymptotically strictly smaller]

  $f(n) \in o(g(n))$ if **for any constant** $c > 0$, there exists a constant $n_0 \geq 0$ s.t. $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$
- Think of c as being arbitrarily small
- No matter how small c is, $c \cdot g(n)$ is eventually larger than $f(n)$
- Meaning: f grows slower than g, or growth rate of f is less than growth rage of g
- Useful for certain statements
  - there is no general-purpose sorting algorithm with run-time $o(n \log n)$

Big-Oh vs. Little-o

- Big-Oh, means f grows at the same rate or slower than g

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

- Little-o, means f grows slower than g

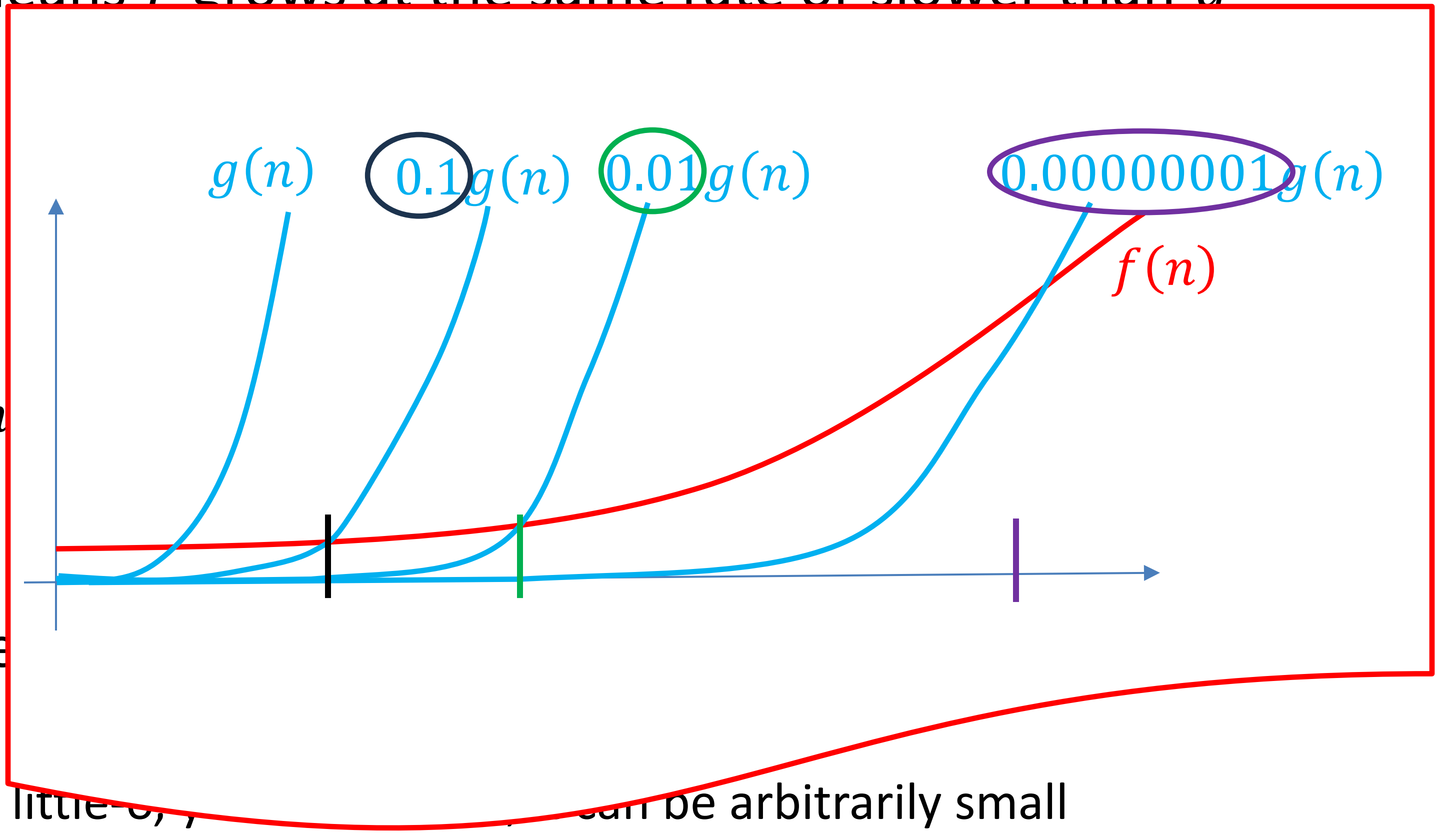
  $f(n) \in o(g(n))$ if for **any** constant $c > 0$, there exists a constant $n_0 \geq 0$ s.t. $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$

- Main difference is the quantifier for c: **exists** vs. **any**
  - for big-Oh, you can choose any c you want
  - for little-o, you are given c, it can be arbitrarily small
  - in proofs for little-o, n_0 will normally depend on c, so it is really a function $n_0(c)$
    - $n_0(c)$ **must** be a constant with respect to n

Big-Oh vs. Little-o

- Big-Oh, means f grows at the same rate or slower than g

  $f(n) \in$



- Little-o, m

  $f(n$

  con

- Main diffe
  - for
  - for little-o, … can be arbitrarily small
  - in proofs for little-o, n_0 will normally depend on c, so it is really a function $n_0(c)$
    - $n_0(c)$ **must** be a constant with respect to n

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. [solve for n in terms of c]

$$5n \leq cn^2 \text{ for all } n \geq n_0$$

divide both sides by n $\Longleftrightarrow$

$$5 \leq cn \text{ for all } n \geq n_0$$

solve for n $\Longleftrightarrow$

$$n \geq \frac{5}{c}$$

- Therefore, $5n \leq cn^2$ for $n \geq \frac{5}{c}$
- Take $n_0 = \frac{5}{c}$
  - n_0 is a function of c
  - if you ever in your proof get something like $n_0 = \frac{5n}{c}$, **it does not work,** n_0 cannot depend on n

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 $5n + 10n^4 \in o(n^5)$

- Given $c > 0$ need to find n_0 s.t.

$$5n + 10n^4 \leq cn^5 \text{ for all } n \geq n_0$$

[difficult to solve for n in terms of c]

- First derive simple upper bound

$$5n + 10n^4 \leq 15n^4 \quad \text{for all } n \geq 1$$

- Solve for n in terms of c for the simple upper bound

$$15n^4 \leq cn^5 \text{ for all } n \geq n_0$$

$$n \geq 15/c$$

- Combine: $5n + 10n^4 \leq 15n^4 \leq cn^5$ for all $n \geq 1$ for all $n \geq \frac{15}{c}$
- Take $n_0 = \max\{15/c, 1\}$

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 $n \geq n_0$

  - think of c as being arbitrarily large

- Meaning: f grows much faster than g

Strictly Larger Asymptotic Bound

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

- Claim: $f(n) \in \omega\big(g(n)\big) \Rightarrow g(n) \in o\big(f(n)\big)$

Proof:

- Given $c > 0$ need to find n_0 s.t.

$$g(n) \leq cf(n) \text{ for all } n \geq n_0$$

divide both sides by c $\Longleftrightarrow$

$$\frac{1}{c}g(n) \leq f(n) \quad \text{for all } n \geq n_0$$

- Since $f(n) \in \omega\big(g(n)\big)$, for any constant, in particular for constant $\frac{1}{c}$ there is m_0 s.t.

$$f(n) \geq \frac{1}{c}g(n) \quad \text{for all } n \geq m_0$$

- $n_0 = m_0$ and we are done!

