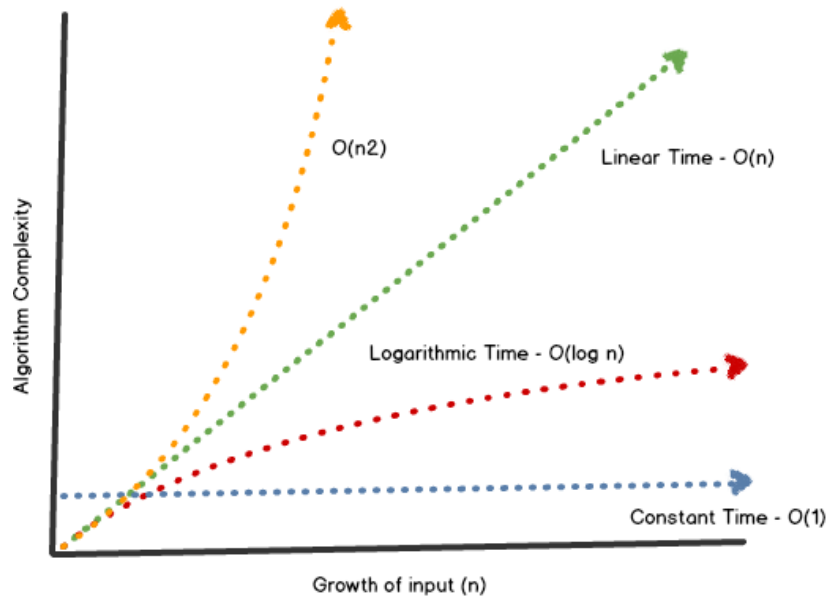
Comparing algorithms: A review of Computational Complexity Analysis



Some slides are from Data Structures and Algorithms in Java, by M. T. Goodrich, et. al.

Objectives

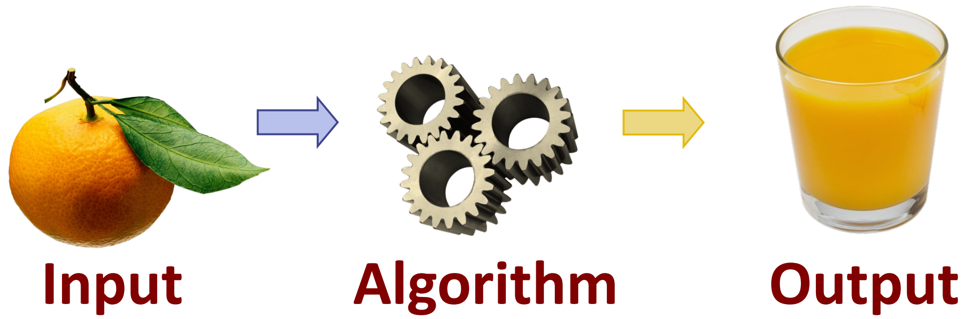
- ❑ Algorithm analysis
- ❑ Introducing the computational complexity
- ❑ Big-O notation and examples

Algorithm Analysis

Algorithm analysis is a methodology to predict the resources that the algorithm requires

- Computational time
- Computer memory

We'll focus on computational time. It does not mean memory is not important. Generally, there is a trade-off between the two factors o Space-time trade-off is a common term



Experimental Studies

- Write a program implementing the algorithm

- Run the program with inputs of varying size and composition, noting the time needed:

- Plot the results

```
long startTime = System.currentTimeMillis();          // record the starting time
/* (run the algorithm) */
long endTime = System.currentTimeMillis();            // record the ending time
long elapsed = endTime − startTime;                   // compute the elapsed time
```

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult

- Results may not be indicative of the running time on other inputs not included in the experiment.

- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

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

Input Size

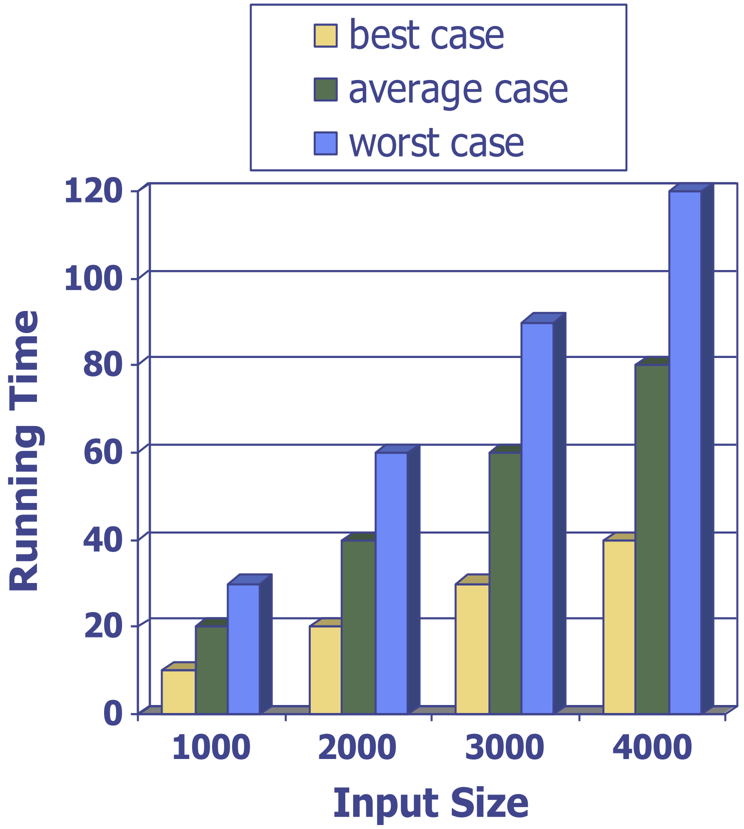
What is meant by the input size n? Provide some application-specific examples.

- Dictionary: # words
- Restaurant: # customers or # food choices
- Airline: # flights, # luggage, or # costumers

We want to express the number of operations performed as a function of the input size n.

Running Time

- ❑ The running time of an algorithm typically grows with the input size.
- ❑ Average case time is often difficult to determine.
- ❑ We focus on the worst case running time.
  - ■ Easier to analyze
  - ■ Crucial to applications such as games, finance and robotics



Pseudocode

- ❑High-level description of an algorithm
- ❑More structured than English prose
- ❑Less detailed than a program
- ❑Preferred notation for describing algorithms
- ❑Programming language independent

Pseudocode Details

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

Algorithm ***method*** (***arg*** [, ***arg***…])
    Input …
    Output …

- Method call

method (***arg*** [, ***arg***…])

- Return value

return ***expression***

- Expressions:
  - $\leftarrow$ Assignment
  - $=$ Equality testing

n^2 Superscripts and other mathematical formatting allowed

Primitive Operations

- Basic computations performed by an algorithm
- Each operation corresponding to a low-level instruction with a constant execution time
- Largely independent from the programming language

- Examples:
  - Evaluating an expression (x + y)
  - Assigning a value to a variable (x = 5)
  - Comparing two numbers (x < y)
  - Indexing into an array (A[i])
  - Calling a method (myCalculator.sum())
  - Returning from a method (return result)

