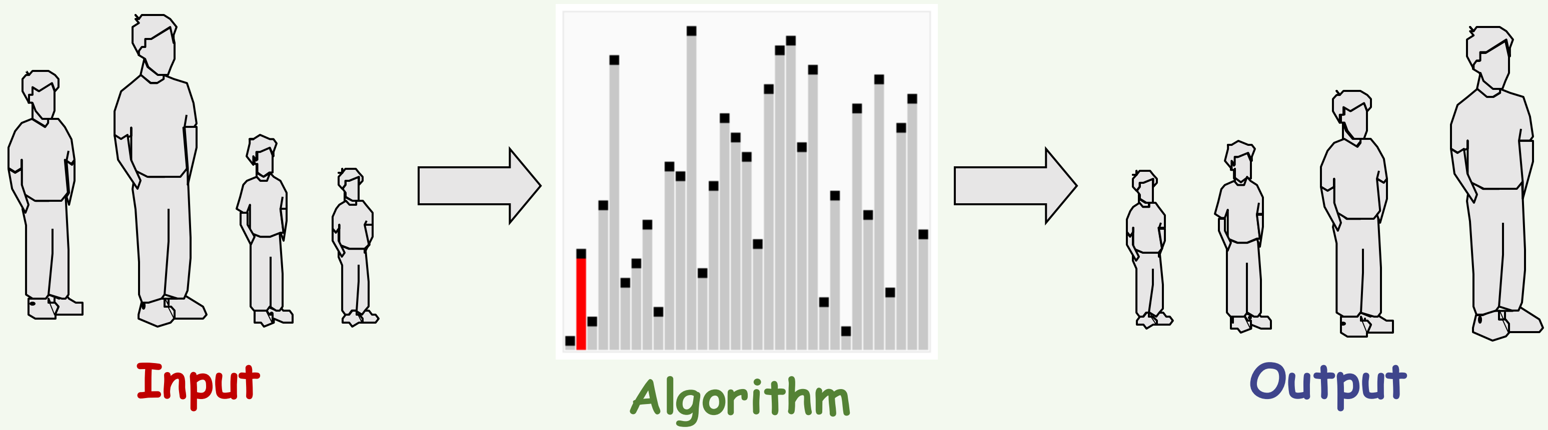
CSCE 2110
Foundations of Data Structures

Analysis of Algorithms

Slides borrowed/adapted from Prof. Yung Li from KAIST

What Is An Algorithm?

- An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.



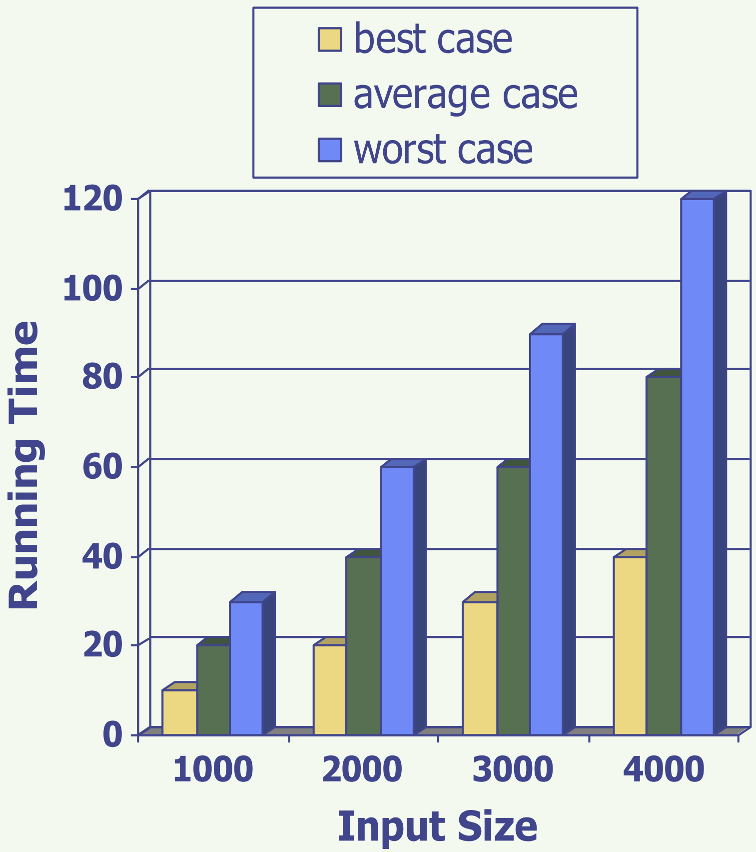
- **Input:** Zero or more quantities (externally produced)
- **Output:** One or more quantities
- **Definiteness:** Clarity, precision of each instruction
- **Finiteness:** The algorithm has to stop after a finite (may be very large) number of steps
- **Effectiveness:** Each instruction has to be basic enough and feasible

What are we going to learn?

