CS2210

Data Structures and Algorithms

Lecture 2:

Analysis of Algorithms

Asymptotic notation

Instructor: **Olga Veksler**

Outline

- Comparing algorithms
- Pseudocode
- Theoretical Analysis of Running time
  - primitive Operations
  - counting primitive operations
- Asymptotic analysis of running time

Comparing Algorithms

- Given 2 or more algorithms to solve the same problem, how do we select the best one?
- Some criteria for selecting an algorithm
  1) Is it easy to implement, understand, modify?
  2) How long does it take to run it to completion?
  3) How much of computer memory does it use?
- Software engineering is primarily concerned with the first criteria
- In this course we are interested in the second and third criteria

Comparing Algorithms

- Time complexity
  - The amount of time algorithm needs to run to completion
- Space complexity
  - The amount of memory algorithm needs to run
- Occasionally will look at space complexity, but mostly interested in time complexity in this course
- Thus the better algorithm is the one which runs faster (has smaller time complexity)

How to Calculate Running time

- Most algorithms transform input objects into output objects

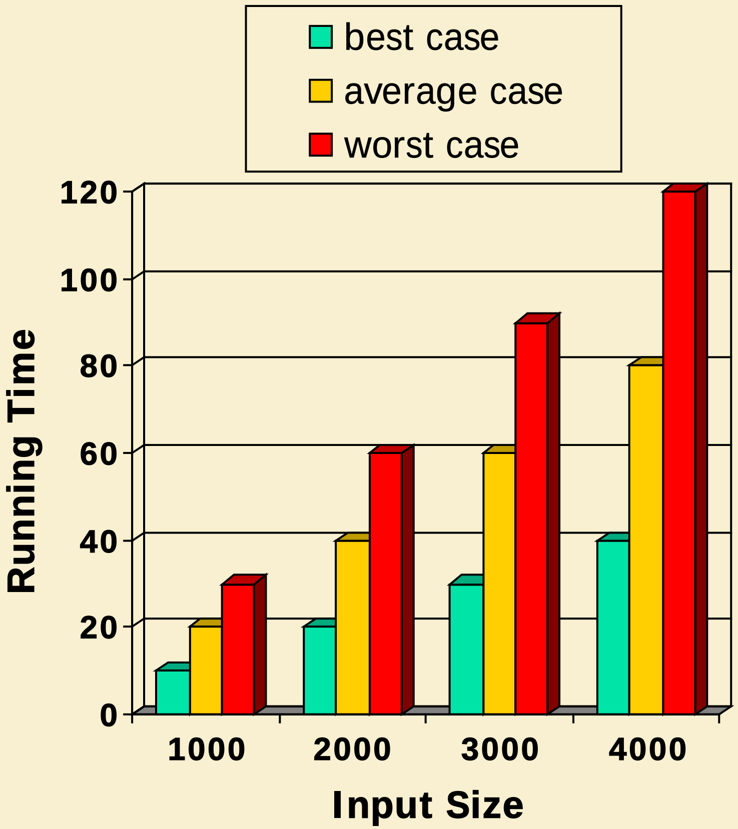
- Algorithm running time typically grows with the input size
- **Analyze running time as a function of input size**
  - ***T***(***n***), where ***n*** is integer expressing the input size
  - Example: ***n*** is the size of the input array

How to Calculate Running Time

- Even on inputs of the same size, running time can be different
  - Example: algorithm that finds the first prime number in an array by scanning it left to right
- Idea: analyze running time in the
  - best case
  - worst case
  - average case