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
Algorithm ArrayMax(A, n)                  # operations
currentMax ←A[0]                          2: (1 +1)
for i←1;i<n; i←i+1 do                     3n-1: (1 + n+2(n- 1))
     if A[i]>currentMax then              2(n − 1)
          currentMax ←A[i]                2(n − 1)
     endif
endfor
return currentMax                         1
                                     Total:  7n − 2
```

Algorithm efficiency: growth rate

- An algorithm’s time requirements can be expressed as a function of (problem) input size

- Problem size depends on the particular problem:
  - For a search problem, the problem size is the number of elements in the search space
  - For a sorting problem, the problem size is the number of elements in the given list

- How quickly the time of an algorithm grows as a function of problem size -- this is often called an algorithm’s growth rate

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects ***T***(***n***) by a constant factor, but
  - Does not alter the growth rate of ***T***(***n***)
- The linear growth rate of the running time ***T***(***n***) is a property of algorithm arrayMax

Why does growth rate matters?

N	$\log_2 N$	$N \log_2 N$	N^2	N^3	2^N
1	0	1	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4,096	65,536
32	5	160	1,024	32,768	4,294,967,296
64	6	384	4,096	262,144	approximately 20 billion billion
128	7	896	16,384	2,097,152	It would take a fast computer a trillion billion years to execute this many instructions
256	8	2,048	65,536	16,777,216	Do not ask!

Algorithmic time complexity

Rather than counting the exact number of primitive operations, we approximate the runtime of an algorithm as a function of data size – ***time complexity***.

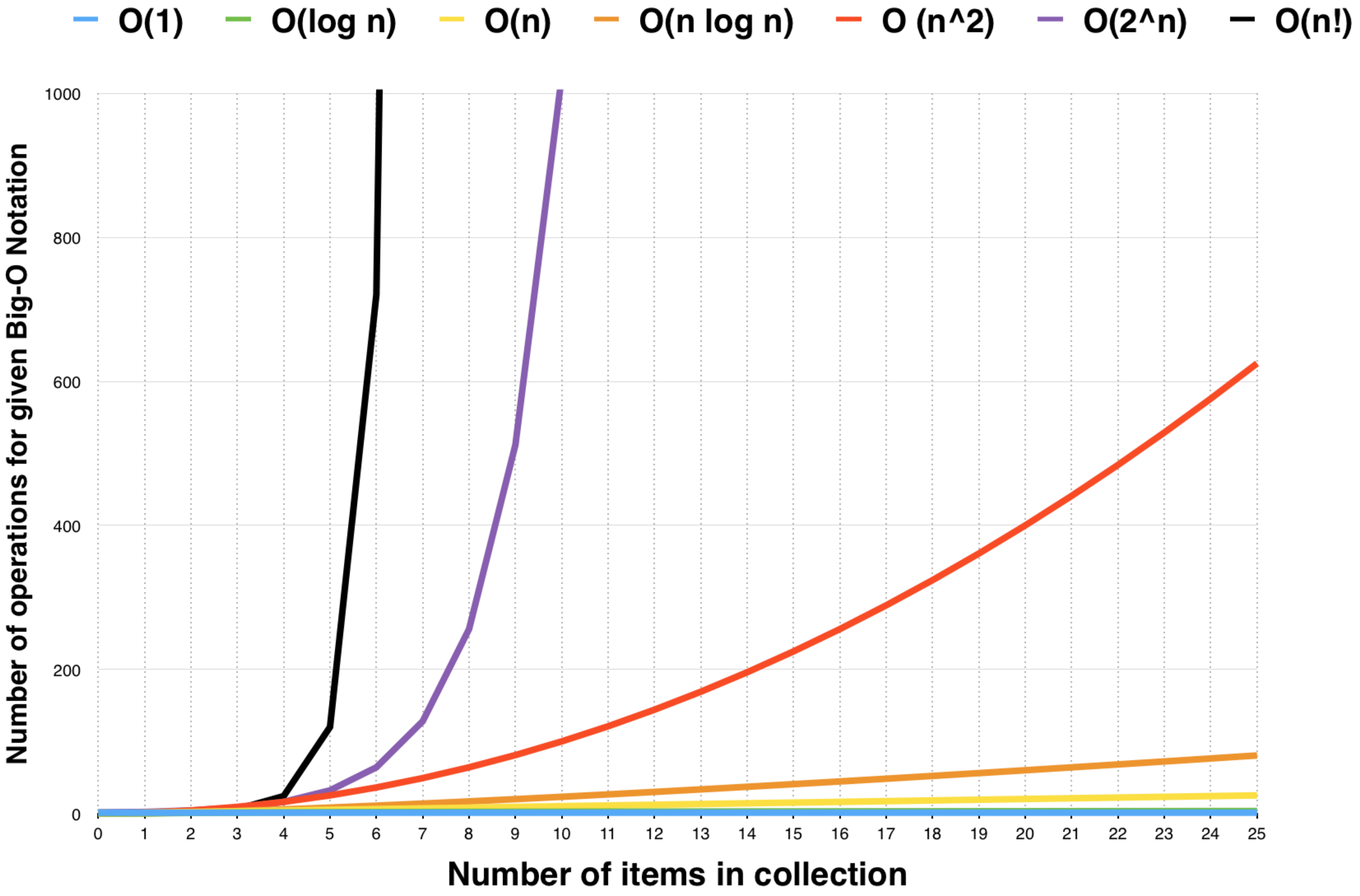
We say an algorithms belongs to a complexity classes.

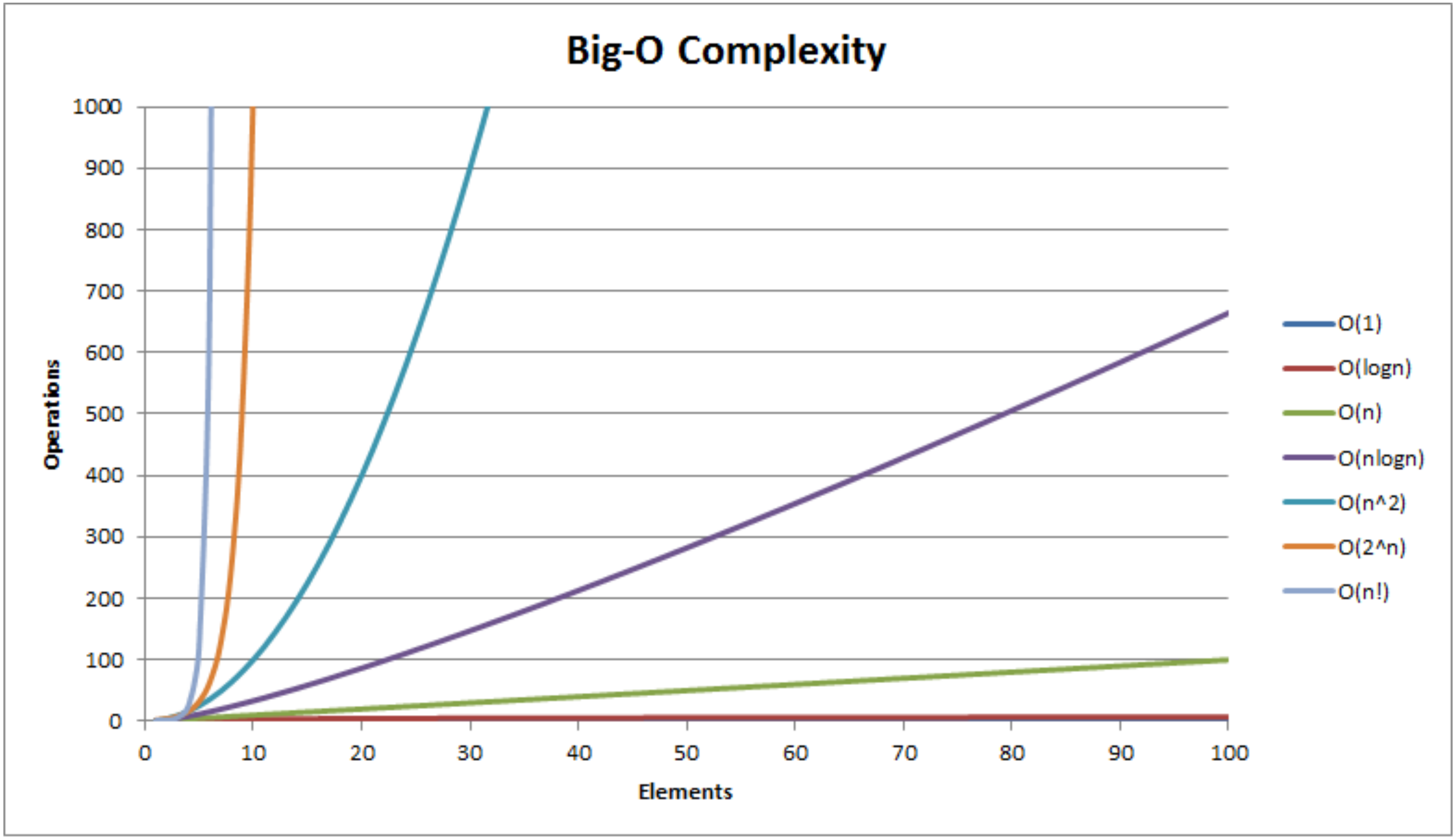
We’ll not cover complexity classes in detail – they will be covered in Algorithm Analysis course.

We’ll briefly discuss seven basic functions which are often used in complexity analysis.

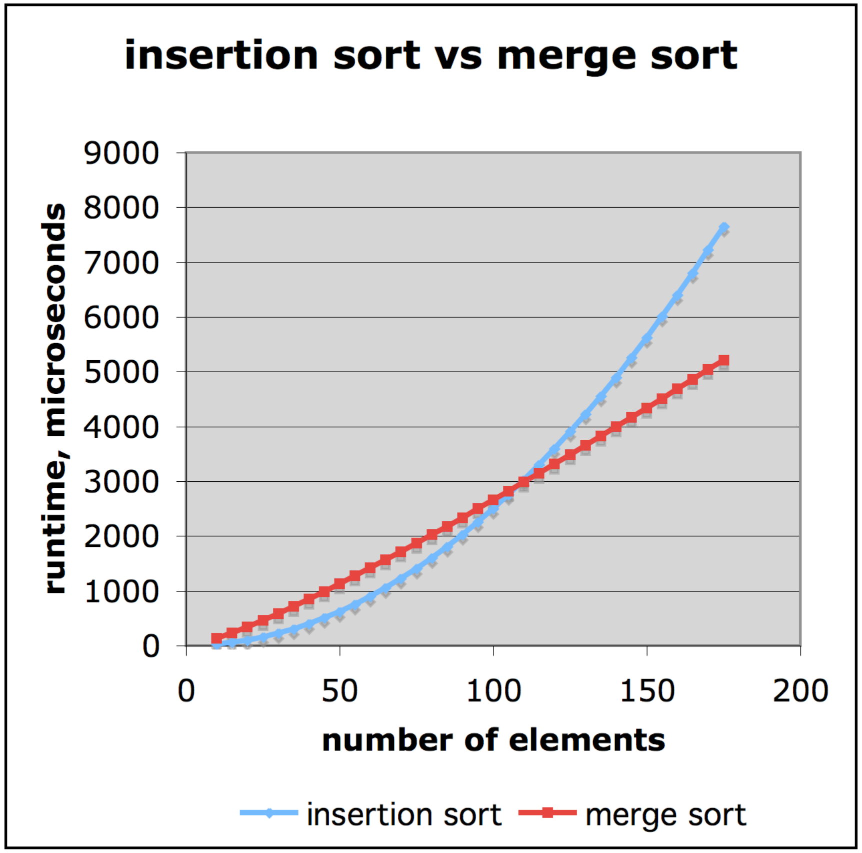
Algorithmic time complexity

- Functions that are often seen in algorithm analysis:
  - Constant: $f(n) = c \approx \mathit{1}$
  - Logarithmic: $f(n) \approx \log n$
  - Linear: $f(n) \approx n$
  - N-Log-N: $f(n) \approx n \log n$
  - Quadratic: $f(n) \approx n^2$
  - Cubic: $f(n) \approx n^3$
