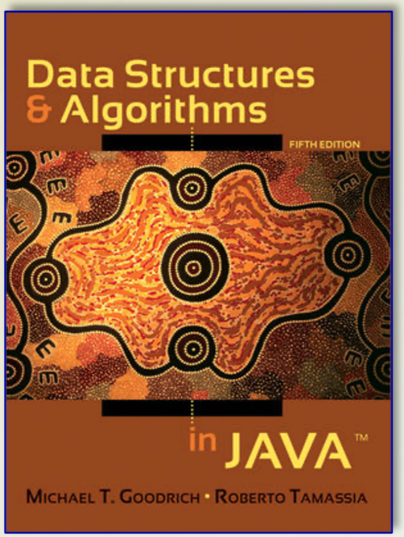
Data Structure & Algorithms in JAVA

5^{th} edition

Michael T. Goodrich
Roberto Tamassia



Chapter 4: Analysis Tools

CPSC 3200

Algorithm Analysis and Advanced Data Structure

Chapter Topics

- The Seven Functions.
- Analysis of Algorithms.
- Simple Justification Techniques.

How to analyze an algorithm

- To **analyze an algorithm** is to determine the amount of resources (such as time and storage) necessary to execute it.

- Most algorithms are designed to work with inputs of arbitrary length.

- Usually the **efficiency** or **complexity** of an algorithm is stated as a function relating the input length to the number of steps (**time complexity**) or storage locations.

Performance of a computer

- The performance of a computer is determined by:

- The hardware:
  - processor used (type and speed).
  - memory available (cache and RAM).
  - disk available.

- The programming language in which the algorithm is specified.
- The language compiler/interpreter used.
- The computer operating system software.

Performance of a Program

- The amount of computer **memory** and **time** needed to run a program.

  - Space complexity
    - Why?
    - Because We need to know the amount of memory to be allocated to the program.

  - Time complexity
    - Why?
    - Because We need upper limit on the amount of time needed by the program. (Real-Time systems)

Performance of a Program cont…

- Space Complexity
  - Instruction space (size of the compiled version)
  - Data space (constants, variables, arrays, etc.)
  - Environment stack space (context switching)

- Time Complexity
  - All the factors that space complexity depends on.
  - Compilation time
  - Execution time
    - Operation counts

What is an algorithm and why do we want to analyze one?

- An algorithm is "a step-by-step procedure for accomplishing some end.'' (solve a problem, complete a task, etc.)
- An algorithm can be given or expressed in many ways.
- For example, it can be written down in English (or French, or any other "natural'' language).
- We seek algorithms which are *correct* and *efficient*.
- *Correctness*
  - For any algorithm, we must prove that it *always* returns the desired output for all legal instances of the problem.
- *Efficiency:* Minimum time and minimum resources.

But what can we analyze?

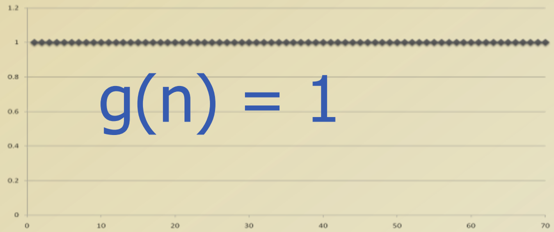
- determine the running time of a program as a function of its inputs.
- determine the total or maximum memory space needed for program data.
- determine the total size of the program code.
- determine whether the program correctly computes the desired result.
- determine the complexity of the program- e.g., how easy is it to read, understand, and modify.
- determine the robustness of the program- e.g., how well does it deal with unexpected or erroneous inputs?
- etc.