Limit Theorem for Order Notation

- So far had proofs for order notation from the *first principles*
    - i.e. from the definition

Limit theorem for order notation

- Suppose for all $n \geq n_{0,}\ f(n) > 0,\ g(n) > 0$ and $\mathrm{L} = \lim\limits_{n \to \infty} \frac{f(n)}{g(n)}$

- Then $f(n) \in \begin{cases} o\big(g(n)\big) & if\ L = 0 \\ \Theta\big(g(n)\big) & if\ 0 < L < \infty \\ \omega\big(g(n)\big) & if\ L = \infty \end{cases}$

- Limit can often be computed using l'Hopital's rule
- Theorem gives sufficient but not necessary conditions
- Can use theorem ***unless*** asked to prove from the first principles

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 $f(n) \in \Theta(n^d)$

Proof:

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

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

$$= c_d > 0$$

Example 2

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

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

L'Hopital rule

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

Example 3

- Prove $(\log n)^a \in o(n^d)$, for any (big) $a > 0$, (small) $d > 0$
    - $(\log n)^{1000000} \in o(n^{0.0000001})$

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 $k - 1$
- $$\lim_{n\to\infty} \frac{\ln^k n}{n} = \lim_{n\to\infty} \frac{\frac{1}{n} k\, ln^{k-1} n}{1} = k \lim_{n\to\infty} \frac{ln^{k-1} n}{n} = 0$$

L'Hopital rule

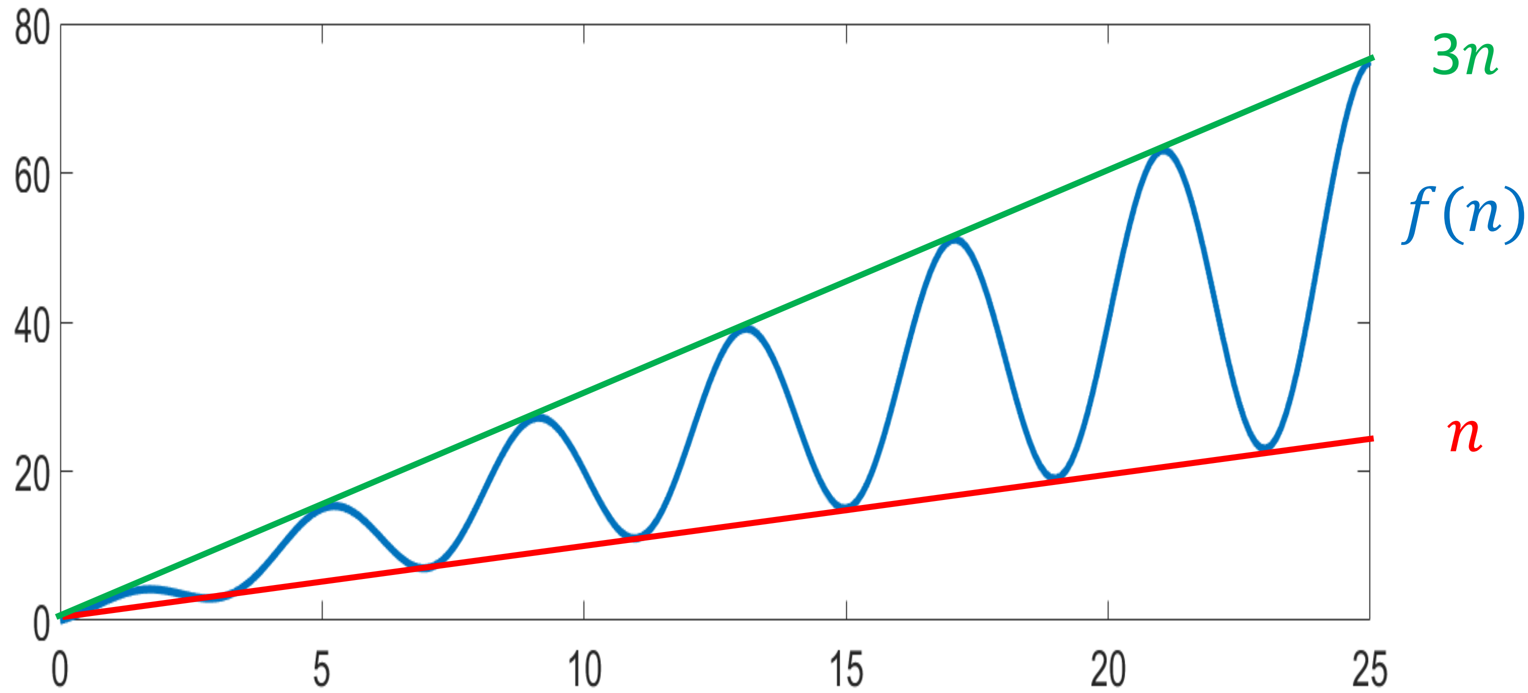
2) Prove $\lim_{n\to\infty} \frac{\ln^a n}{n^d} = 0$

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

3) Finally $$\lim_{n\to\infty} \frac{(\log n)^a}{n^d} = \lim_{n\to\infty} \frac{\left(\frac{\ln n}{ln2}\right)^a}{n^d} = \left(\frac{1}{ln2}\right)^a \lim_{n\to\infty} \frac{(\ln n)^a}{n^d} = 0$$