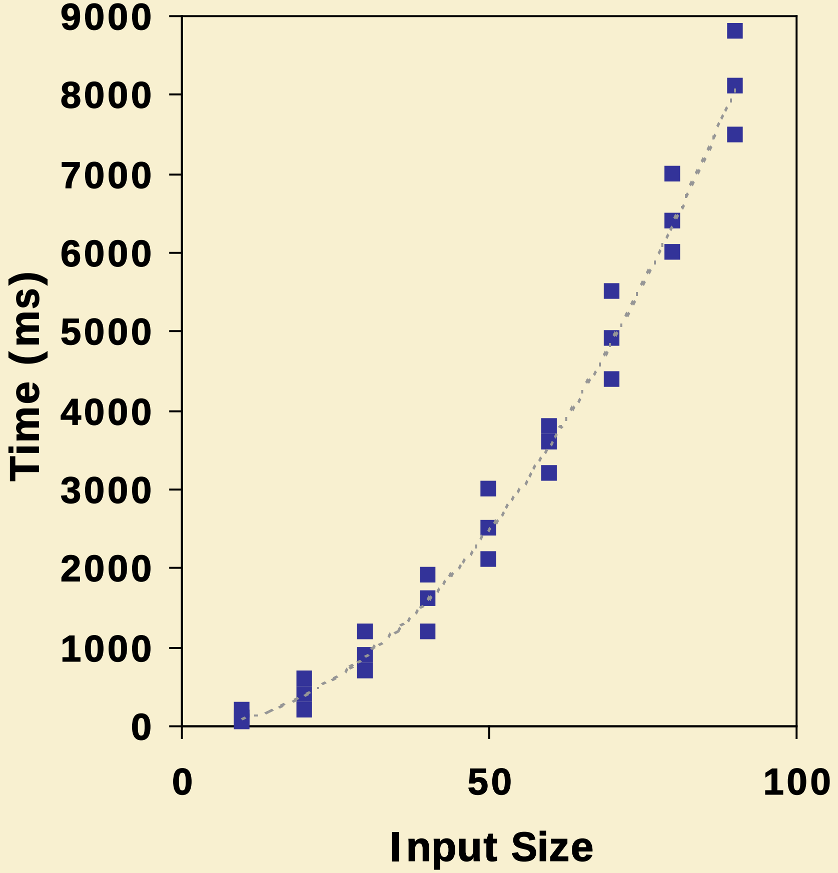
How to Calculate Running Time

- Best case running time is usually useless
- Average case running time is very useful but often difficult to determine
- We focus on the worst case running time
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



Experimental Evaluation of Running Time

- Implementing algorithm
- Run the program with inputs of varying size and composition
- Use System.currentTimeMillis() to measure actual running time
- Plot results



Limitations of Experiments

- Experimental evaluation of running time is useful *but*
  - necessary to implement algorithm, which may be difficult
  - results may not be indicative of the running time on other inputs not included in the experiment
  - to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis of Running Time

- Uses a pseudo-code description of the algorithm instead of implementation
- Characterizes running time as a function of the input size, ***n***
- Takes into account all possible inputs
- Allows to evaluate the speed of algorithm independent of hardware/software

Pseudocode

- We mostly use pseudocode to describe algorithm
- Pseudocode is a high-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max array element

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***

Pseudocode Details

- Control flow
  - **if … then … [else …]**
  - **while … do …**
  - **repeat … until …**
  - **for … do …**
  - Indentation replaces braces
- 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***

Pseudocode Details

- Method call

  var.method **(*arg*** [, ***arg***…])

- Return value

  return *expression*

- Expressions

  $\leftarrow$ Assignment (like $=$ in Java)

  $=$ Equality testing (like $==$ in Java)

  n^2 superscripts and other mathematical formatting allowed

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*

Primitive Operations

- For theoretical analysis, count **primitive** or **basic** operations
  - are simple computations performed by algorithm
- Basic operations are:
  - Identifiable in pseudocode
  - Largely independent from the programming language
  - Exact definition not important (will see why later)
  - **Assumed to take a constant amount of time**
    - i.e. independent of the input size

Primitive Operations

- Examples of primitive operations:
  - evaluating an expression x^2+e^y
  - assigning a value to a variable $cnt \leftarrow cnt+1$
  - indexing into an array A[5]
  - calling a method mySort(A,n)
  - returning from a method return(cnt)