- Need to say that some algorithms are "better" than others
- Criteria for evaluation
  - Structure of programs (simplicity, elegance, OO, etc.)
  - Running time
  - Memory space
  - What else???

Running Time

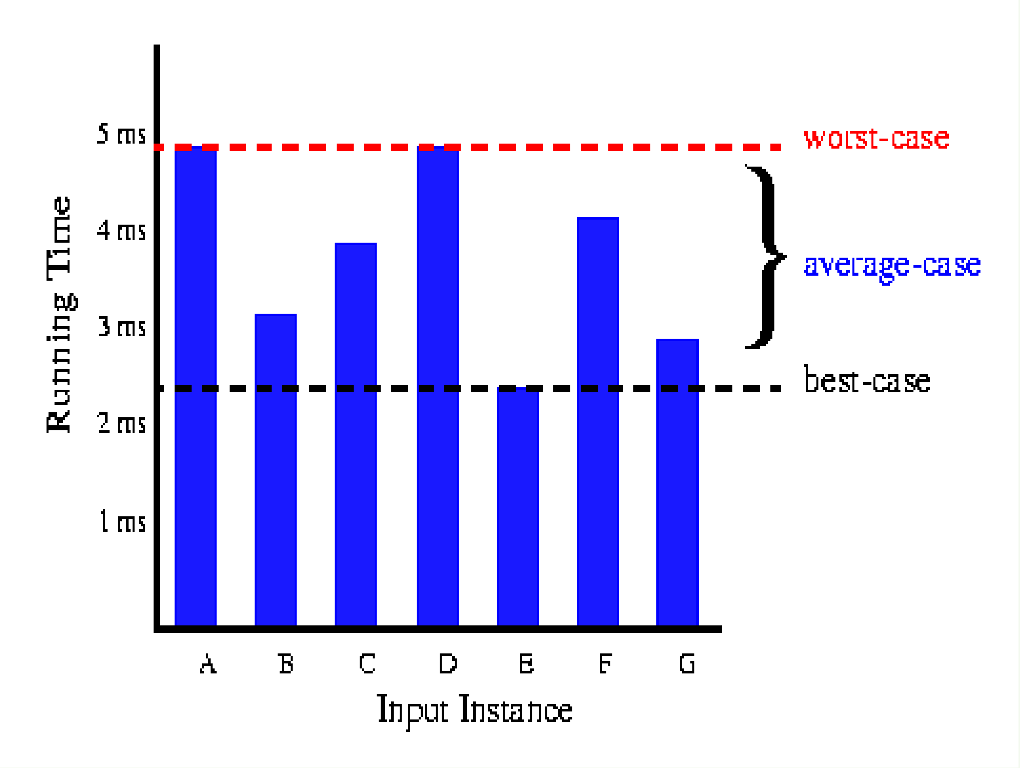
- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average-case running time is often difficult to determine.
    - Why?
- We focus on the worst case running time.
    - Easier to analyze
    - Crucial to applications such as games, finance and robotics