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

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

$$f(n) \leq n(2+1) = 3n \quad \text{for all } n \geq 0$$

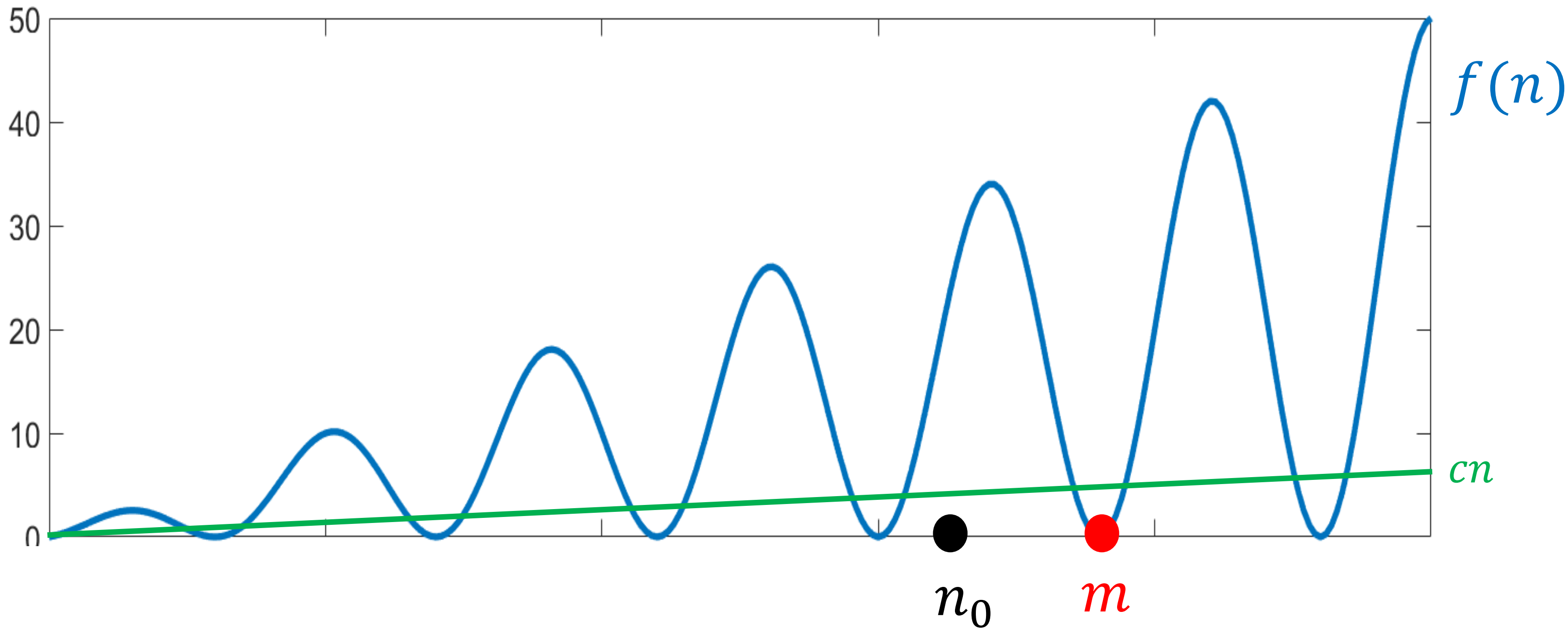
$$n = n(2-1) \leq f(n) \quad \text{for all } n \geq 0$$

- Use $c_1 = 1, c_2 = 3, n_0 = 0$

Example 5

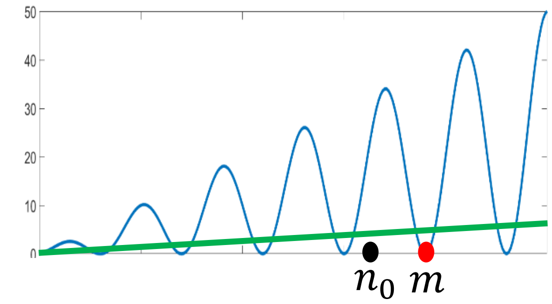
$f(n) \in \Omega(g(n))$ if $\exists$ constants $c > 0,\ n_0 \geq 0$ s.t. $|f(n)| \geq c|g(n)|$ for $n \geq n_0$

- Let $f(n) = n(1 + \sin n\pi/2)$, prove that $f(n)$ is not $\Omega(n)$



- Many points do not satisfy $f(n) \geq cn$ for $n \geq n_0$, but easiest to use zero-valued one for the formal proof

Example 5



- Let $f(n) = n(1 + \sin n\pi/2)$
- Prove that $f(n)$ is not $\Omega(n)$
- Proof: (by contradiction)
  - suppose $f(n)$ is $\Omega(n)$
  - then $\exists$ $n_0 \geq 0$ and $c > 0$ s.t. $f(n) \geq cn$ for $n \geq n_0$
  - [for contradiction, will find $m \geq n_0$ s.t. $0 = f(m)$]

$$n(1 + \sin n\pi/2) \geq cn \quad \text{for all } n \geq n_0$$
$$\Leftrightarrow$$
$$(1 + \underbrace{\sin n\pi/2}) \geq c \quad \text{for all } n \geq n_0$$

need to make this -1 for contradiction for some $m \geq n_0$

- need $\frac{m\pi}{2} = \frac{3\pi}{2} + 2\pi i$ for some integer i and $m \geq n_0$ $\Leftrightarrow$
- need $m = 3 + 4i$ for some integer i and $m \geq n_0$
- take $m = 3 + 4\lceil n_0 \rceil > n_0$

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
- $f(n) \in \omega(g(n))$: growth rate of f is greater than growth rate of g
- $f(n) \in O(g(n))$: 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

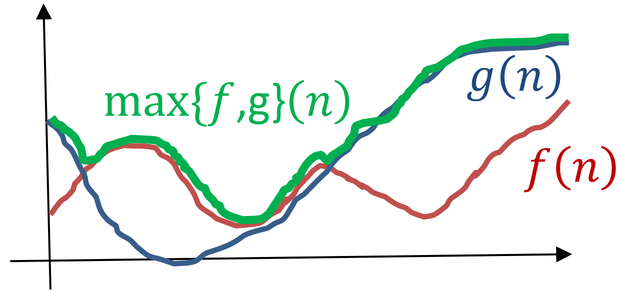
- $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$
- $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
- $f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \notin \Omega(g(n))$
- $f(n) \in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

Algebra of Order Notations (1)

- The following rules are easy to prove [exercise]

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 $f(n) \in \Omega\big(h(n)\big)$
   - if $f(n) \in O\big(g(n)\big)$ and $g(n) \in o\big(h(n)\big)$ then $f(n) \in o\big(h(n)\big)$

Algebra of Order Notations (2)



$$max\{f,g\}(n) = \begin{cases} f(n) & \text{if } f(n) > g(n) \\ g(n) & \text{otherwise} \end{cases}$$

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:

function positivity

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

b) $f(n) + g(n) = max\{f(n), g(n)\} + min\{f(n), g(n)\}$

$\leq max\{f(n), g(n)\} + max\{f(n), g(n)\}$

$= 2max\{f(n), g(n)\}$

- Usage: $n^2 + \log n \in \Theta(n^2)$

Abuse of Order Notation

- Normally, say $f(n) \in \Theta\big(g(n)\big)$ because $\Theta\big(g(n)\big)$ is a set
- Sometimes it is convenient to abuse notation
  - $f(n) = 200n^2 + \Theta(n)$
    - $f(n)$ is $200n^2$ plus a term with linear growth rate
    - nicer to read than $200n^2 + 30n + \log n$
    - does not hide the constant term 200, unlike if we said $O(n^2)$
  - $f(n) = n^2 + o(1)$
    - $f(n)$ is n^2 plus a vanishing term (term goes to 0)
      - example: $f(n) = n^2 + 1/n$
- Use these sparingly, typically only for stating final result
- But **avoid** arithmetic with asymptotic notation, can go very wrong
- Instead, replace $\Theta(g(n))$ by $c \cdot g(n)$
  - still sloppy, but less dangerous
  - if $f(n) \in \Theta\big(g(n)\big)$, more accurate statement is $c \cdot g(n) \le f(n) \le c' \cdot g(n)$ for *large enough* n

