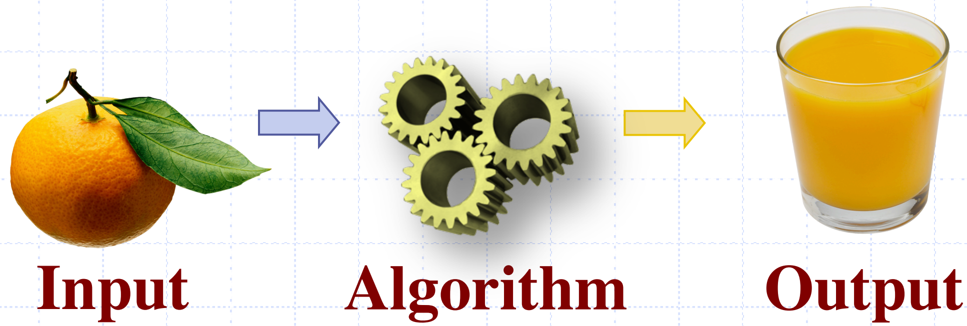
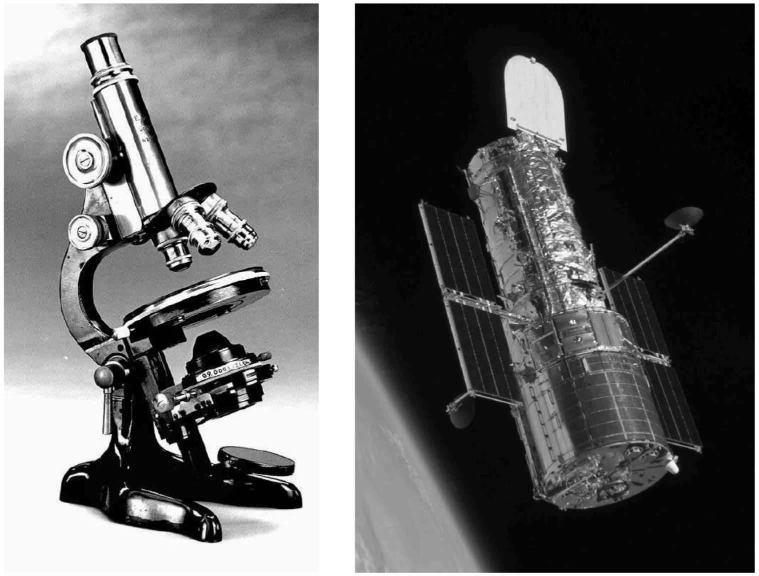
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Analysis of Algorithms



Scalability

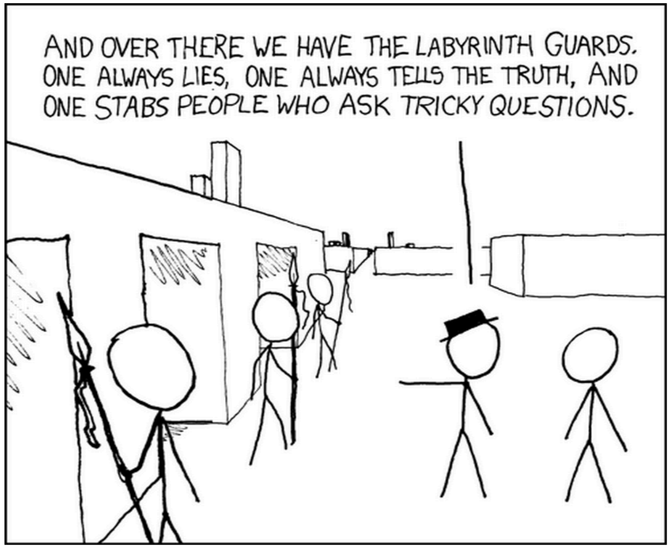
- Scientists often have to deal with differences in scale, from the microscopically small to the astronomically large.
- Computer scientists must also deal with scale, but they deal with it primarily in terms of data volume rather than physical object size.
- **Scalability** refers to the ability of a system to gracefully accommodate growing sizes of inputs or amounts of workload.



Microscope: U.S. government image, from the N.I.H. Medical Instrument Gallery, DeWitt Stetten, Jr., Museum of Medical Research. Hubble Space Telescope: U.S. government image, from NASA, STS-125 Crew, May 25, 2009.