Counting Primitive Operations

- Determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

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

Estimating Running Time

- Algorithm ***arrayMax*** executes $7n - 1$ primitive operations in the worst case
- Let

  a = time taken by the fastest primitive operation

  b = time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of ***arrayMax***, then

$$a\,(7n - 1) \leq T(n) \leq b(7n - 1)$$

  - bounded by two linear functions

Growth Rate of Running Time

$$a\,(7n - 1) \leq T(n) \leq b(7n - 1)$$

- ***T***(***n***) has a **linear growth** rate
  - grows proportionally with ***n***, i.e. running time is ***n*** times a constant factor
- Changing hardware/software environment affects ***T***(***n***) by a constant factor, but does not change growth rate
- Thus linear growth rate of ***T***(***n***) is an intrinsic property of algorithm ***arrayMax***
- Want to focus on the **growth rate** of an algorithm, i.e. "the big picture"

Growth Rates Examples

- These often appear in algorithm analysis:
  - constant $\approx$ ***1***
  - logarithmic $\approx$ log ***n***
  - linear $\approx$ ***n***
  - N-Log-N $\approx$ ***n*** log ***n***
  - quadratic $\approx$ $\boldsymbol{n}^2$
  - cubic $\approx$ $\boldsymbol{n}^3$
  - exponential $\approx$ $\mathbf{2}^n$

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}

Growth Rates Illustration

Running Time in ms (10^{-3} of sec)	Maximum Problem Size (n) 1000 ms (1 second)	60000 ms (1 minute)	$36*10^5$ ms (1 hour)
n	1000	60,000	3,600,000
n^2	32	245	1,897
2^n	10	16	22

Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order (slowly growing) terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function
- How do we "ignore" constant factors and focus on the essential part (growth rate) of the running time?

Asymptotic Analysis Motivation

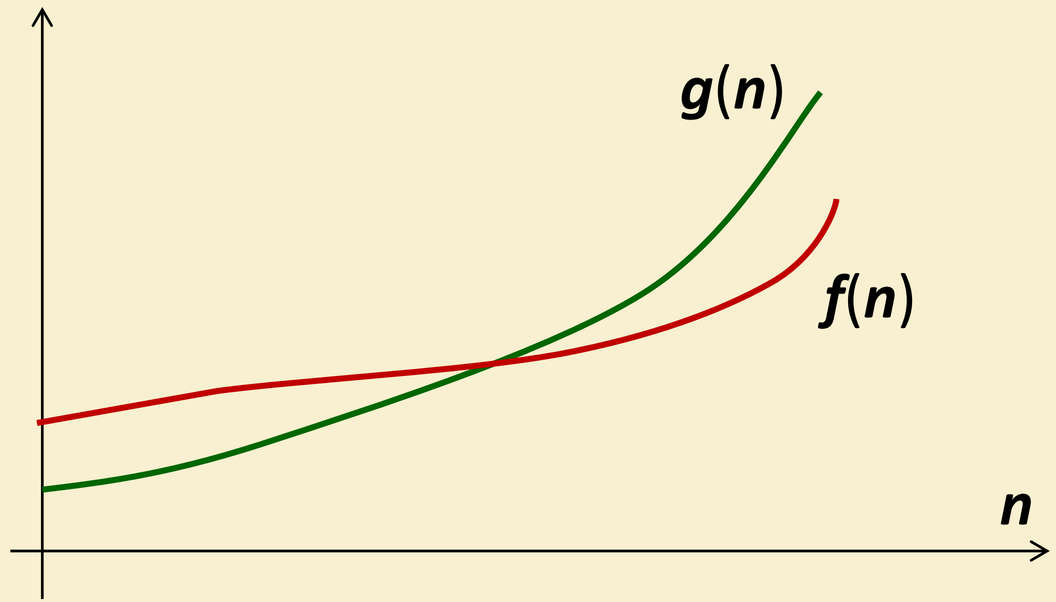
- big-Oh notation is used widely to characterize running times and space bounds
- big-Oh notation allows us to
  - ignore constant factors and
  - ignore lower order terms
  - focus on the main components that affect growth rate

Asymptotic Analysis Motivation

- Want to show $T(n)$ has some growth rate
  - Example: $T(n) = 2n + 10$
- Rename $T(n) = f(n)$ for now
  - consistency with the most commonly used notation
- Growth rate is also some function, call it $g(n)$
  - $g(n) = n$, $g(n) = n^2$, etc.
- Want to show $f(n)$ has growth rate $g(n)$
  - Example: $f(n) = 2n + 10$ has growth rate $g(n) = n$
- **Main step**: show $f(n)$ grows slower or the same as $g(n)$