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 Runtime 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
```

- Identify *primitive operations*: these require constant 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 Runtime 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        c
    return sum
```

- Identify *primitive operations*: these require constant 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
```

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

- Identify *primitive operations*: these require constant 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*: these require constant 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
```

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

- Identify *primitive operations*: these require constant 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 ← 0
2.  for i ← 1 to n do
3.      for j ← i to n do
4.          sum ← 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 even as $$1 + \sum_{i=1}^{n} \sum_{j=i}^{n} 1$$
- Now need to work out the sum

Sums: Review

$j = 1$ $\quad j = 2$ $\quad j = 3$ $\quad \dots$ $\quad j = n$

summand

$$\sum_{j=1}^{n} 1 = 1 + 1 + 1 \dots + 1 = n$$

index of summation

Sums: Review

terms from 1 to $i-1$
are missing

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

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

Sums: Review

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

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

Sums: Review

$n+1 \qquad n+1 \qquad n+1 \qquad n+1$

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

$$+ \quad S = n \quad +(n-1) \quad +(n-2) \quad \cdots \quad +1$$

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

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

Sums: Review

$$\begin{array}{rcccc} & a+b & a+b & & a+b \\ S = \sum_{i=a}^{b} i = & a & +(a+1) & \cdots & +b \\ + \quad S = & b & +(b-1) & \cdots & +a \end{array}$$

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

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

Techniques for Algorithm Analysis

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

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

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

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

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

Techniques for Algorithm Analysis

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

- Can use Θ-bounds earlier, before working out the sum

$$c + \sum_{i=1}^{n} \sum_{j=i}^{n} c \qquad \text{is} \qquad \Theta\left(\sum_{i=1}^{n} \sum_{j=i}^{n} c\right)$$

- Therefore, can drop the lower order term and work on

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

- Using Θ-bounds earlier makes final expressions simpler
- 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
     - used this strategy on previous slides for Test1 Θ-bound
  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

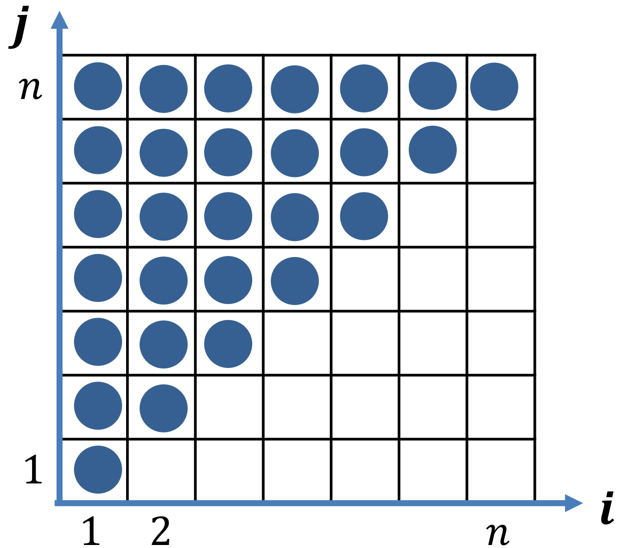
```
Test1(n)
    sum ← 0
    for i ← 1 to n do
        for j ← i to n do
            sum ← sum + (i − j)^2
    return sum
```

- Second strategy: upper bound for Test1

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

- **Add** more iterations to make sum easier to work out

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



Techniques for Algorithm Analysis

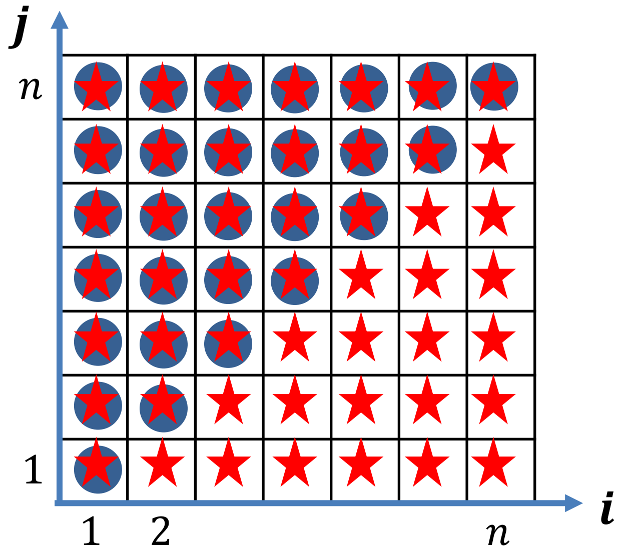
- Second strategy: upper bound for Test1

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

- **Add** more iterations to make sum easier to work out

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

upper bound ★



- Test1 is $O(n^2)$

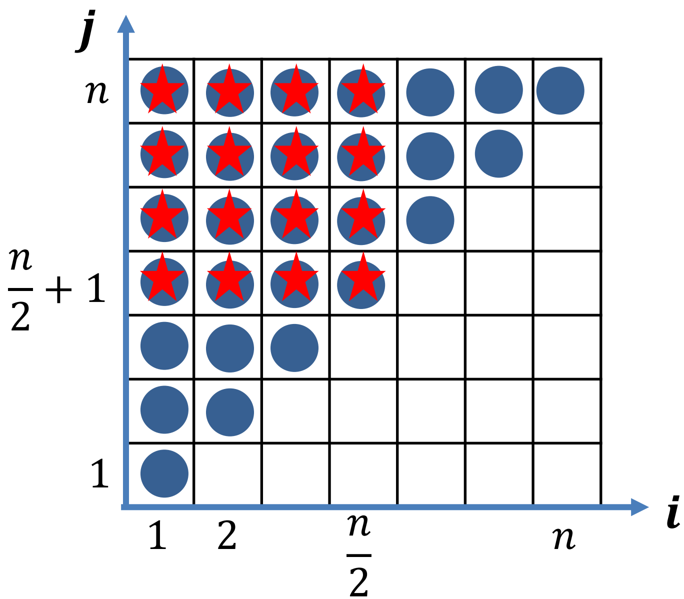
Techniques for Algorithm Analysis

- Second strategy: lower bound for Test1

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

- **Remove** iterations to make sum easier to work out

$$\sum_{i=1}^{n} \sum_{j=i}^{n} c \geq \sum_{i=1}^{n/2} \sum_{j=1+n/2}^{n} c = \sum_{i=1}^{n/2} c\frac{n}{2} = c\sum_{i=1}^{n/2} \frac{n}{2} = c\left(\frac{n}{2}\right)^2$$