Application: Job Interviews

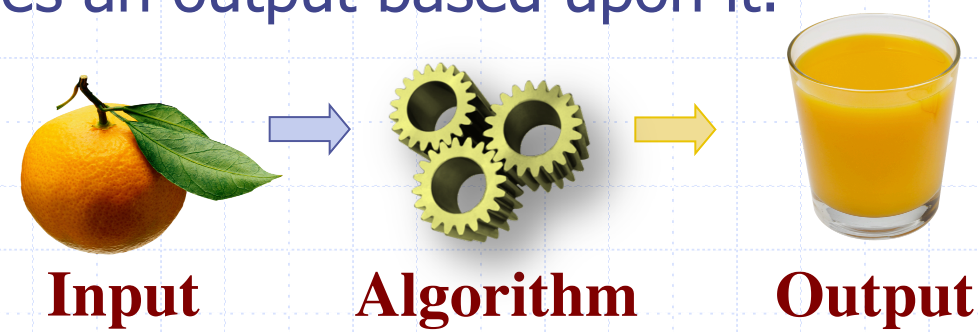
- High technology companies tend to ask questions about **algorithms and data structures** during job interviews.
- Algorithms questions can be short but often require critical thinking, creative insights, and subject knowledge.
  - All the "Applications" exercises in Chapter 1 of the Goodrich-Tamassia textbook are taken from reports of actual job interview questions.



xkcd "Labyrinth Puzzle." http://xkcd.com/246/

Algorithms and Data Structures

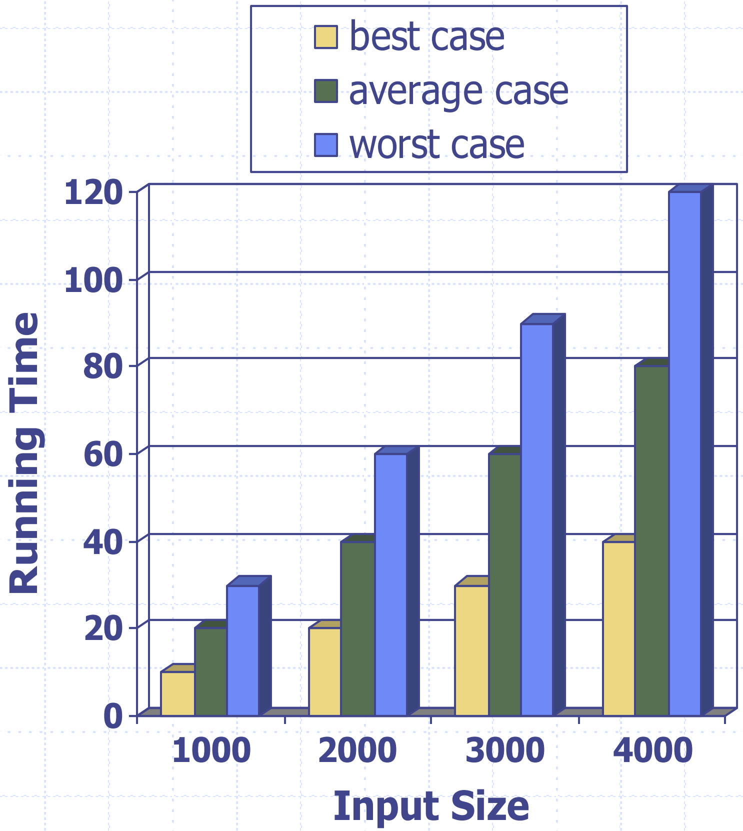
- An **algorithm** is a step-by-step procedure for performing some task in a finite amount of time.
  - Typically, an algorithm takes input data and produces an output based upon it.



- A **data structure** is a systematic way of organizing and accessing data.