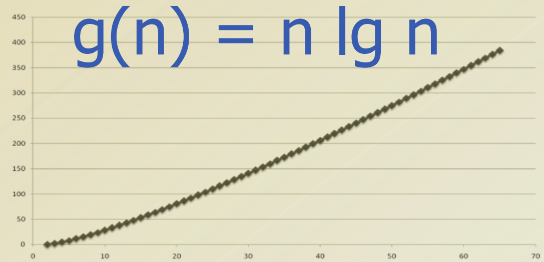
Seven Important Functions

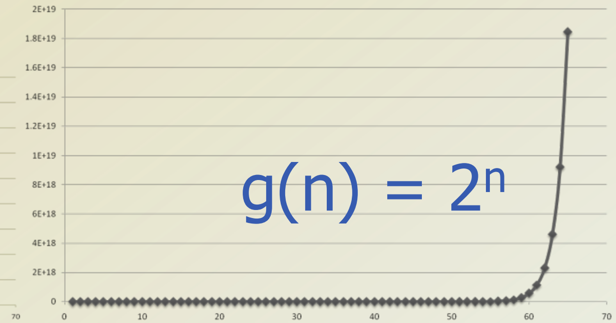
- Seven functions that often appear in algorithm analysis:
  - Constant ≈ $\boldsymbol{1}$
  - Logarithmic ≈ $\log \boldsymbol{n}$
  - Linear ≈ $\boldsymbol{n}$
  - N-Log-N ≈ $\boldsymbol{n} \log \boldsymbol{n}$
  - Quadratic ≈ $\boldsymbol{n}^2$
  - Cubic ≈ $\boldsymbol{n}^3$
  - Exponential ≈ $\mathbf{2}^n$

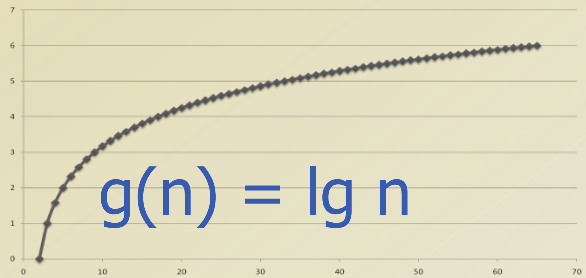
Functions Graphed Using "Normal" Scale

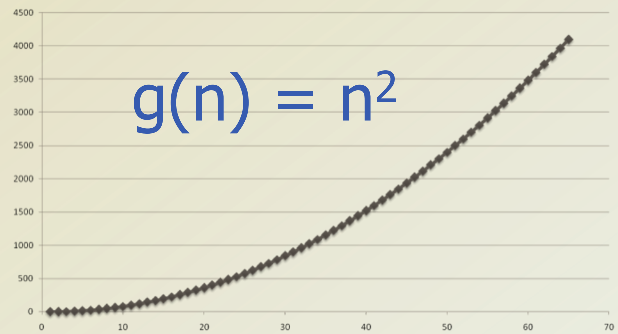
Slide by Matt Stallmann included with permission.

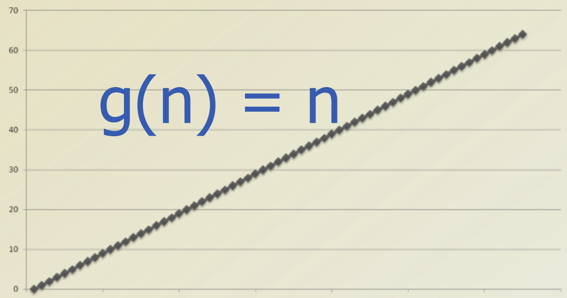


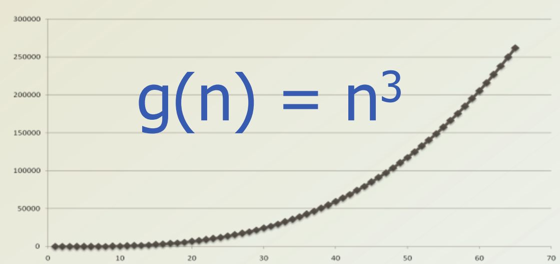












Proposition 4.1 (Logarithm Rules): *Given real numbers $a > 0$, $b > 1$, $c > 0$ and $d > 1$, we have:*