Average Case vs. Worst Case

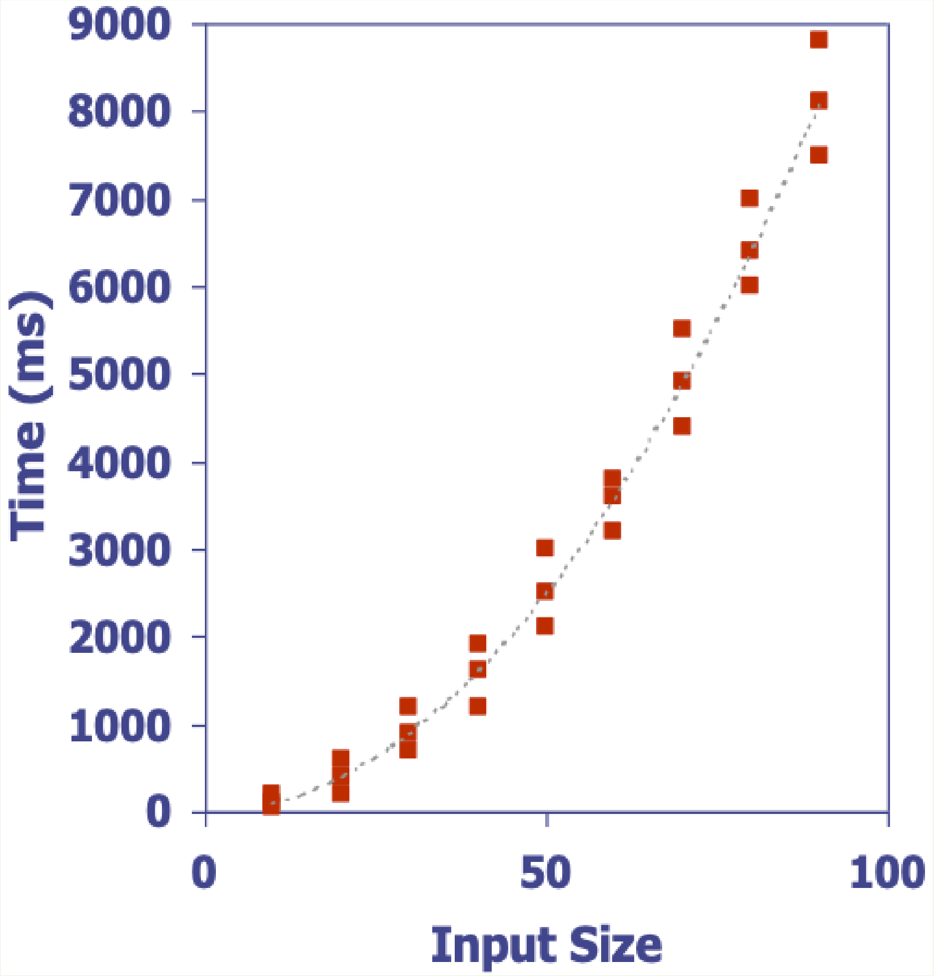
- The average case running time is harder to analyze because you need to know the probability distribution of the input.

- In certain apps (air traffic control, weapon systems, etc.), knowing the worst case time is important.



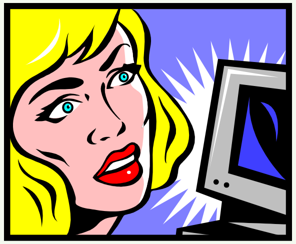
Experimental Approach

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a wall clock to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

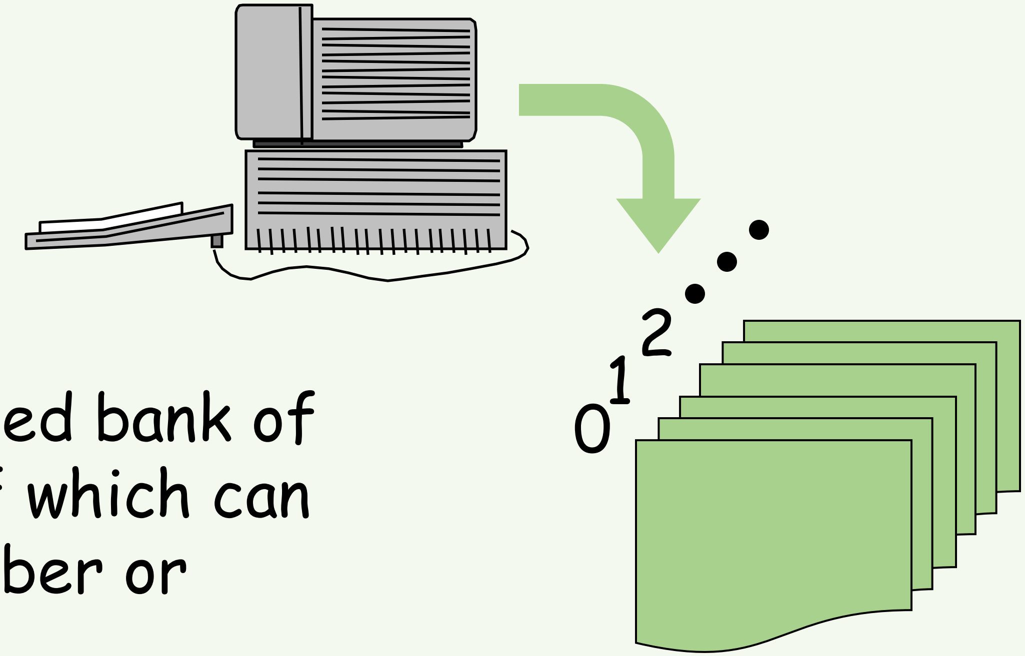
- It is necessary to implement the algorithm, which may be difficult and often time-consuming
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
  - Restrictions