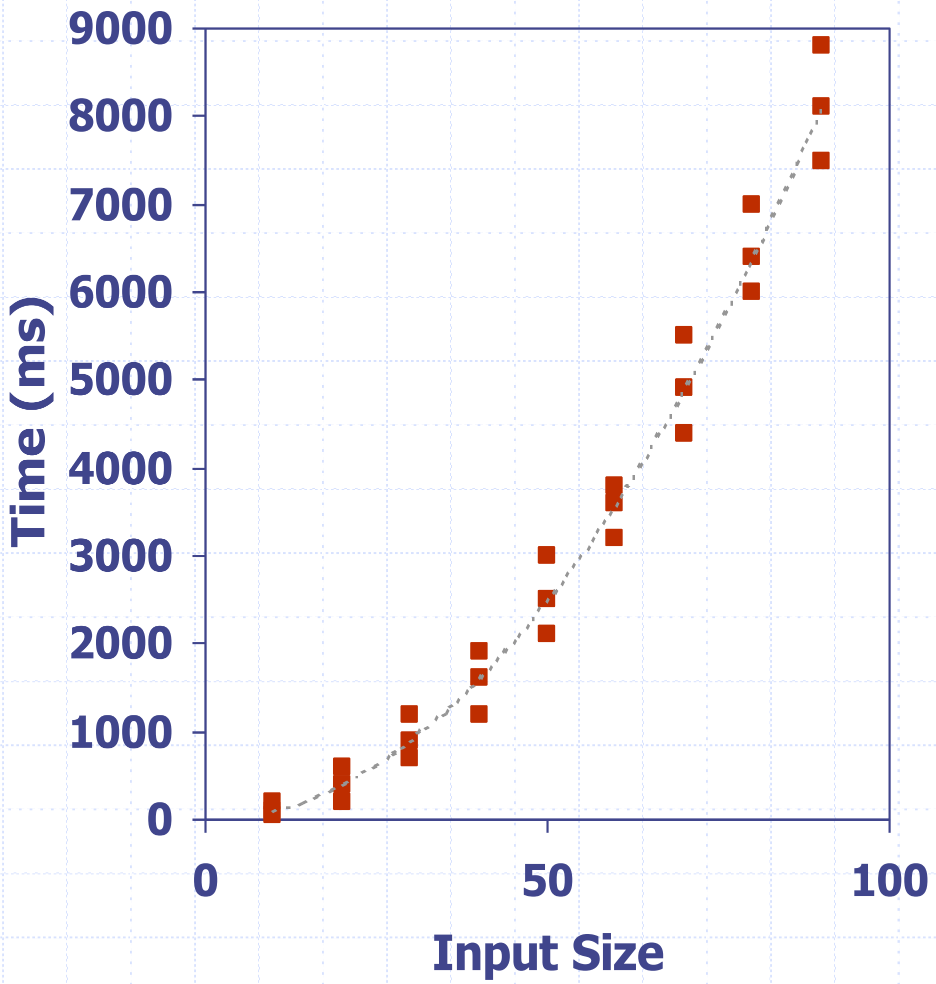
Running Time

- 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 primarily on the **worst case running time**.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