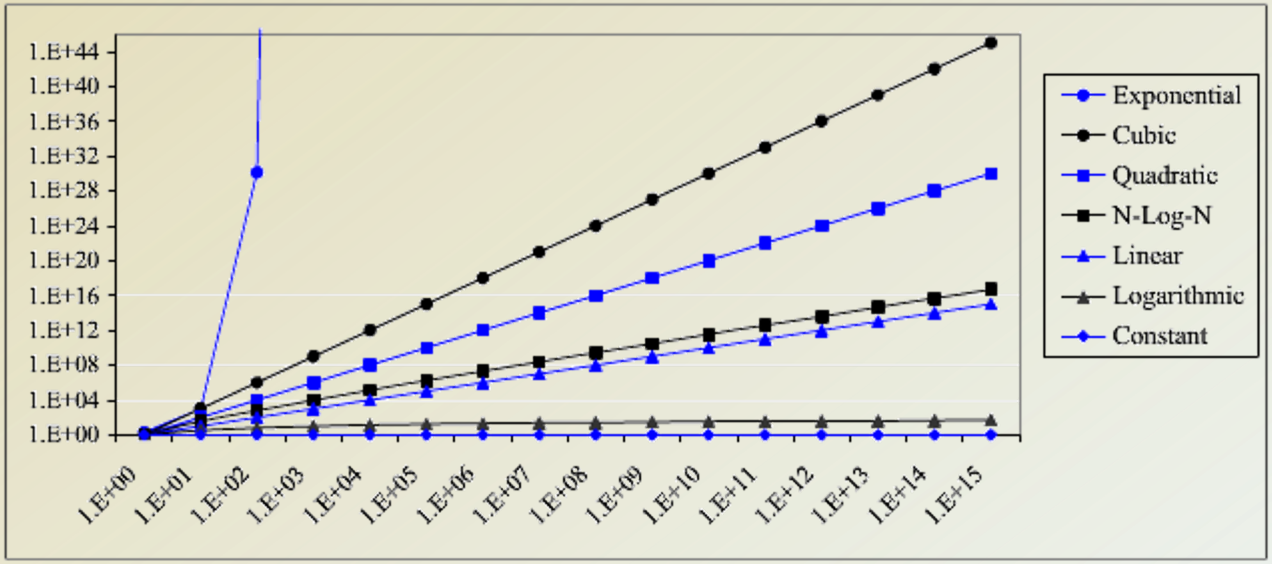
1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \log_b a$
4. $\log_b a = (\log_d a)/\log_d b$
5. $b^{\log_d a} = a^{\log_d b}$.

Example 4.2: *We demonstrate below some interesting applications of the logarithm rules from Proposition 4.1 (using the usual convention that the base of a logarithm is 2 if it is omitted).*

- $\log(2n) = \log 2 + \log n = 1 + \log n$, *by rule 1*
- $\log(n/2) = \log n - \log 2 = \log n - 1$, *by rule 2*
- $\log n^3 = 3 \log n$, *by rule 3*
- $\log 2^n = n \log 2 = n \cdot 1 = n$, *by rule 3*
- $\log_4 n = (\log n)/\log 4 = (\log n)/2$, *by rule 4*
- $2^{\log n} = n^{\log 2} = n^1 = n$, *by rule 5.*