Asymptotic Analysis Motivation

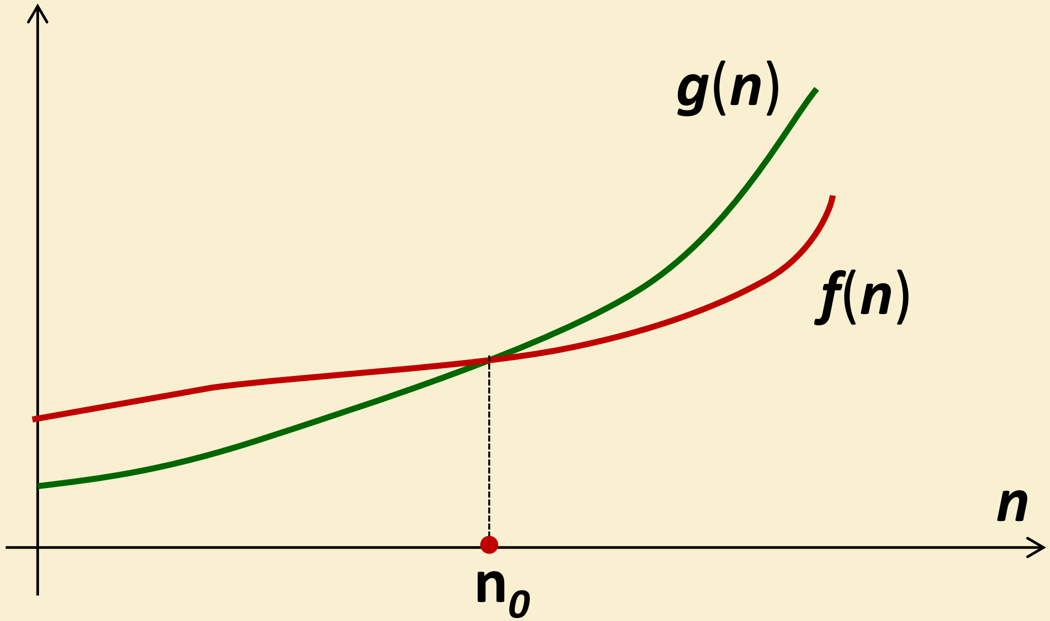
- **Main step**: show ***f***(***n***) grows slower or the same as ***g***(***n***)



- Need to show ***f***(***n***) ≤ ***g***(***n***)