- Test1 is $\Omega(n^2)$

Techniques for Algorithm Analysis

- Second strategy: lower bound for Test1

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

- **Remove** iterations to make sum easier to work out
- Can get the same result without visualization
- To remove iterations, increase lower or increase upper range bounds, or both
- Examples: $\sum_{k=10}^{100} c \geq \sum_{k=\mathbf{20}}^{\mathbf{80}} c$ $\qquad \sum_{k=i}^{j} 1 \geq \sum_{k=\boldsymbol{i}+\mathbf{1}}^{\boldsymbol{j}-\mathbf{1}} 1$
- In our case:

$$\sum_{i=1}^{n} \sum_{j=i}^{n} c \geq \sum_{i=1}^{n/2} \sum_{j=i}^{n} c \geq \sum_{i=1}^{n/2} \sum_{j=1+n/2}^{n} c = c\left(\frac{n}{2}\right)^2$$

now $i \leq n/2$

- Test1 is $\Omega(n^2)$, previously concluded that Test1 is $O(n^2)$
- Therefore Test1 is $\Theta(n^2)$

Techniques for Algorithm Analysis

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

- Annoying to carry constants around $\sum_{i=1}^{n} \sum_{j=i}^{n} c$
- Running time is proportional to the number of iterations
- Can first compute the number of iterations

$$\sum_{i=1}^{n} \sum_{j=i}^{n} 1 = \frac{n^2}{2} + \frac{n}{2} + 1$$

- And then say running time is c times the number of iterations

Techniques for Algorithm Analysis

Algorithm ***Test2***(n)

```
sum ← 0
i = n
while i ≥ 2 do
    j = 0
    while j < i^2 do
        sum ← sum + 1
        j = j + i
    i = i/2
return sum
```

$O(1)$

- Inner **while** loop
  - iteration 1: $j = 0$
  - iteration 2: $j = 1 \cdot i$
  - iteration k: $j = (k-1) \cdot i$
  - terminate when $(k-1) \cdot i \geq i^2$
    - $k \geq 1 + i$
  - inner while loop takes $(1+i)c$ time
- Outer **while** loop
  - iteration 1: $i = n$
  - iteration 2: $i = n/2^{2-1}$
  - iteration t: $i = n/2^{t-1}$
  - terminates when $\frac{n}{2^{t-1}} < 2$
    - $t > \log n$ (more precisely, last iteration is at $t = \lceil \log n \rceil - 1$)
- Total time, ignoring multiplicative c

some constant

$$\sum_{t=1}^{\log n} (1 + n/2^{t-1}) = \sum_{t=1}^{\log n} 1 + n \sum_{t=1}^{\log n} 1/2^t < \log n + n \sum_{t=1}^{\infty} 1/2^t \in O(n)$$

Worst Case Time Complexity

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

```
insertion-sort(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 $T(I)$ be running time of an algorithm on instance I
  - Let $I_n = \{I: Size(I) = n\}$
  - **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

$$T_{worst}(n) = \max_{I \epsilon I_n}\{T(I)\}$$

Worst Case Time Complexity

- **Worst-case complexity of an algorithm**: take worst instance I

```
insertion-sort(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
```

worst I is reverse sorted array

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

- $T_{worst}(n) = c(n-1)n/2$
    - this is primitive operation count as a function of input size n
    - after 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 running time of *insertion-sort*

Best Case Time Complexity

```
insertion-sort(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
```

best instance is sorted array

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

- **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 $f: Z^+ \rightarrow R$ mapping n (the input size) to the *smallest* running time for any input instance of size n

$$T_{best}(n) = \min_{I \epsilon I_n}\{T(I)\}$$

- $T_{best}(n) = c(n-1)$
  - this is primitive operation count as a function of input size n
  - after primitive operation count, apply asymptotic analysis
    - $\Theta(n)$ or $O(n)$ or $\Omega(n)$ are all valid about best case running time

Best Case Time Complexity

- Note that best-case complexity is a function of input size n
- Think of the best instance of size n
- For *insertion-sort*, best instance is sorted (non-increasing) array A of size n
- **Best instance is not an array of size 1**
- Best-case complexity is $\Theta(n)$

```
insertion-sort(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
```

- For *hasNegative*, best instance is array A of size n where $A[0] < 0$
- **Best instance is not an array of size 1**
- Best-case complexity is $\Theta(1)$

```
hasNegative(A, n)
    Input: array A of n integers
    for i ← 0 to n − 1 do
        if A[i] < 0
            return True
    return False
```

Best Case Running Time Exercise

- $T(n) = \begin{cases} c & \text{if } n = 5 \\ cn & \text{otherwise} \end{cases}$

```
Algorithm Mystery(A, n)
    Input: array A of n integers
    if n = 5
        return A[0]
    else
        for i ← 1 to n − 1 do
            print(A[i])
        return
```

- Best case running time?
  a) $\Theta(1)$
  b) $\Theta(n)$

Average Case Time Complexity

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

$$T_{avg}(n) = \frac{1}{|I_n|} \sum_{I \epsilon I_n} T(I)$$

- Will assume $|I_n|$ is finite
- If all instances are used equally often, $T_{avg}(n)$ gives a good estimate of a running time of an algorithm on average in practice

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 optimistic, 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, we 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
- Goal in CS240: for a problem, find an algorithm that solves it and whose tight bound on the worst case running time has the smallest growth rate

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***
- O-notation is only an upper bound
  - ***A*** could have worst case runtime $O(n)$
  - while for ***B*** the bound of $O(n^2)$ could be tight
- To compare algorithms, it is better to use Θ notation

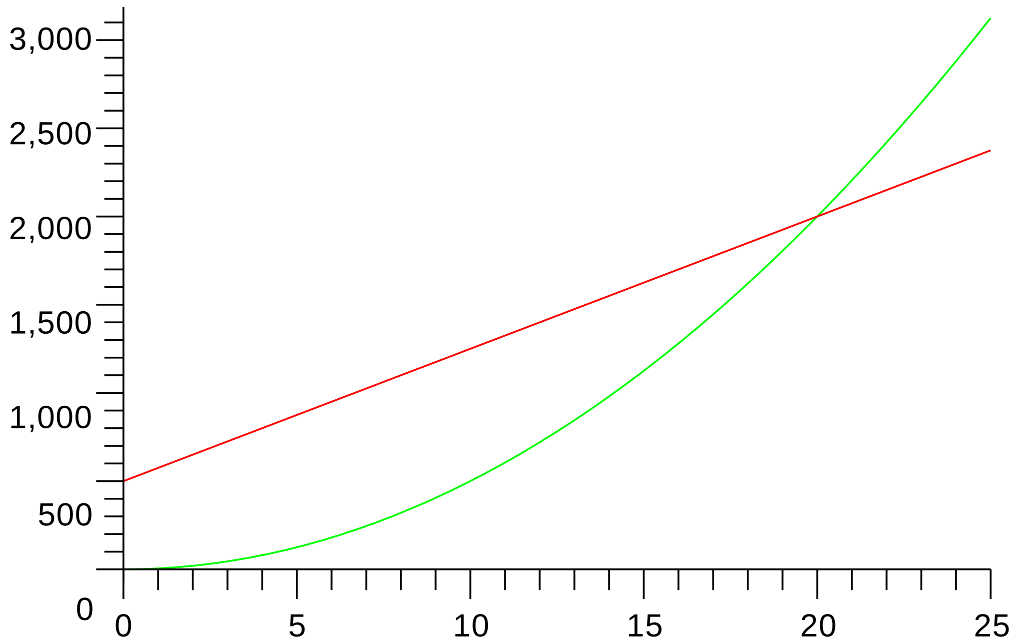
Θ-notation and Running Time of Algorithms

- Have to be careful with Θ-notation
- Suppose algorithm ***A*** and ***B*** both solve the same problem
  - ***A*** has worst-case runtime $\Theta(n^3)$
  - ***B*** has worst-case runtime $\Theta(n^2)$
- Cannot conclude that ***B*** is more efficient that ***A*** for **all inputs**
  - the worst case runtime may be achieved only on some instances

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 are concerned with theory (mostly), and multiplicative constants do not matter in asymptotic analysis

Running Time: Theory and Practice, Small Inputs



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

Theoretical Analysis of Space

- Interested in *auxiliary* space
  - space used in addition to the space used by the input data
- To find *space* used by an algorithm, count total number of auxiliary memory cells ever accessed (for reading or writing or both) by algorithm
  - as a function of input size $\boldsymbol{n}$
  - space used must always be initialized, although it may not be stated explicitly in pseudocode
- *arrayMax* uses 2 memory cells
  - $T(n) = 2$
  - space efficiency is $O(1)$