Comparing Growth Rate

constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n



Math you need to Review

- Summations
- Logarithms and Exponents

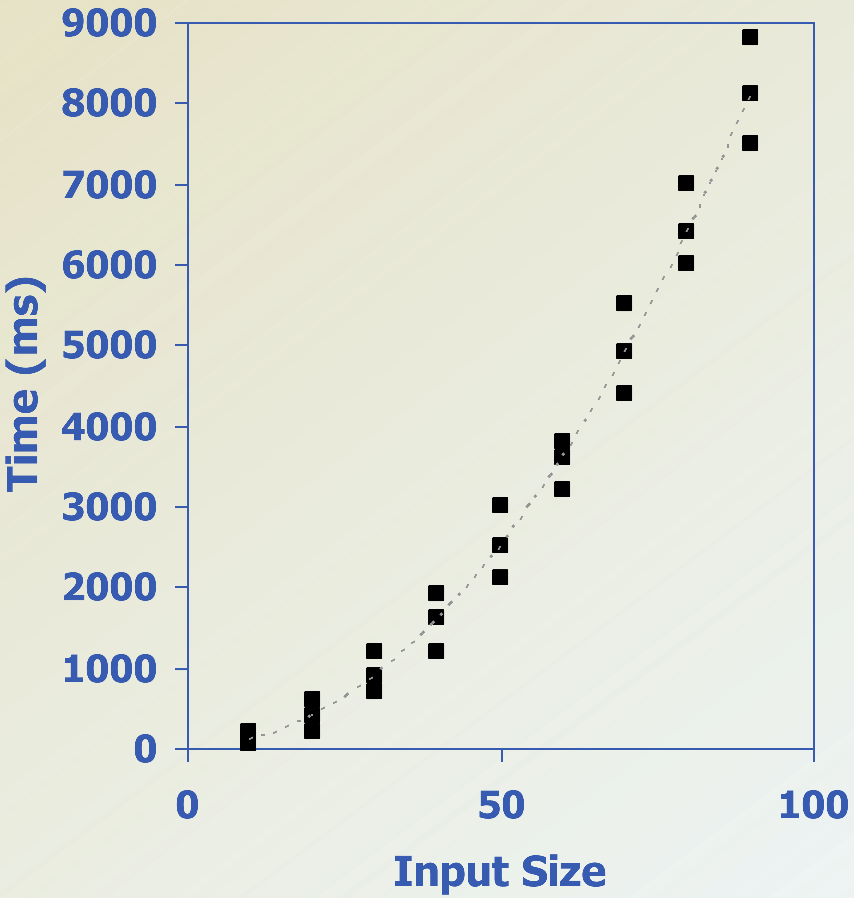
- **properties of logarithms:**
  $\log_b(xy) = \log_b x + \log_b y$
  $\log_b (x/y) = \log_b x - \log_b y$
  $\log_b xa = a\log_b x$
  $\log_b a = \log_x a/\log_x b$

- Proof techniques
- Basic probability

- **properties of exponentials**:
  $a^{(b+c)} = a^b a^{c}$
  $a^{bc} = (a^b)^c$
  $a^b / a^c = a^{(b-c)}$
  $b = a^{\log_a b}$
  $b^c = a^{c*\log_a b}$

Experimental Studies

- Write a program implementing the algorithm.
- Run the program with inputs of varying size and composition.
- Use a method like **System.currentTimeMillis()** to get an accurate measure of the actual running time.
- Plot the results.



Limitations of Experiments

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

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

3. 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.

Pseudocode

- 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 element of an array

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 …

- 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

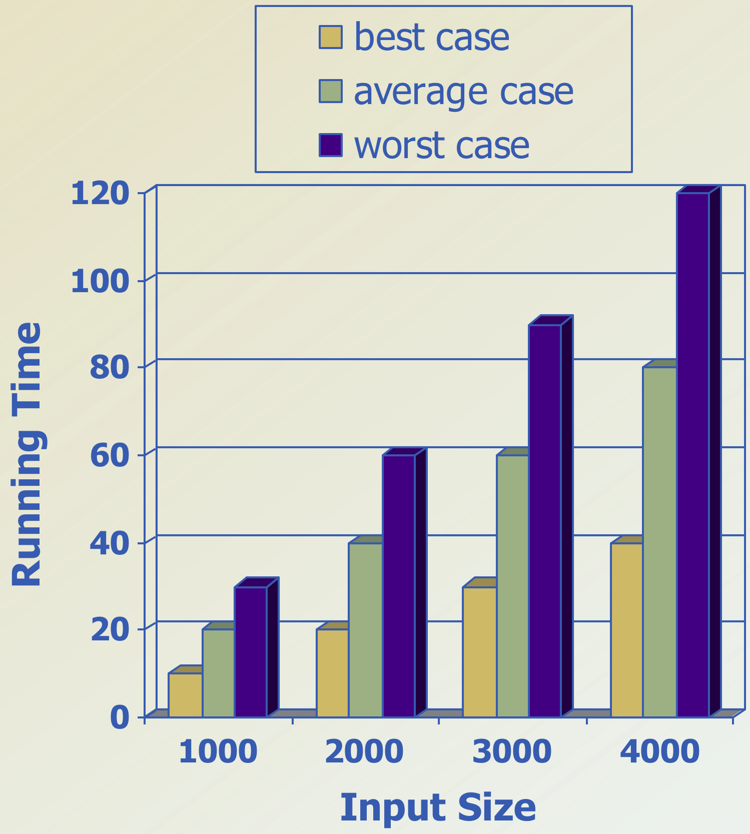
Primitive Operations

- Basic computations performed by an algorithm.
- Identifiable in pseudocode.
- Largely independent from the programming language.
- Exact definition not important.
- Assumed to take a constant amount of time in the RAM model.