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

The Random Access Machine (RAM) Model



- A **CPU**

- A potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character

- Memory cells are numbered and accessing any cell in memory takes unit time

Pseudocode

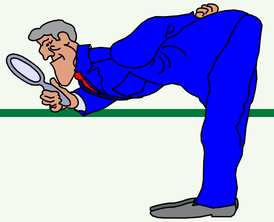
- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find the 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

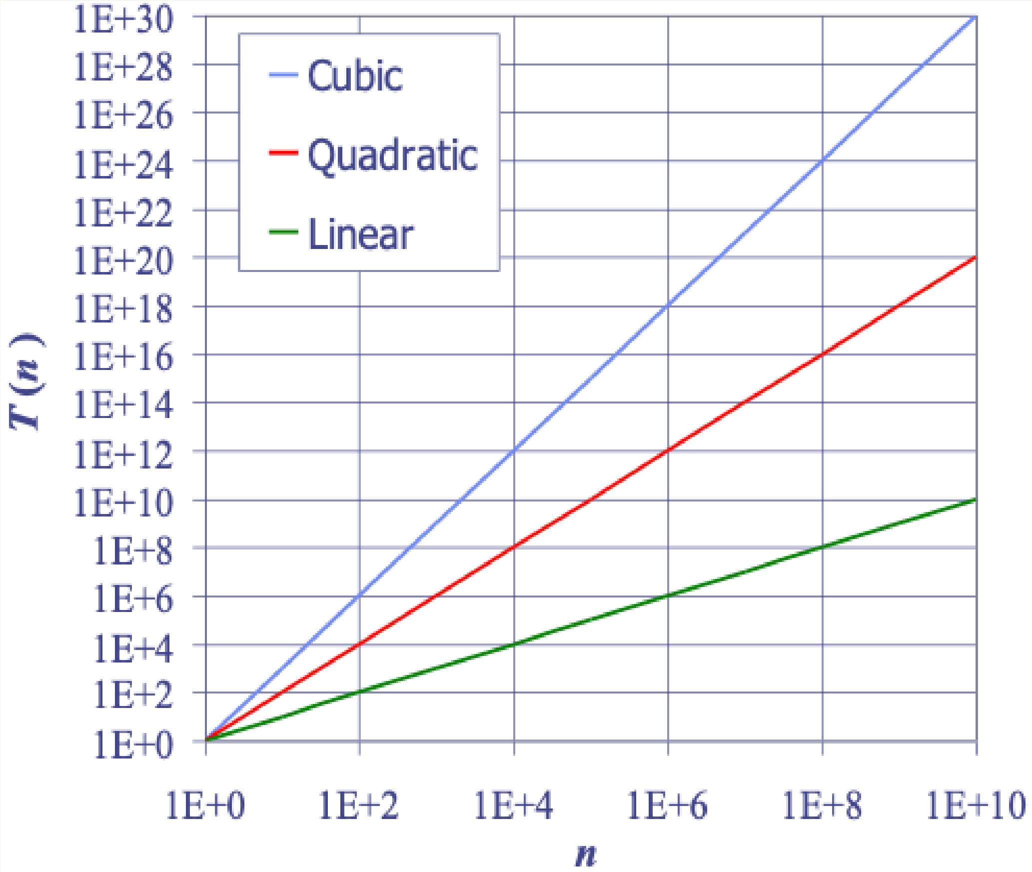
  $\leftarrow$ Assignment (like $=$ in C, C++)

  $=$ Equality testing (like $==$ in C, C++)

  n^2 Superscripts and other mathematical formatting allowed

Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ $\log n$
  - Linear ≈ n
  - N-Log-N ≈ $n \log n$
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n
- In a log-log chart, the slope of the line corresponds to the growth rate of the function



Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- 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

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 $8n - 2$ primitive operations in the worst case. Define:

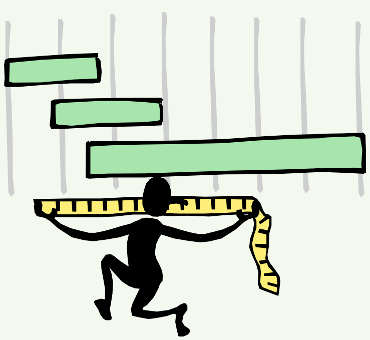
  a = Time taken by the fastest primitive operation

  b = Time taken by the slowest primitive operation

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

$$a\,(8n - 2) \leq T(n) \leq b(8n - 2)$$

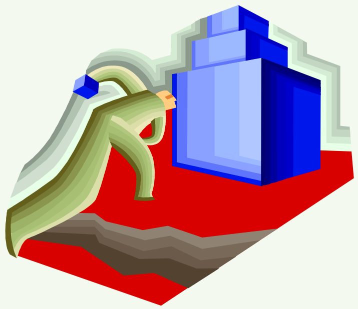
- Hence, the running time $T(n)$ is bounded by two linear functions



Growth Rate of Running Time

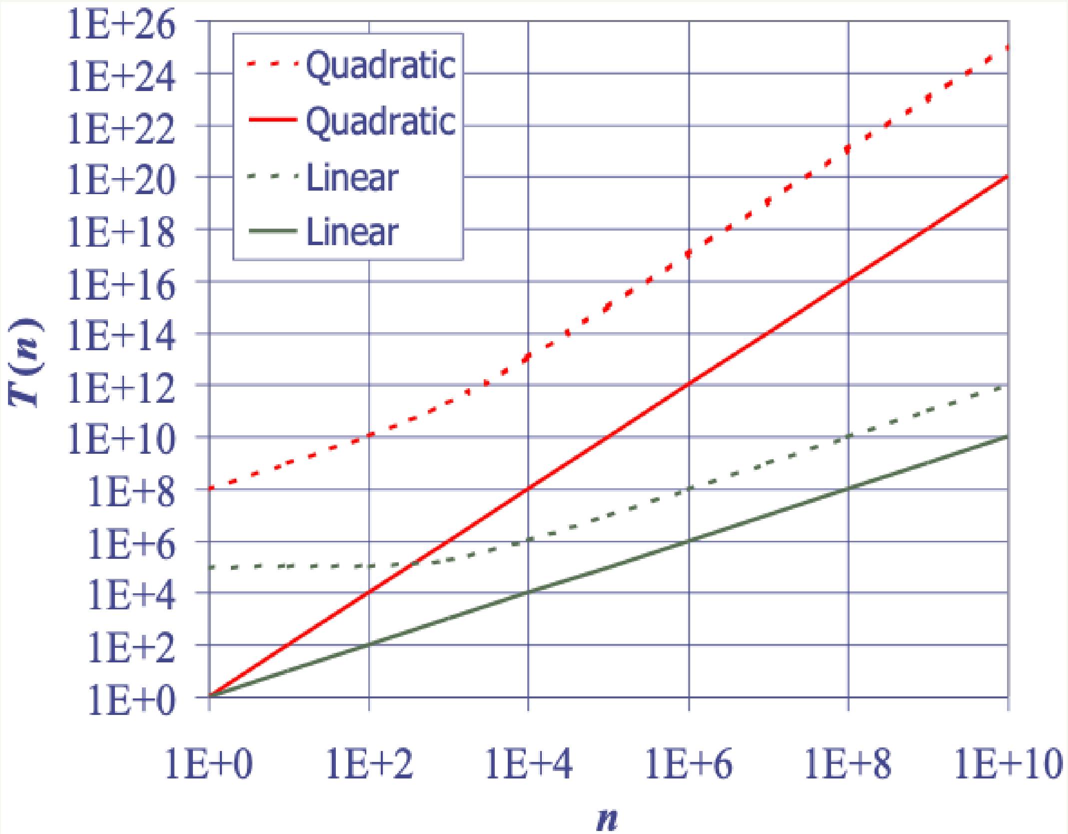
- Changing the hardware/ software environment
    - Affects ***T***(***n***) by a constant factor, but
    - Does not alter the growth rate of ***T***(***n***)

- The linear growth rate of the running time ***T***(***n***) is an intrinsic property of algorithm ***arrayMax***



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
- We consider when $\boldsymbol{n}$ is sufficiently large
  - We call this "Asymptotic Analysis"