```
Algorithm arrayMax(A, n)
      currentMax ← A[0]
   for i ← 1 to n − 1 do
      if A[i] > currentMax then
         currentMax ← A[i]
   return currentMax
```

Theoretical Analysis of Space

- *arrayCumSum* uses $1 + n$ memory cells
    - $T(n) = 1 + n$
    - space efficiency is $O(n)$

Algorithm ***arrayCumSum***(***A***, ***n***)

Input: array *A* of *n* integers

initialize array *B* of size ***n*** *to 0*

B[*0*] ← *A*[*0*]

for ***i*** ← 1 **to** ***n*** – 1 **do**

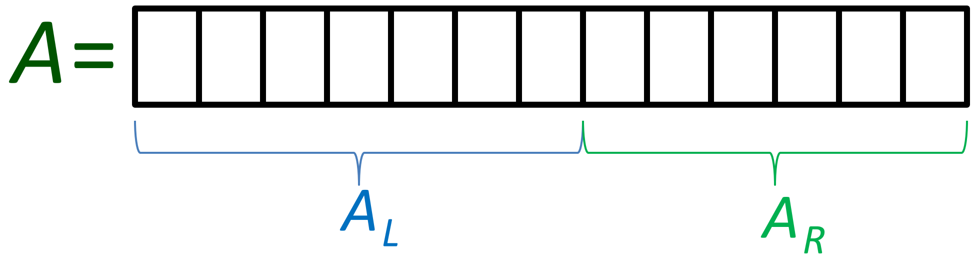
B[*i*] ← *B*[*i* - 1] + *A*[*i*]

return *B*

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

MergeSort: Overall Idea



Input: Array A of n integers

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

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