  - Exponential: $f(n) \approx 2^n$
  - Factorial: $f(n) \approx n!$


O(1)
O(log n)
O(n)
O(n log n)
O (n^2)
O(2^n)
O(n!)
1000
800
600
400
200
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Number of operations for given Big-O Notation
Number of items in collection


Big-O Complexity
Operations
Elements
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Comparison of Two Algorithms



insertion sort is

$n^2 / 4$

merge sort is

$2\, n \lg n$

sort a million items?

insertion sort takes roughly 70 hours

while

merge sort takes roughly 40 seconds

This is a slow machine, but if 100 x as fast then it's 40 minutes versus less than 0.5 seconds

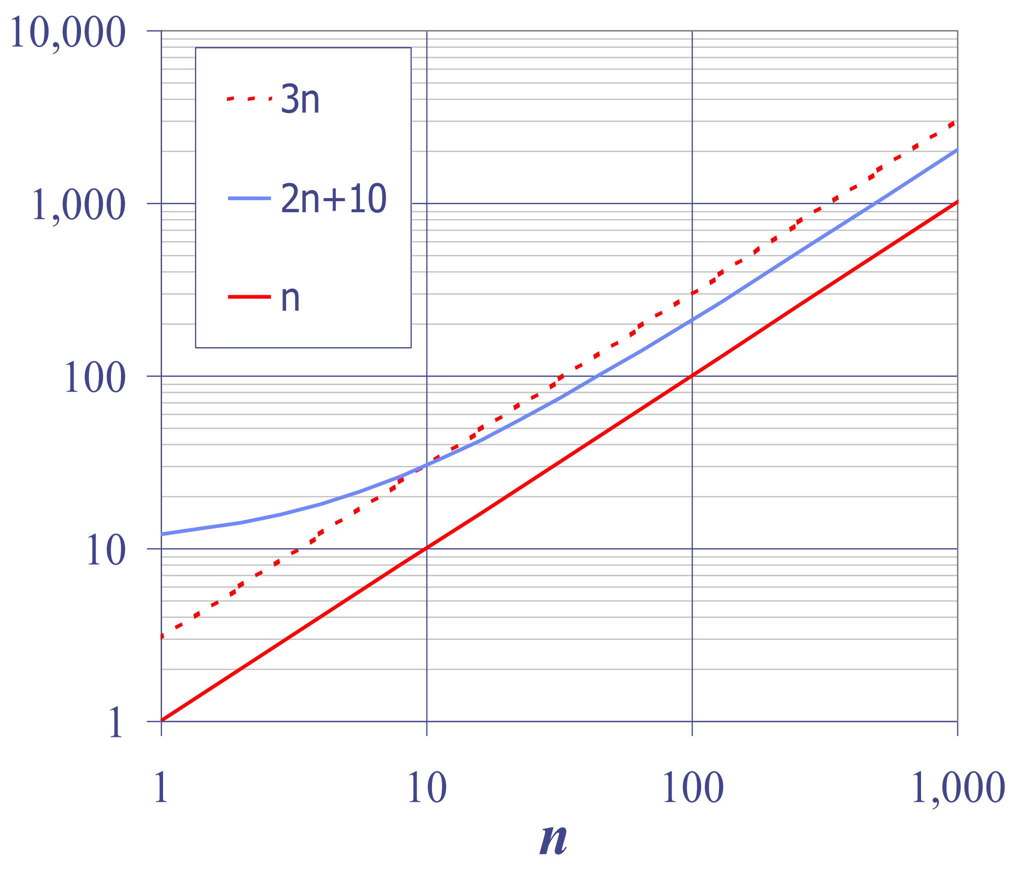
Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2\boldsymbol{n} + 10^5$ is a linear function
  - $10^5\boldsymbol{n}^2 + 10^8\boldsymbol{n}$ is a quadratic function

Big-O Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants
- c and n_0 such that

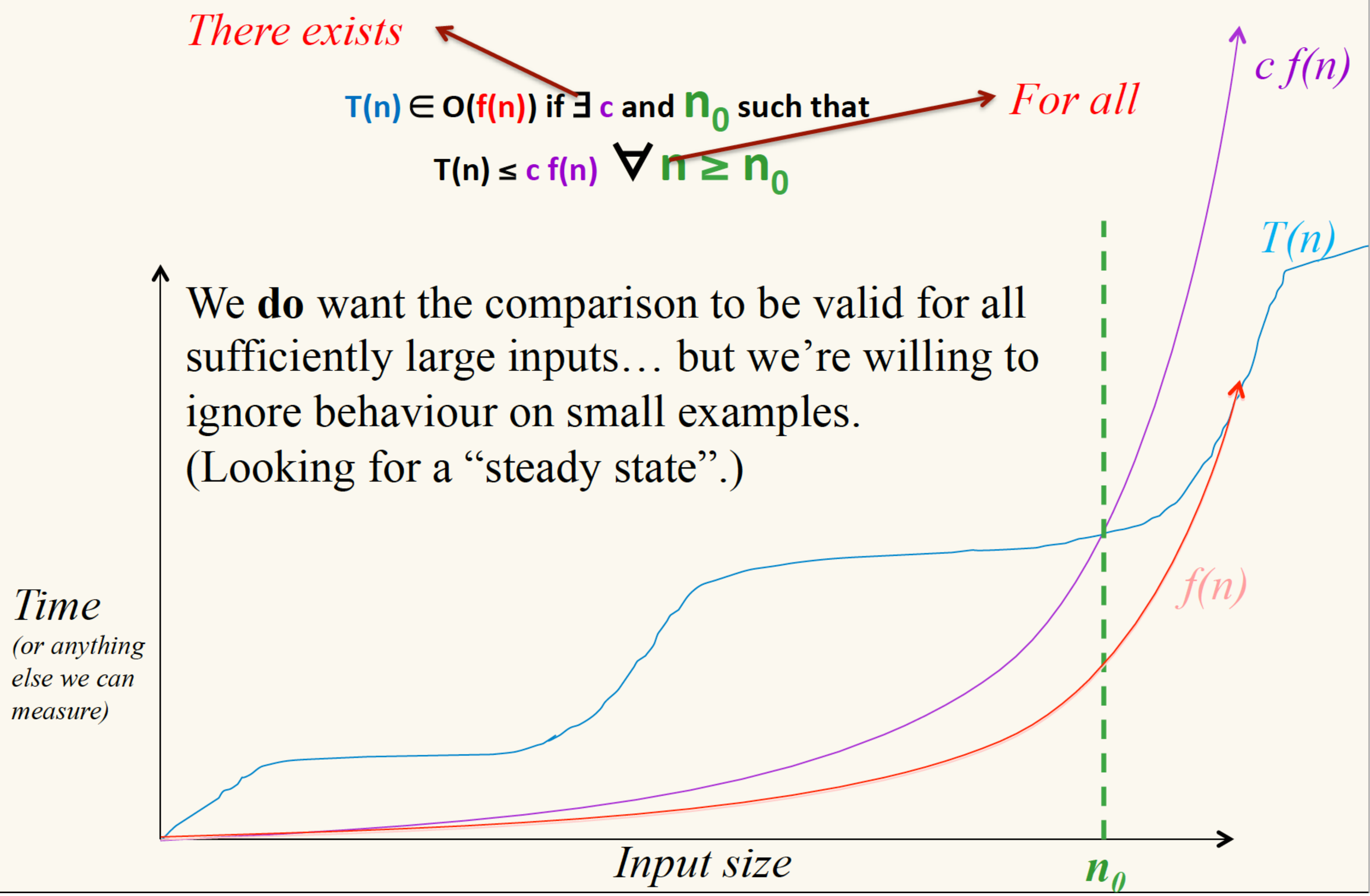
  $f(n) \leq cg(n)$ for $n \geq n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Big-O Notation

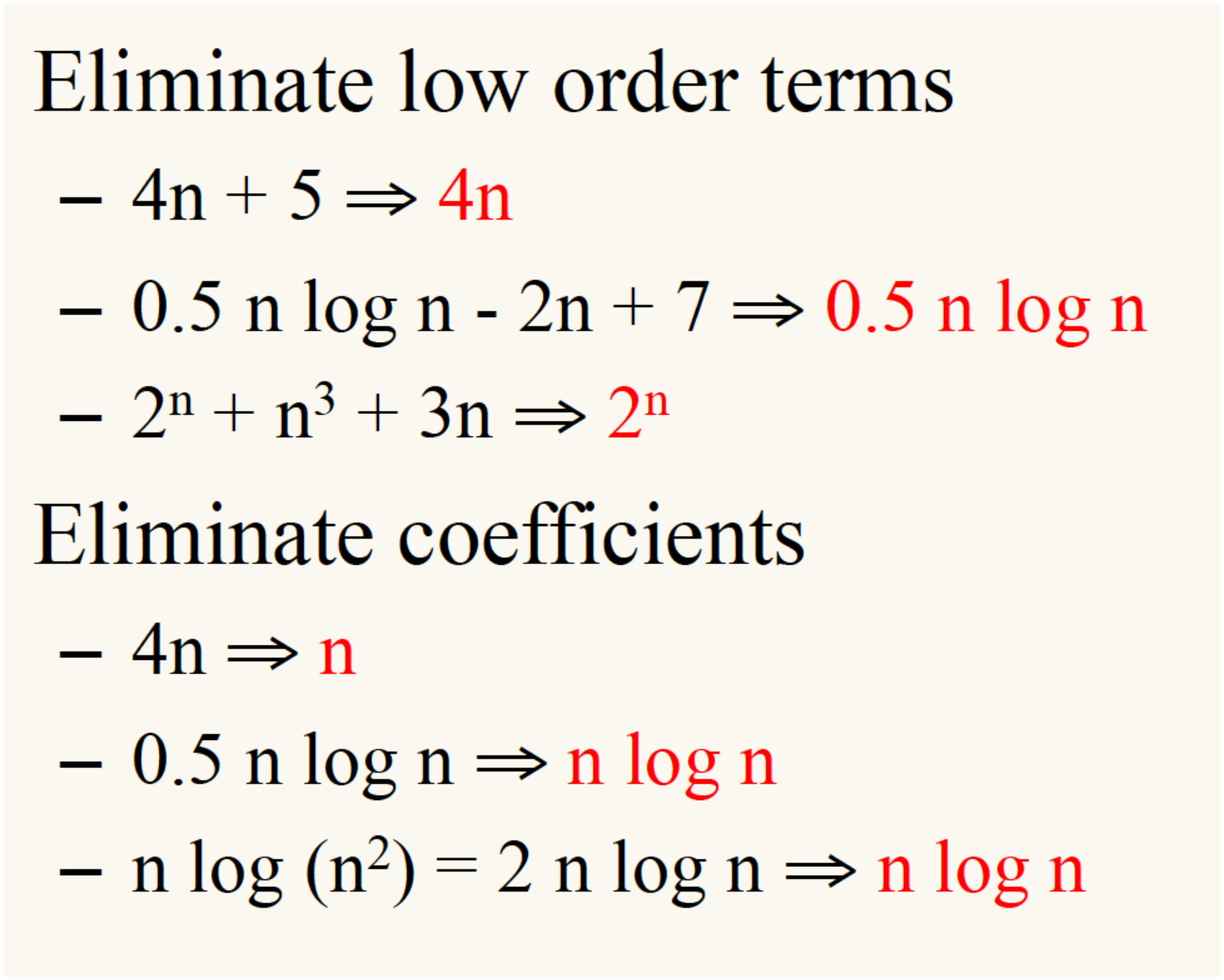
$T(n) \in O(f(n))$ if $\exists$ c and n_0 such that