- Examples:
  - Evaluating an expression.
  - Assigning a value to a variable.
  - Indexing into an array.
  - Calling a method.
  - Returning from a method.

Running Time

- Most algorithms transform input objects into output objects.
- 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.

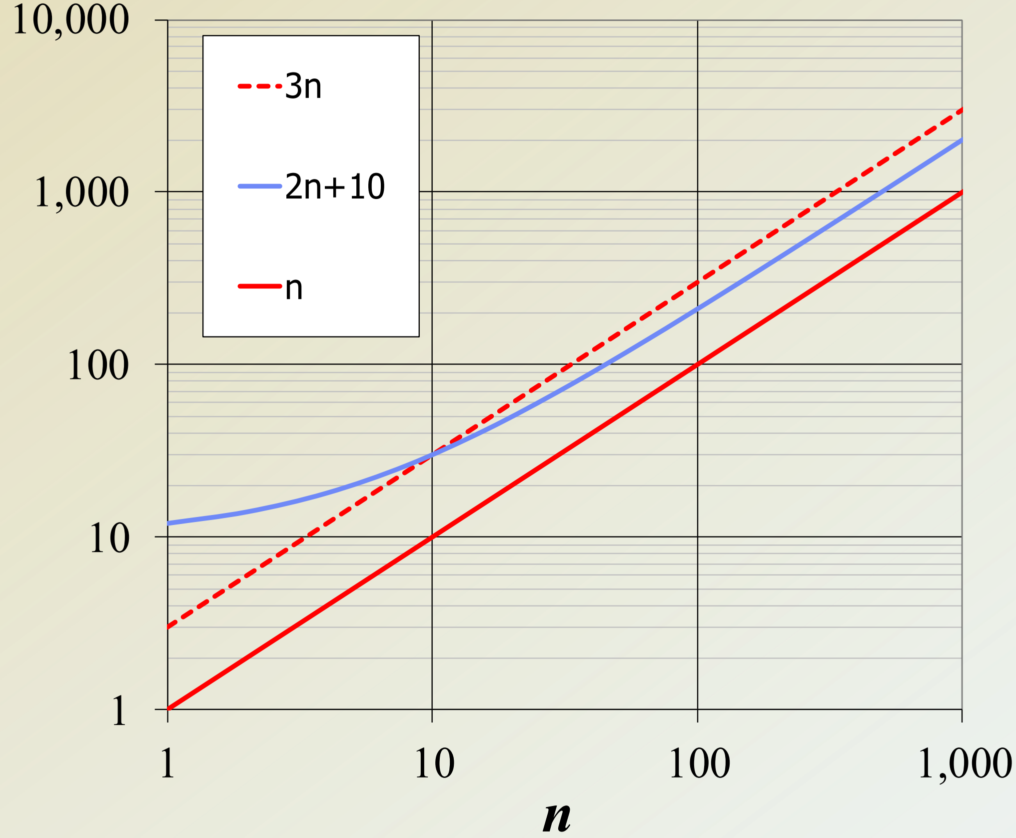


Big-Oh Notation

- 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

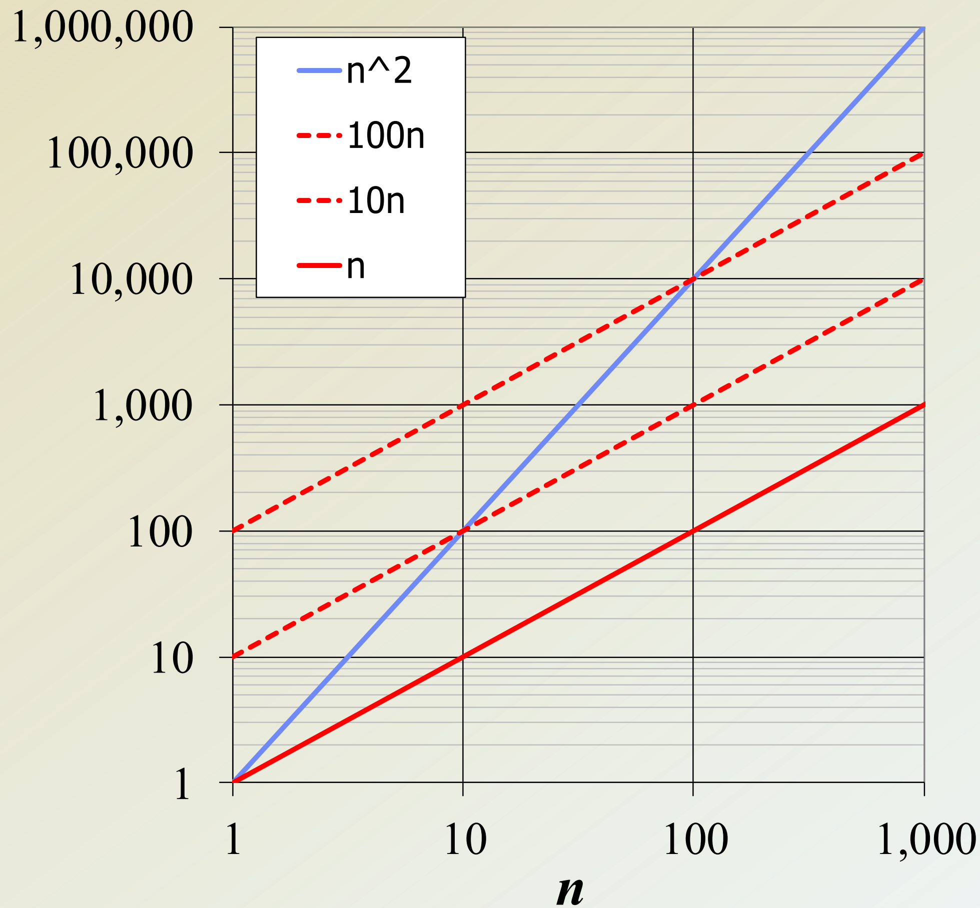
$$f(n) \le cg(n) \quad \text{for } n \ge n_0$$

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



Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \le cn$
  - $n \le c$
  - The above inequality cannot be satisfied since c must be a constant.



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
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2n$
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return *currentMax*	1
Total	$8n - 2$

Estimating Running Time

- Algorithm ***arrayMax*** executes $8\boldsymbol{n} - 2$ primitive operations in the worst case. Define:

  $\boldsymbol{a}$ = Time taken by the fastest primitive operation

  $\boldsymbol{b}$ = Time taken by the slowest primitive operation

- Let $\boldsymbol{T}(\boldsymbol{n})$ be worst-case time of ***arrayMax.*** Then

$$\boldsymbol{a}\,(8\boldsymbol{n} - 2) \le \boldsymbol{T}(\boldsymbol{n}) \le \boldsymbol{b}(8\boldsymbol{n} - 2)$$

- Hence, the running time $\boldsymbol{T}(\boldsymbol{n})$ is bounded by two linear functions.

Big-Oh Rules

- If $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms.
  2. Drop constant factors.
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

More Big-Oh Examples

7n-2

7n-2 is O(n)

need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c\bullet n$ for $n \geq n_0$

this is true for $c = 7$ and $n_0 = 1$

$3n^3 + 20n^2 + 5$

$3n^3 + 20n^2 + 5$ is $O(n^3)$

need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c\bullet n^3$ 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 $3 \log n + 5 \leq c\bullet \log n$ for $n \geq n_0$

this is true for $c = 8$ and $n_0 = 2$

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 determine that algorithm ***arrayMax*** executes at most $8\boldsymbol{n} - 2$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $\boldsymbol{O}(\boldsymbol{n})$ time"

- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations.

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function.
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate.