Asymptotic Analysis Motivation

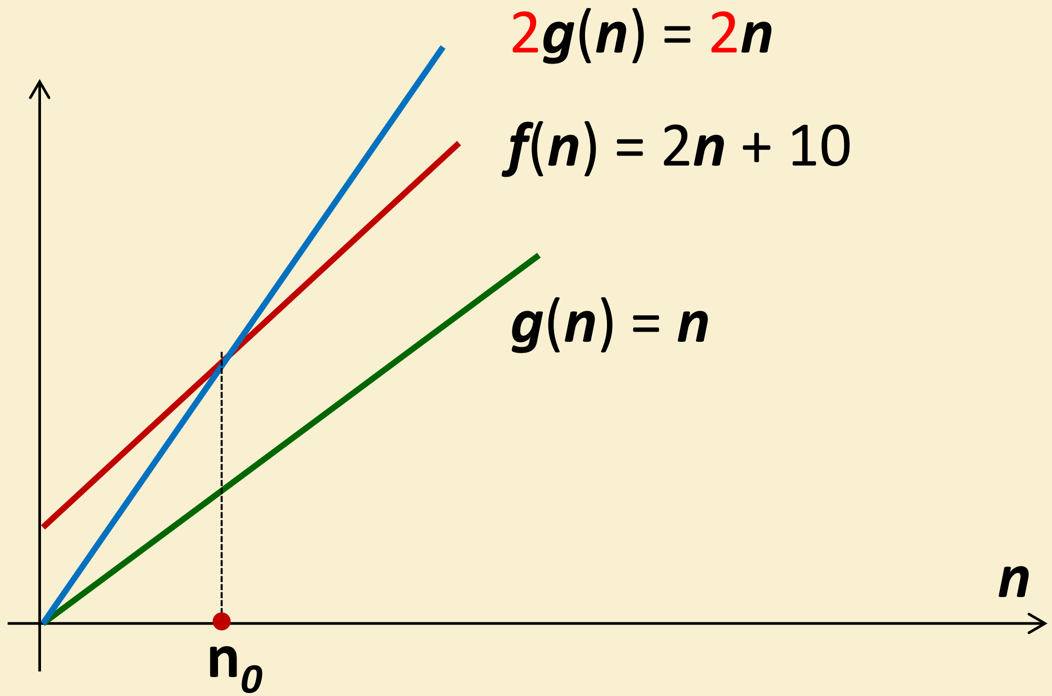
- Need to show $f(n) \leq g(n)$



- initial range of n can be ignored, interested in what happens eventually, for large n
  - thus name "asymptotic" analysis
- formally: for $n \geq n_0$, for some constant integer $n_0 \geq 1$

Asymptotic Analysis Motivation

- Constants do not affect growth rate, want to ignore them



- Instead of $f(n) \le g(n)$ show $f(n) \le c\,g(n)$ for a positive constant c
- and, as before, for $n \ge n_0$, for some constant integer $n_0 \ge 1$

Big-Oh Notation Definition

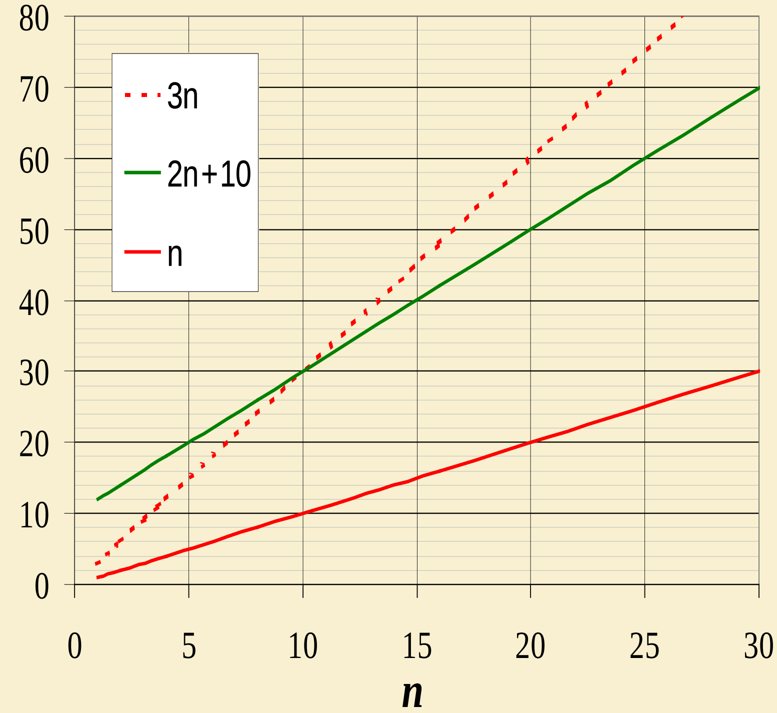
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq cg(n) \text{ for } n \geq n_0$$