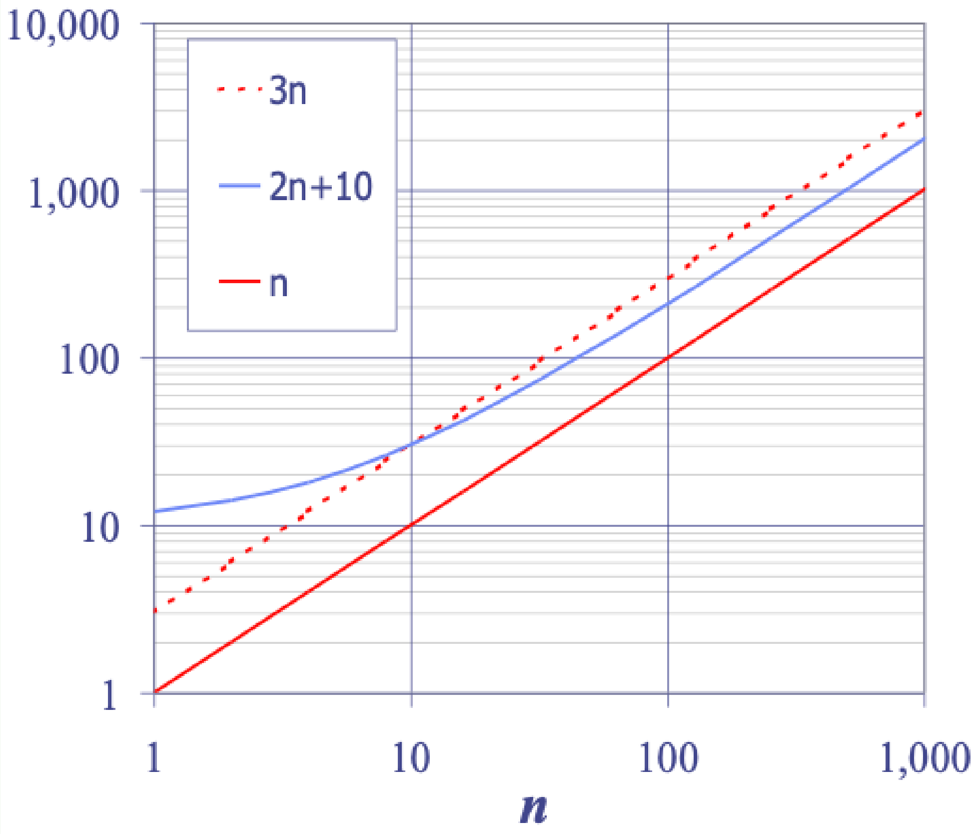


Big-Oh Notation

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

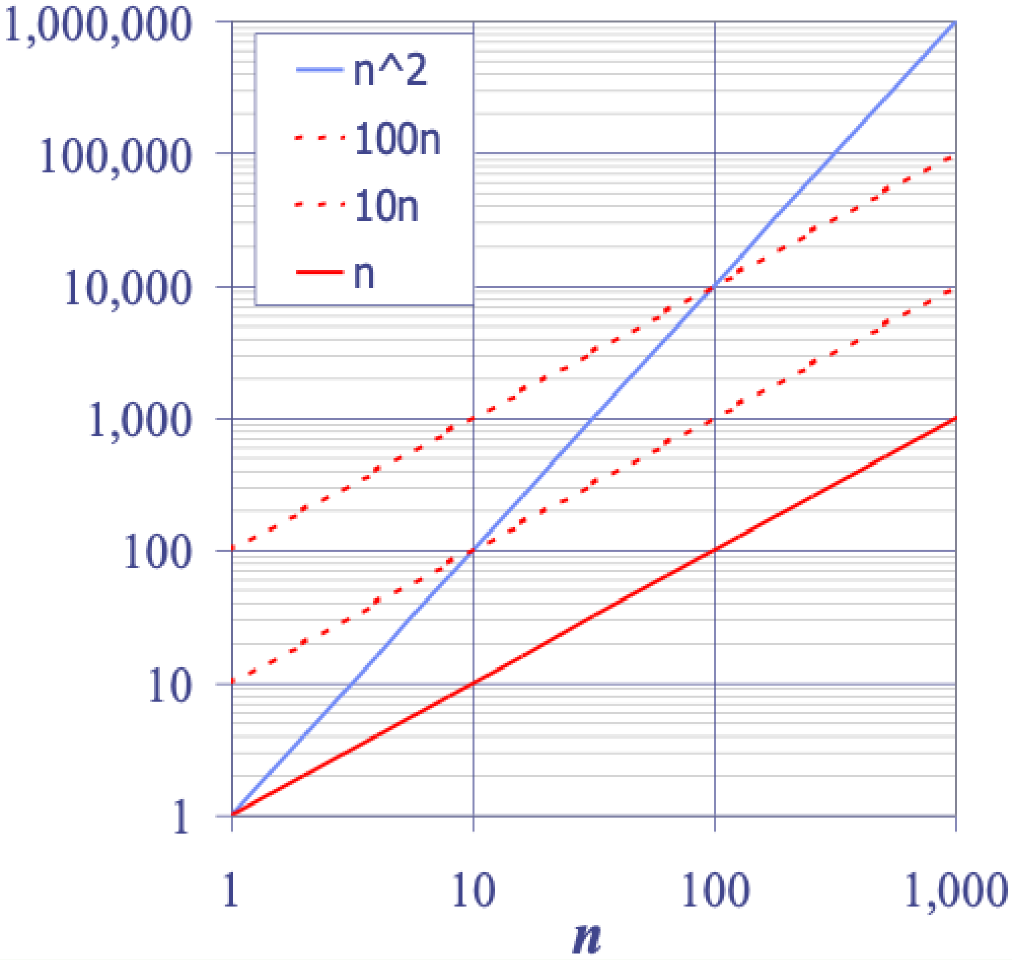
  $f(n) \le cg(n)$ for $n \ge n_0$

- 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 $\mathbf{n}^2$ is not $\mathbf{O(n)}$
  - $\mathbf{n^2 \le cn}$
  - $\mathbf{n \le c}$
  - The above inequality cannot be satisfied since **c** must be a constant



More Big-Oh Examples

7n-2

$7n-2$ is $O(n)$

need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \cdot n$ 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 $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ 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 $3 \log n + 5 \leq c \cdot \log n$ for $n \geq n_0$

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

- (Question) $3 \log n + 5$ is $O(n)$? Yes or No?

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