	$f(n)$ 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

Asymptotic Analysis

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	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

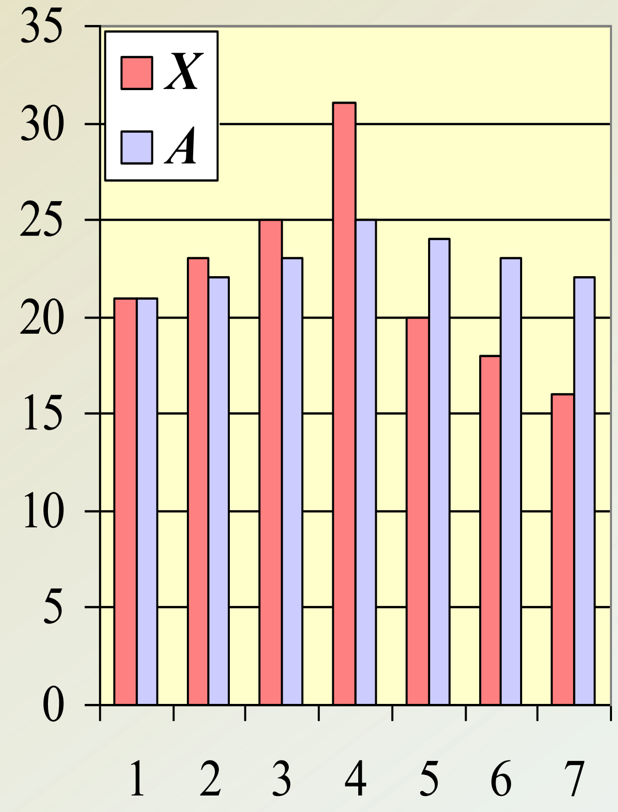
Running Time (μs)	Maximum Problem Size (n) 1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages.
- The $\boldsymbol{i}$-th prefix average of an array $\boldsymbol{X}$ is average of the first $(\boldsymbol{i} + 1)$ elements of $\boldsymbol{X}$:

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

- Computing the array $\boldsymbol{A}$ of prefix averages of another array $\boldsymbol{X}$ has applications to financial analysis.



Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

Algorithm ***prefixAverages1***(X, n)
  Input array X of n integers
  Output array A of prefix averages of X
  $A \leftarrow$ new array of n integers
  for $i \leftarrow 0$ **to** $n - 1$ **do**
    $s \leftarrow X[0]$
    for $j \leftarrow 1$ **to** i **do**
      $s \leftarrow s + X[j]$
    $A[i] \leftarrow s / (i + 1)$
  return A

Arithmetic Progression

- The running time of ***prefixAverages1*** is $\boldsymbol{O}(1 + 2 + \ldots + \boldsymbol{n})$
- The sum of the first $\boldsymbol{n}$ integers is $\boldsymbol{n}(\boldsymbol{n} + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm ***prefixAverages1*** runs in $\boldsymbol{O}(\boldsymbol{n}^2)$ time

Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum

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

Computing Power - Recursive

Algorithm Power(x,n):

Input: A number x and integer n ≥ 0

Output: The value xn

O(log n)

```
if n = 0 then
    return 1
if n is odd then
    y ← Power(x,(n−1)/2)
    return x·y·y
else
    y ← Power(x,n/2)
    return y·y
```

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,n-1) & \text{otherwise.} \end{cases}$$

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n > 0 \text{ is even.} \end{cases}$$

Constant Time Method

```
public static int capacity(int[] arr)
{
      return arr.length; // the capacity of an array
                               is its length
}
```

O(1)

Finding the Maximum in an Array

```
public static int findMax(int[] arr)
{
   int max = arr[0]; // start with the first integer in arr
   for (int i=1; i < arr.length; i++)
       if (max < arr[i])
          max = arr[i];  // update the current maximum
       return max; // the current maximum is now the
                     global maximum
}
```

O(n)

Simple Justification Techiniques

1. **By Example.**
   - Counter Example.
   - $2^i - 1$ is prime !!!
2. **The "Contra" Attack.**
   - Contrapositive.
     - If ***ab*** is even, then ***a*** is even, or ***b*** is even.
   - Contradiction.
     - If ***ab*** is odd, then ***a*** is odd, and ***b*** is odd.
3. **Induction and Loop Invariants.**

End of Chapter 4