- Meaning: $f(n)$ does not grow faster than $g(n)$ asymptotically

Example Proof

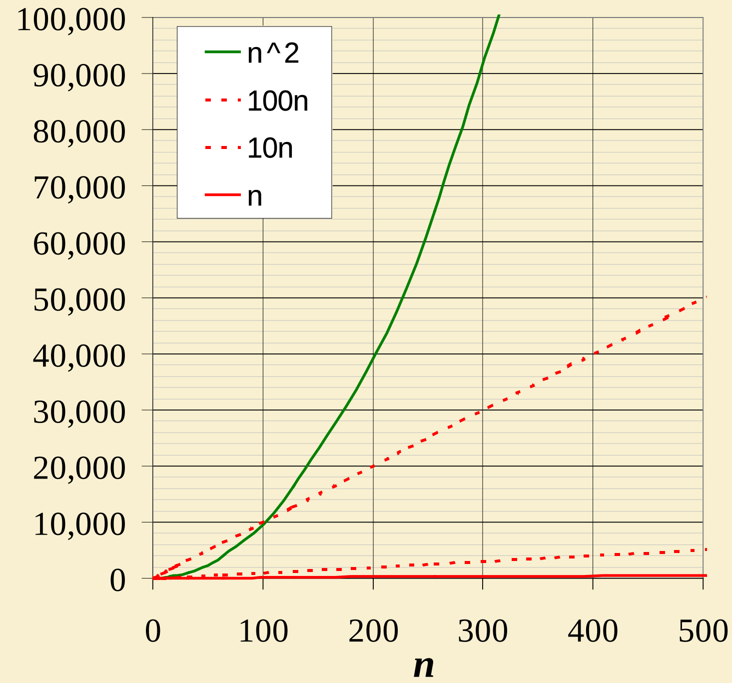
- Show that $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$
  - $(c - 2)\, n \ge 10$
  - $n \ge 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



- For $c = 3$, we took the exact intersection point for n_0
- But $n_0 = 11, 12, \ldots$ would work just as well
- There are infinitely many choices for c and n_0

"Negative" Example

- Example: n^2 is not $O(n)$
  - Suppose $n^2 \leq cn$ for some $c > 0$ and all $n \geq n_0$
  - $n \leq c$ for some $c > 0$ and all $n \geq n_0$
  - take $m = \lceil c \rceil + n_0$
  - $m > c$ and $m > n_0$
  - ***Contradiction!***



More Big-Oh Examples

- $7\boldsymbol{n} + 2$ is $\boldsymbol{O}(\boldsymbol{n})$

Proof:

$7\boldsymbol{n} + 2 \le 7\boldsymbol{n} + 2\boldsymbol{n}$ for all $\boldsymbol{n} \ge 1$

$7\boldsymbol{n} + 2 \le 9\boldsymbol{n}$ for all $\boldsymbol{n} \ge 1$

Take $\boldsymbol{c} = 9$ and $\boldsymbol{n}_0 = 1$, then

$7\boldsymbol{n} + 2 \le \boldsymbol{cn}$ for all $\boldsymbol{n} \ge \boldsymbol{n_0}$

More Big-Oh Examples

- $3\boldsymbol{n}^3 + 20\boldsymbol{n}^2 + 5$ is $\boldsymbol{O}(\boldsymbol{n}^3)$

Proof:

$3\boldsymbol{n}^3 + 20\boldsymbol{n}^2 + 5 \leq 3\boldsymbol{n}^3 + 20\boldsymbol{n}^3 + 5\boldsymbol{n}^3$ for $\boldsymbol{n} \geq 1$

$3n^3 + 20n^2 + 5 \leq 28\ n^3$ for $\boldsymbol{n} \geq 1$

Take $c = 28$, $\boldsymbol{n}_0 = 1$ then

$$3n^3 + 20n^2 + 5 \leq cn^3 \text{ for } \boldsymbol{n} \geq \boldsymbol{n}_0$$

More Big-Oh Examples

- $3 \log \boldsymbol{n} + 5$ is $O(\log \boldsymbol{n})$

Proof:

$3 \log \boldsymbol{n} + 5 \leq 3 \log \boldsymbol{n} + 5 \log \boldsymbol{n} = 8 \log \boldsymbol{n}$ for $\boldsymbol{n} \geq 2$

$3 \log \boldsymbol{n} + 5 \leq 8 \log \boldsymbol{n}$ for $\boldsymbol{n} \geq 2$

Take $\boldsymbol{c} = 8$, $\boldsymbol{n}_0 = 2$, then

$$3 \log \boldsymbol{n} + 5 \leq \boldsymbol{c} \log \boldsymbol{n} \quad \text{for } \boldsymbol{n} \geq \boldsymbol{n}_0$$

Big-Oh Etiquette

- Use the smallest possible class of functions
  - ✗ $2n$ is $O(n^2)$
    - true but is nota as accurate and informative
    - therefore considered to be a “poor taste”
  - ☑ $2n$ is $O(n)$
    - precise and accurate
    - preferred statement
- Use the simplest expression of the class
  - ✗ $3n + 5$ is $O(3n)$
    - true but more complicated than needed
  - ☑ $3n + 5$ is $O(n)$
    - preferred statement