Which is possible?

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows faster	Yes	No
$f(n)$ grows faster	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

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

- Use the smallest possible class of functions
  - Say "$2\mathbf{n}$ is $\mathbf{O}(\mathbf{n})$" instead of "$2\mathbf{n}$ is $\mathbf{O}(\mathbf{n}^2)$"

- Use the simplest expression of the class
  - Say "$3\mathbf{n} + 5$ is $\mathbf{O}(\mathbf{n})$" instead of "$3\mathbf{n} + 5$ is $\mathbf{O}(3\mathbf{n})$"

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 $8n - 2$ primitive operations
  - 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

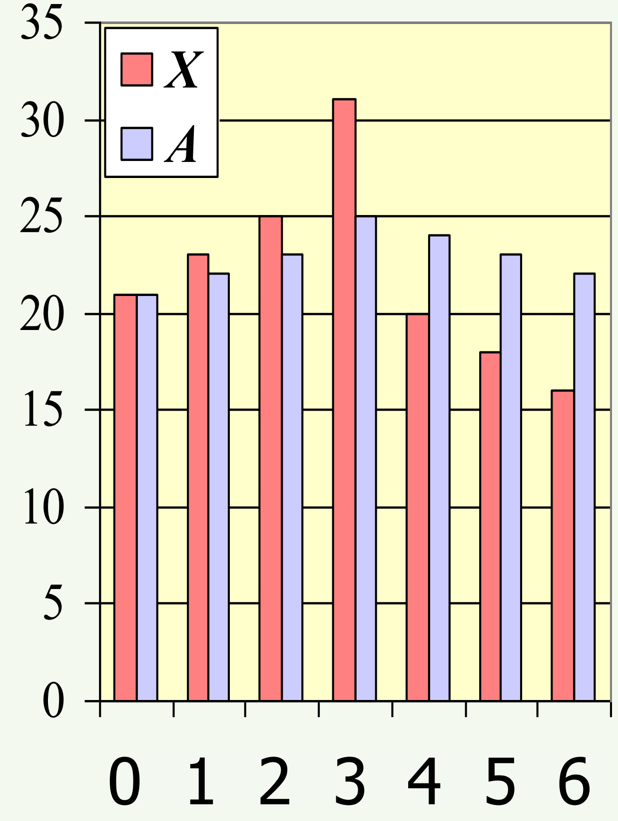
Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages

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

  A[**i**] = (**X**[0] + **X**[1] + … + **X**[**i**])/(i+1)