3: After A_L and A_R are sorted, use function *Merge* to merge them into a single sorted array

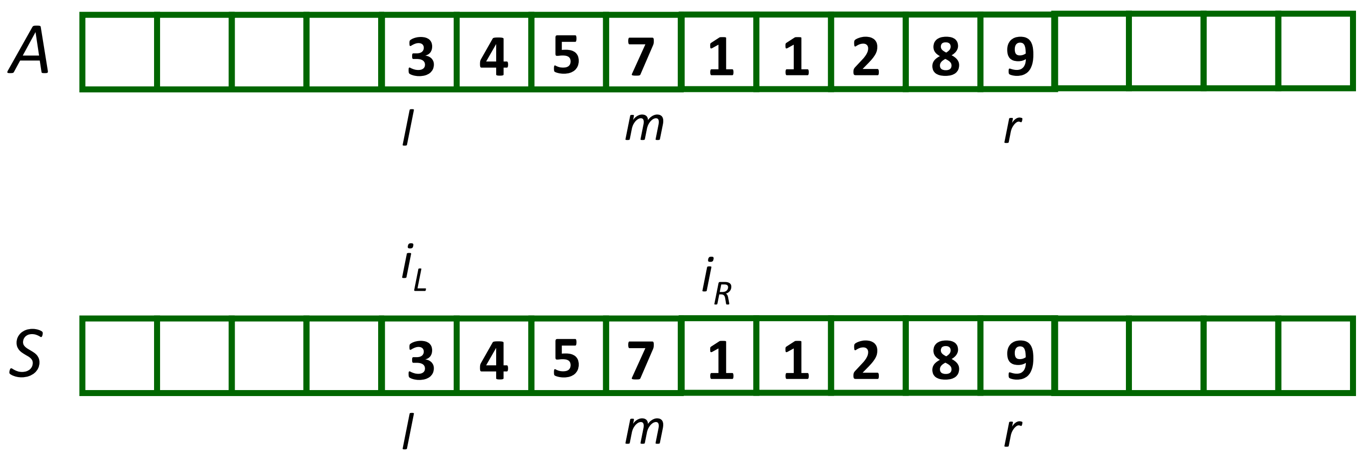
MergeSort: Pseudo-code

merge-sort$(A, n, \ell \leftarrow 0, r \leftarrow n-1, S \leftarrow \texttt{NIL})$
A: array of size n, $0 \le \ell \le r \le n-1$

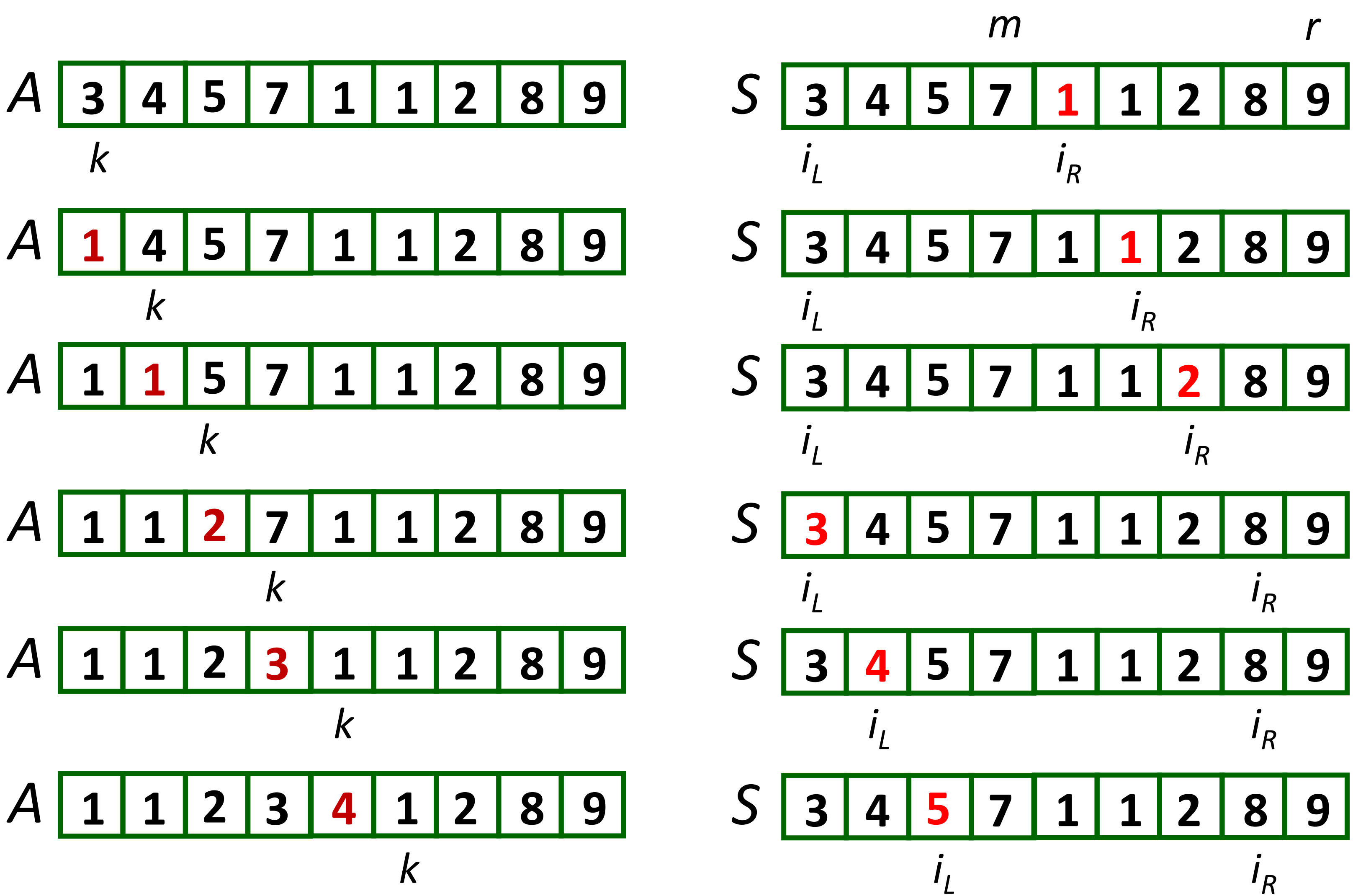
- **if** S is $\texttt{NIL}$ initialize it as array $S[0..n-1]$
- **if** $(r \le \ell)$ **then**
  - **return**
- **else**
  - $m = \lfloor (r+\ell)/2 \rfloor$
  - *merge-sort*(A, n, ℓ, m, S)
  - *merge-sort*$(A, n, m+1, r, S)$
  - *merge*(A, ℓ, m, r, S)

- 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

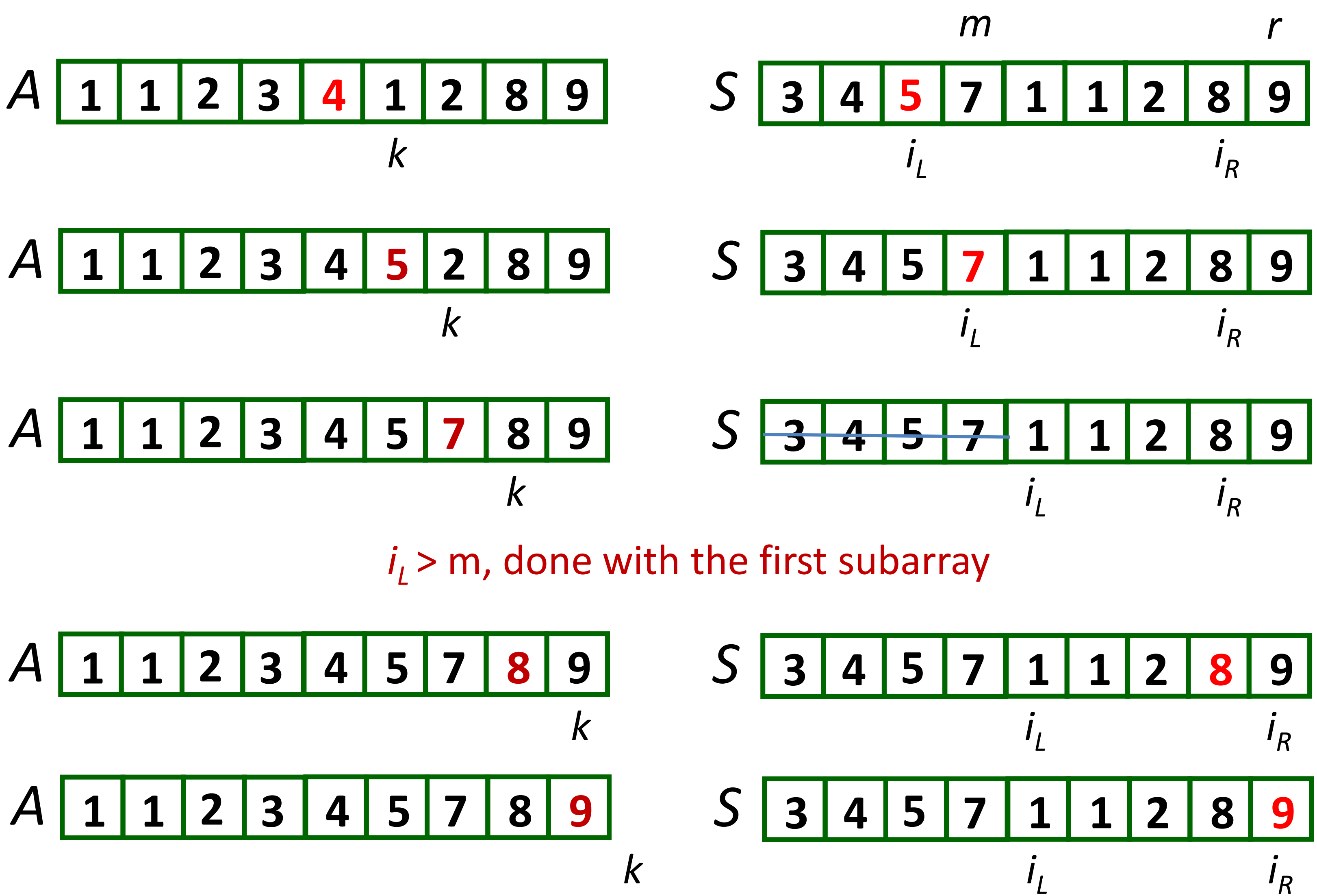
Merging Two Sorted Subarrays: Initialization



Merging Two Sorted Subarrays: Merging Starts



Merging Two Sorted Subarrays: Merging Cont.



Merge: Pseudocode