Big-Oh is an Upper Bound

- Statement "***f***(***n***) is ***O***(***g***(***n***))" means that the growth rate of ***f***(***n***) is no more than the growth rate of ***g***(***n***)
- Thus big-Oh notation gives an upper bound on the growth rate of a function

	f(***n***) is ***O***(***g***(***n***))	***g***(***n***) is ***O***(***f***(***n***))
g(***n***) grows more	Yes	No
f(***n***) grows more	No	Yes
Same growth	Yes	Yes

- Are there theoretical concepts that say
  - ***f***(***n***) grows at the same rate as ***g***(***n***)?
  - ***f***(***n***) and ***g***(***n***) have the same growth rate?

Big-Omega, a Relative of Big-Oh

- **big-Omega**
  - $f(n)$ is $\Omega(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

    $$f(n) \geq cg(n) \text{ for } n \geq n_0$$

  - means $f(n)$ is asymptotically **greater than or equal** to $g(n)$
  - $f(n)$ is $\Omega(g(n))$ if and only if $g(n)$ is $O(f(n))$

Big-Theta, a Relative of Big-Oh

- **big-Theta**
  - $\boldsymbol{f}(\boldsymbol{n})$ is $\Theta(\boldsymbol{g}(\boldsymbol{n}))$ if there are real constants $\boldsymbol{c'} > 0$ and $\boldsymbol{c''} > 0$ and an integer constant $\boldsymbol{n_0} \geq 1$ such that

$$\boldsymbol{c'g}(\boldsymbol{n}) \leq \boldsymbol{f}(\boldsymbol{n}) \leq \boldsymbol{c''g}(\boldsymbol{n}) \text{ for } \boldsymbol{n} \geq \boldsymbol{n_0}$$

  - means $\boldsymbol{f}(\boldsymbol{n})$ is asymptotically **equal** to $\boldsymbol{g}(\boldsymbol{n})$
  - $\boldsymbol{f}(\boldsymbol{n})$ is $\Theta(\boldsymbol{g}(\boldsymbol{n}))$ if and only if $\boldsymbol{g}(\boldsymbol{n})$ is $\boldsymbol{O}(\boldsymbol{f}(\boldsymbol{n}))$ **and** if $\boldsymbol{f}(\boldsymbol{n})$ is $\boldsymbol{O}(\boldsymbol{g}(\boldsymbol{n}))$

Big-Oh Polynomial Rule

$$f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$$

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$
  1. Drop lower-order terms
  2. Drop constant factors