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



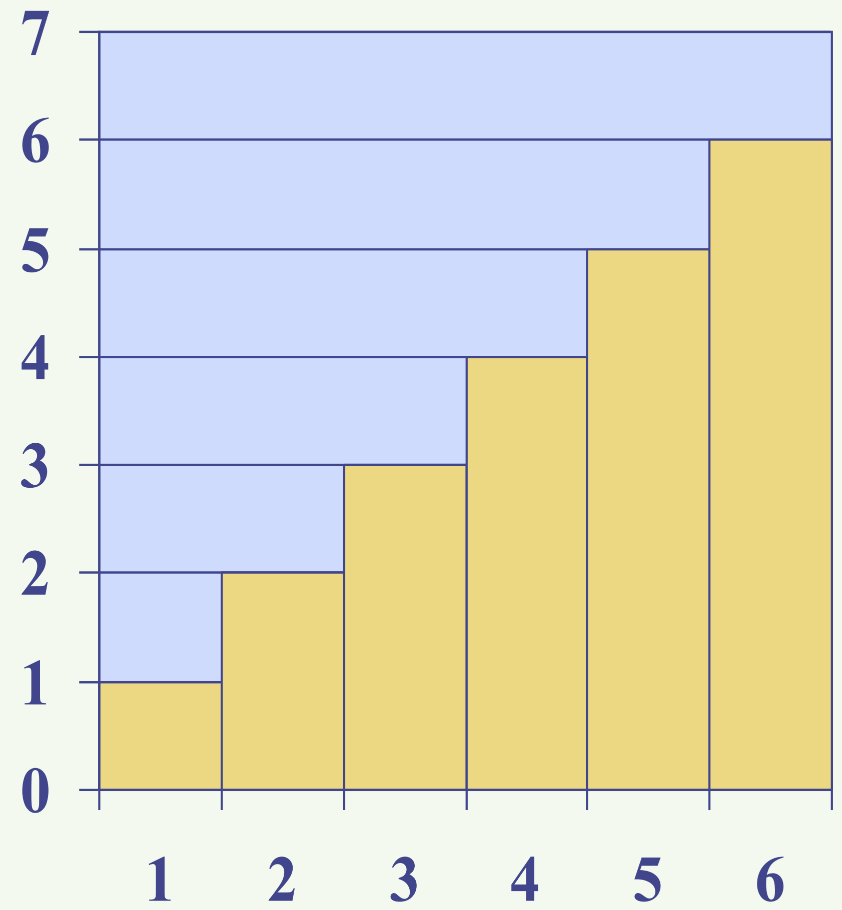
Prefix Averages (Quadratic)

- 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        #operations
     A ← new array of n integers                   n
    for i ← 0 to n − 1 do                          n
     s ← X[0]                                      n
     for j ← 1 to i do                             1 + 2 + …+ (n − 1)
        s ← s + X[j]                               1 + 2 + …+ (n − 1)
     A[i] ← s / (i + 1)                            n
    return A                                       1
```

Arithmetic Progression

- The running time of **prefixAverages1** is $O(1 + 2 + \ldots + n)$
- The sum of the first **n** integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm **prefixAverages1** runs in $O(n^2)$ 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
    #operations
    A ← new array of n integers                n
    s ← 0                                      1
    for i ← 0 to n − 1 do                      n
        s ← s + X[i]                           n
        A[i] ← s / (i + 1)                     n
    return A                                   1
```

- Algorithm prefixAverages2 runs in $O(n)$ time

Another Example

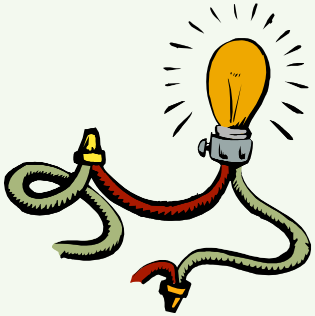
```
Result ← 0; m ← 1;
for I ← 1 to n
    m ← m*2;
    for j ← 1 to m do
        result ← result + i*m*j
```

Relatives of Big-Oh

- 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 $f(n) \geq c \cdot g(n)$ 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 $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$

Intuition for Asymptotic Notation



- Big-Oh
  - 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)

Examples (1)

$5n^2$ 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 $f(n) \geq c \cdot g(n)$ 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 $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

let $c = 1$ and $n_0 = 1$

Examples (2)

$5n^2$ 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 (for $O(n^2)$)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 $f(n) \leq c \cdot g(n)$ for $n \geq n_0$

Let $c = 5$ and $n_0 = 1$

Examples (3)

- Practice on your own: find c and n_0
  - $2n^2 = O(n^3)$
  - $n = O(n^2)$
  - $\frac{n}{1000} = O(n^2)$
  - $n^{1.999} = O(n^2)$
  - $n^2 + n = O(n^2)$
  - $n^2 + 1000n = O(n^2)$
  - $1000n^2 + 1000n = O(n^2)$