```
Merge(A, ℓ, m, r, S)
A[0..n − 1] is an array, A[ℓ..m] is sorted, A[m + 1..r] is sorted
S[0..n − 1] is an array
    copy A[ℓ..r] into S[ℓ..r]
    (i_L, i_R) ← (ℓ, m + 1);
    for (k ← ℓ; k ≤ r; k++) do
        if (i_L > m) A[k] ← S[i_R++]
        else if (i_R > r) A[k] ← S[i_L++]
        else if (S[i_L] ≤ S[i_R]) A[k] ← S[i_L++]
        else A[k] ← S[i_R++]
```

- Merge takes $\Theta(r - l + 1)$ 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

merge-sort$(A, n, l \leftarrow 0, r \leftarrow n-1, S \leftarrow NULL)$	
A: array of size n, $0 \leq l \leq r \leq n-1$	
if $r \leq l$ **then** \\ base case	c
return	
if S is $NULL$ initialize it as array $S[0 \ldots n-1]$	cn
$m = \lfloor (l+r)/2 \rfloor$	c
merge-sort(A, n, l, m, S)	$T\left(\left\lceil \frac{n}{2} \right\rceil\right)$
merge-sort$(A, n, m+1, r, S)$	$T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$
merge(A, l, m, r, S)	cn

- **Recurrence relation** for *MergeSort*

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

Analysis of MergeSort

- Recurrence relation for *MergeSort*

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

- *Sloppy recurrence* with floors and ceilings removed

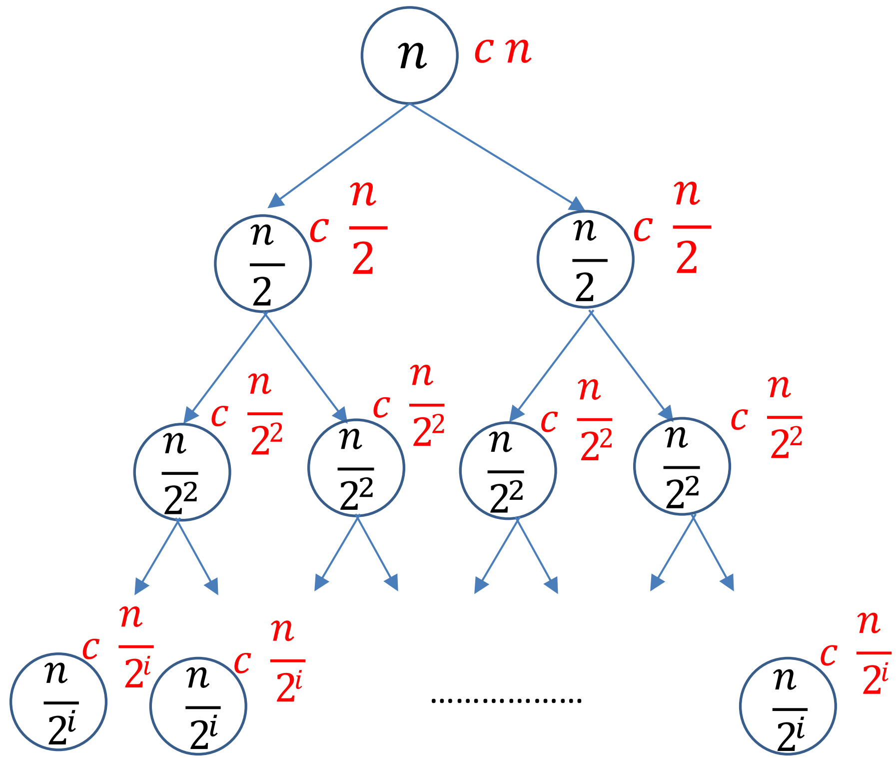
$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \\ c & \text{if } n = 1 \end{cases}$$

- 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

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

tree levels	#nodes	work per level
0	2^0	cn
1	2^1	cn
2	2^2	cn
⋮		
i	2^i	cn



- Stop recursion when node size is 1 $\Rightarrow \frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow i = \log n$
- cn operations on each tree level, $\log n$ levels, total time 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 (not sloppy) recurrence
    - sloppy recurrence is good enough for this course

Explaining Solution of a Problem

- For Merge-sort design, we had four steps
  1. describe the overall idea
  2. give pseudocode or detailed description
  3. argue correctness
     - key ingredients, no need for a formal proof
     - sometimes obvious enough from idea-description
  4. analyze runtime
- Follow these 4 steps when asked to ‘solve a problem’

Some Recurrence Relations

Recursion	resolves to	example
$T(n) \leq T(n/2) + O(1)$	$T(n) \in O(\log n)$	binary-search
$T(n) \leq 2T(n/2) + O(n)$	$T(n) \in O(n \log n)$	merge-sort
$T(n) \leq 2T(n/2) + O(\log n)$	$T(n) \in O(n)$	heapify (*)
$T(n) \leq cT(n-1) + O(1)$ for some $c < 1$	$T(n) \in O(1)$	avg-case analysis (*)
$T(n) \leq 2T(n/4) + O(1)$	$T(n) \in O(\sqrt{n})$	range-search (*)
$T(n) \leq T(\sqrt{n}) + O(\sqrt{n})$	$T(n) \in O(\sqrt{n})$	interpol. search (*)
$T(n) \leq T(\sqrt{n}) + O(1)$	$T(n) \in O(\log \log n)$	interpol. search (*)

- 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

Useful Sums

- Arithmetic

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \qquad \sum_{i=0}^{n-1}(a+di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2) \text{ if } d \neq 0$$

- Geometric

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1 \qquad \sum_{i=0}^{n-1} ar^i = \begin{cases} a\dfrac{r^n-1}{r-1} \in \Theta(r^{n-1}) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a\dfrac{1-r^n}{1-r} \in \Theta(1) & \text{if } 0 < r < 1 \end{cases}$$

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

- A few more

$$\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6} \in \Theta(1) \qquad \sum_{i=1}^{n} i^k \in \Theta\left(n^{k+1}\right) \text{ for } k \geq 0$$

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = \in \Theta(1) \qquad \sum_{i=0}^{\infty} ip(1-p)^{i-1} = \frac{1}{p} \quad \text{for } 0 < p < 1$$

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

Useful Math Facts

Logarithms:

- $y = \log_b(x)$ means $b^y = x$. e.g. $n = 2^{\log n}$.
- $\log(x)$ (in this course) means $\log_2(x)$
- $\log(x \cdot y) = \log(x) + \log(y)$, $\log(x^y) = y \log(x)$, $\log(x) \leq x$
- $\log_b(a) = \frac{\log_c a}{\log_c b} = \frac{1}{\log_a(b)}$, $a^{\log_b c} = c^{\log_b a}$
- $\ln(x) =$ natural log $= \log_e(x)$, $\frac{\mathrm{d}}{\mathrm{d}x} \ln x = \frac{1}{x}$

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:

- $E[X]$ is the expected value of X.
- $E[aX] = aE[X]$, $E[X+Y] = E[X] + E[Y]$ (linearity of expectation)