$T(n) \leq c\, f(n) \ \forall\, n \geq n_0$



We **do** want the comparison to be valid for all sufficiently large inputs… but we're willing to ignore behaviour on small examples. (Looking for a "steady state".)

Asymptotic Analysis Hacks



Eliminate low order terms

- $4n + 5 \Rightarrow 4n$
- $0.5\ n \log n - 2n + 7 \Rightarrow 0.5\ n \log n$
- $2^n + n^3 + 3n \Rightarrow 2^n$

Eliminate coefficients

- $4n \Rightarrow n$
- $0.5\ n \log n \Rightarrow n \log n$
- $n \log (n^2) = 2\ n \log n \Rightarrow n \log n$

More Big-O Examples

- $7n - 2$

$7n-2$ is $O(n)$
need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq c.n$ for $n \geq n_0$
this is true for $c = 7$ and $n_0 = 1$

- $3\,n^3 + 20\,n^2 + 5$

$3\,n^3 + 20\,n^2 + 5$ is $O(n^3)$
need $c > 0$ and $n_0 \geq 1$ such that $3\,n^3 + 20\,n^2 + 5 \leq c.n^3$ for $n \geq n_0$
this is true for $c = 4$ and $n_0 = 21$

- $3 \log n + 5$

$3 \log n + 5$ is $O(\log n)$
need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c.\log n$ for $n \geq n_0$
this is true for $c = 8$ and $n_0 = 2$

Rates of Growth

- Suppose a computer executes 10^{12} ops per second:

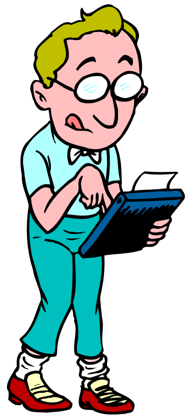
n =	**10**	**100**	**1,000**	**10,000**	**10^{12}**