Useful Big-Oh Rules

1. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$ then

   $$d(n) + e(n) \text{ is } O(f(n) + g(n))$$

   $$d(n)e(n) \text{ is } O(f(n)\, g(n))$$

2. If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$ then

   $$d(n) \text{ is } O(g(n))$$

3. If $p(n)$ is a polynomial in n then

   $$\log p(n) \text{ is } O(\log(n))$$

Asymptotic Algorithm Analysis

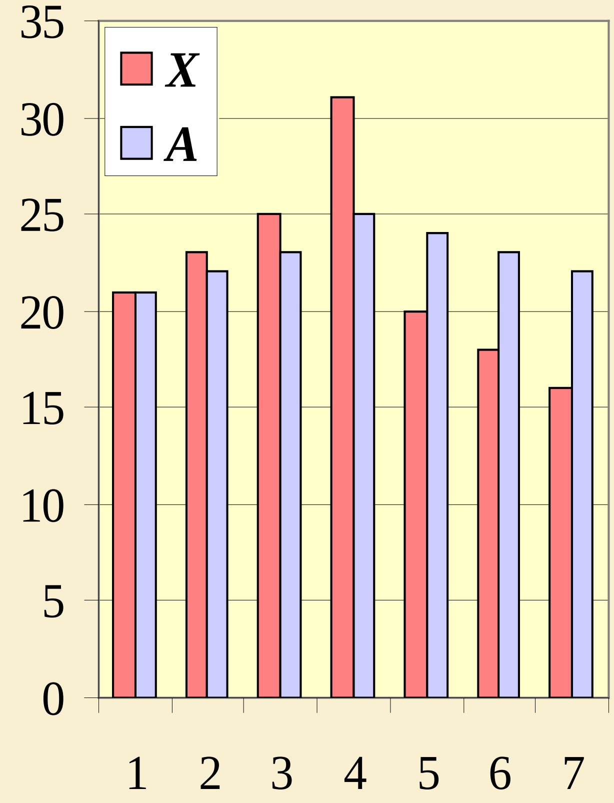
- Asymptotic algorithm analysis determines running time in big-Oh notation (or big-Theta, or big-Omega)
- To perform the asymptotic analysis
  - find the worst-case number of primitive operations executed as a function of input size
  - express this function with big-Oh notation
- Example:
  - Algorithm ***arrayMax*** executes at most 7***n*** – 1 primitive operations
  - Algorithm ***arrayMax*** runs in ***O***(***n***) time
- Since constant factors and lower-order terms are eventually dropped , can disregard them when counting primitive operations

Computing Prefix Averages

- The ***i***-th prefix average of an array ***X*** is average of first (i + 1) elements of ***X***:

  $A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$

- Computing array ***A*** of prefix averages of another array ***X*** has applications to financial analysis



Prefix Averages: Quadratic Algorithm

Algorithm ***prefixAverages1***(***X***, ***n***)
Input array ***X*** of ***n*** integers
Output array ***A*** of prefix averages of ***X***

	#operations
$A \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$\quad s \leftarrow X[0]$	n
$\quad$ **for** $j \leftarrow 1$ **to** i **do**	$1 + 2 + \ldots + (n - 1)$
$\quad\quad s \leftarrow s + X[j]$	$1 + 2 + \ldots + (n - 1)$
$\quad A[i] \leftarrow s / (i + 1)$	n
return A	1

Arithmetic Progression

- Running time of ***prefixAverages1*** is

$$O(1 + 2 + \ldots + n)$$

- Adding up

$$
\begin{array}{rcccccccc}
 & S & = & 1 & + & 2 & + & \ldots & + \; n \\
+ & S & = & n & + & (n-1) & + & \ldots & + \; 1 \\
\hline
 & 2S & = & (n+1) & + & (n+1) & + & & (n+1)
\end{array}
$$

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

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

- The sum of the first n integers is $n(n+1)/2$
- ***prefixAverages1*** runs in $O(n^2)$ time

Prefix Averages: Linear Algorithm

Algorithm ***prefixAverages2***(***X, n***)	
Input array ***X*** of ***n*** integers	
Output array ***A*** of prefix averages of ***X***	
	#operations
A $\leftarrow$ new array of ***n*** integers	***n***
s $\leftarrow$ 0	1
for ***i*** $\leftarrow$ 0 **to** ***n*** – 1 **do**	***n***
s $\leftarrow$ ***s*** + ***X***[***i***]	***n***
A[***i***] $\leftarrow$ ***s*** / (***i*** + 1)	***n***
return ***A***	1

- 4***n***+2 is ***O***(***n***)
- Algorithm ***prefixAverages2*** runs in ***O***(***n***) time

Math you need to Review

- Summations
- Logarithms and Exponents
  - **properties of logarithms:**

    $\log_b(xy) = \log_b x + \log_b y$

    $\log_b (x/y) = \log_b x - \log_b y$

    $\log_b x^a = a\log_b x$

    $\log_b a = \log_x a/\log_x b$
  - **properties of exponentials**:

    $a^{(b+c)} = a^b a^c$

    $a^{bc} = (a^b)^c$

    $a^b / a^c = a^{(b-c)}$

    $b = a^{\log_a b}$

    $b^c = a^{c*\log_a b}$

Final Notes

- We focus on the asymptotic growth using big-Oh notation, but practitioners do care about constant factors occasionally
- Suppose
  - Algorithm A has running time 30000n
  - Algorithm B has running time $3n^2$
- Asymptotically, algorithm A is better than algorithm B
- However, if the problem size you deal with is always less than 10000, then the quadratic one is faster