n	10^{-11}s	10^{-10}s	10^{-9}s	10^{-8}s	1s
n lg n	10^{-11}s	10^{-9}s	10^{-8}s	10^{-7}s	40s
n^2	10^{-10}s	10^{-8}s	10^{-6}s	10^{-4}s	10^{12}s
n^3	10^{-9}s	10^{-6}s	10^{-3}s	1s	10^{24}s
2^n	10^{-9}s	10^{18}s	10^{289}s		

10^4s = 2.8 hrs ***10^{18}s = 30 billion years***

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We say that algorithm arrayMax "runs in ***O***(***n***) time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Math you need to Review



- ❑Summations
- ❑Powers
- ❑Logarithms
- ❑Proof techniques
- ❑Basic probability

❑Properties of powers:

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

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

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

$b = a^{\log_a b}$

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

❑Properties of logarithms:

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

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

$\log_b xa = a\log_b x$

$\log_b a = \log_x a / \log_x b$

Relatives of Big-O

big-Omega

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

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

big-Theta

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

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

Intuition for Asymptotic Notation

big-O
f(n) is O(g(n)) if f(n) is asymptotically less than or equal to g(n)

big-Omega
f(n) is Ω(g(n)) if f(n) is asymptotically greater than or equal to g(n)

big-Theta
f(n) is Θ(g(n)) if f(n) is asymptotically equal to g(n)

Example Uses of the Relatives of Big-O

■**5n^2 is $\Omega(n^2)$**

$f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \,.\, g(n)$ for $n \geq n_0$
let $c = 5$ and $n_0 = 1$

■**5n^2 is $\Omega(n)$**

$f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that f(n) ⍰ $c\ g(n)$ for n ⍰ n_0
let $c = 1$ and $n_0 = 1$

■**5n^2 is $\Theta(n^2)$**

$f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that f(n) $\leq c\ g(n)$ for $n \geq n_0$
Let $c = 5$ and $n_0 = 1$

Pseudo-code 1

```
Algorithm1 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
```

- Running time of algorithm is O(n)

Pseudo-code 2

```
Algorithm2 prefixAverages(A, n)
  Input array A of n integers
  Output array X of n doubles

  Let X be an array of n doubles
  for i ← 1 to n − 1 do
      a ← 0
      for j ← 0 to i − 1 do
          a ← a + A[j]
      X[i] ← a / (i+1)
  return X
```

- Running time of algorithm is $O(n^2)$