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



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

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

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

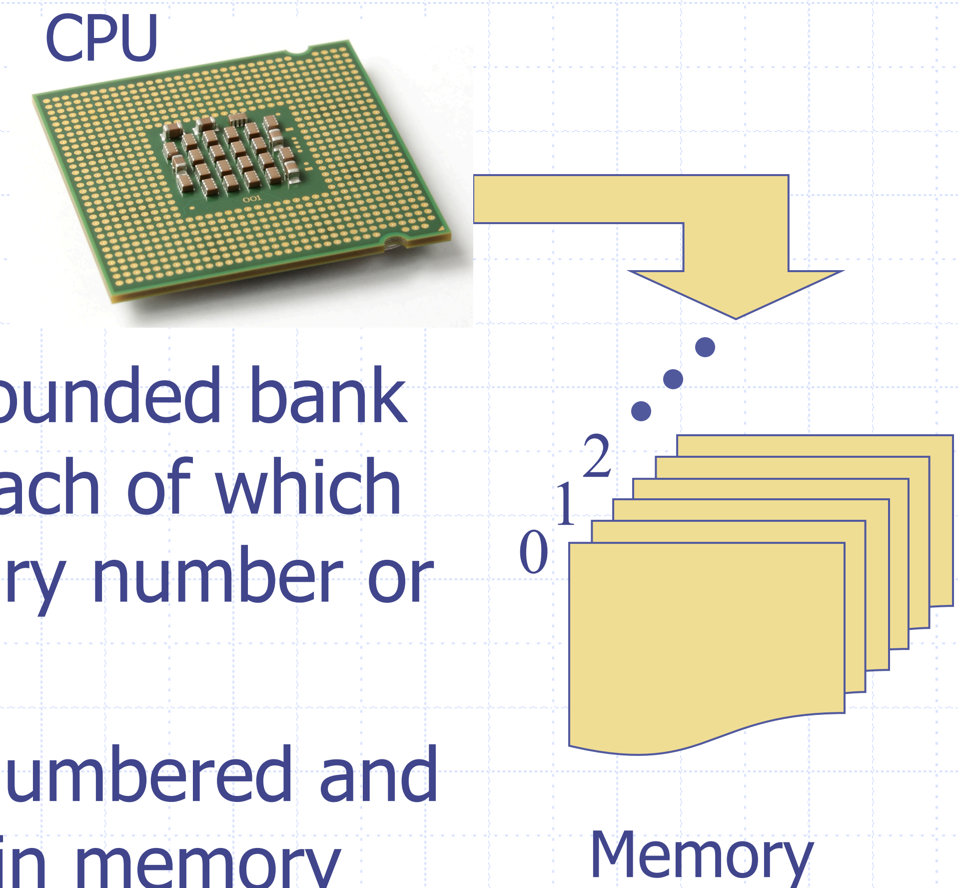
  $=$ Equality testing

  n^2 Superscripts and other mathematical formatting allowed

The Random Access Machine (RAM) Model

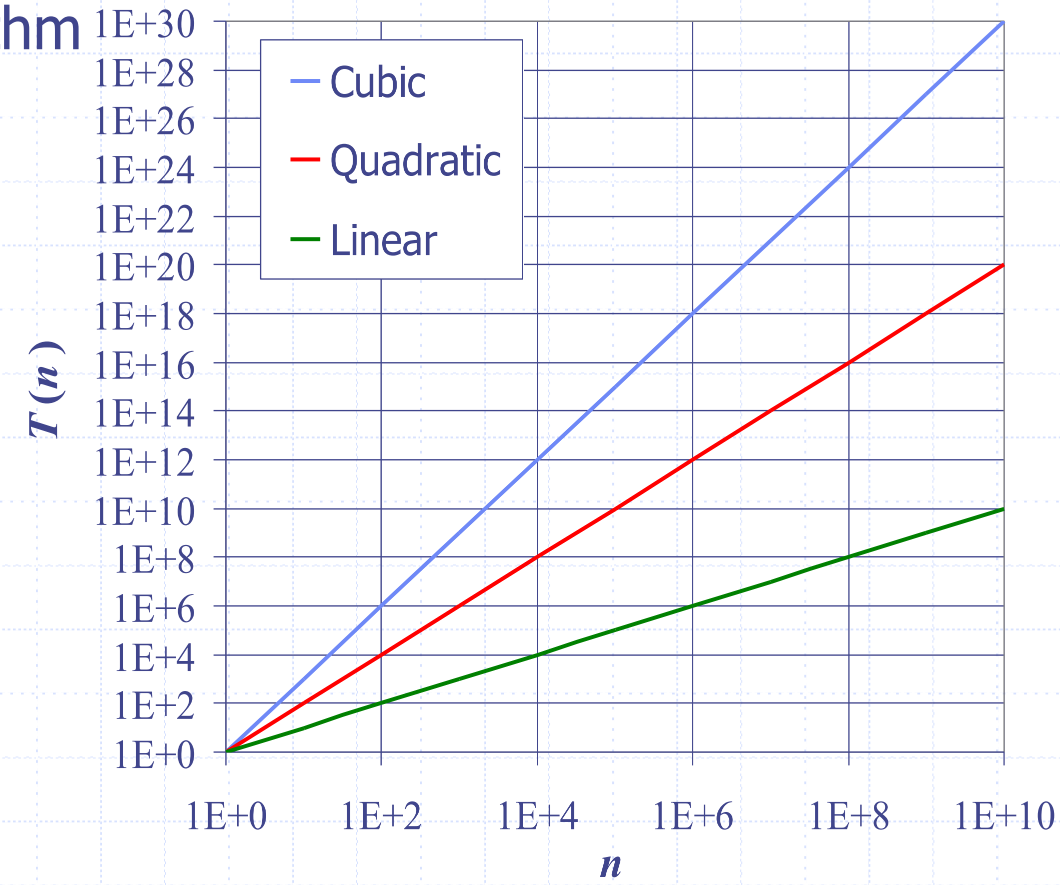
A **RAM** consists of

- A CPU
- An 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



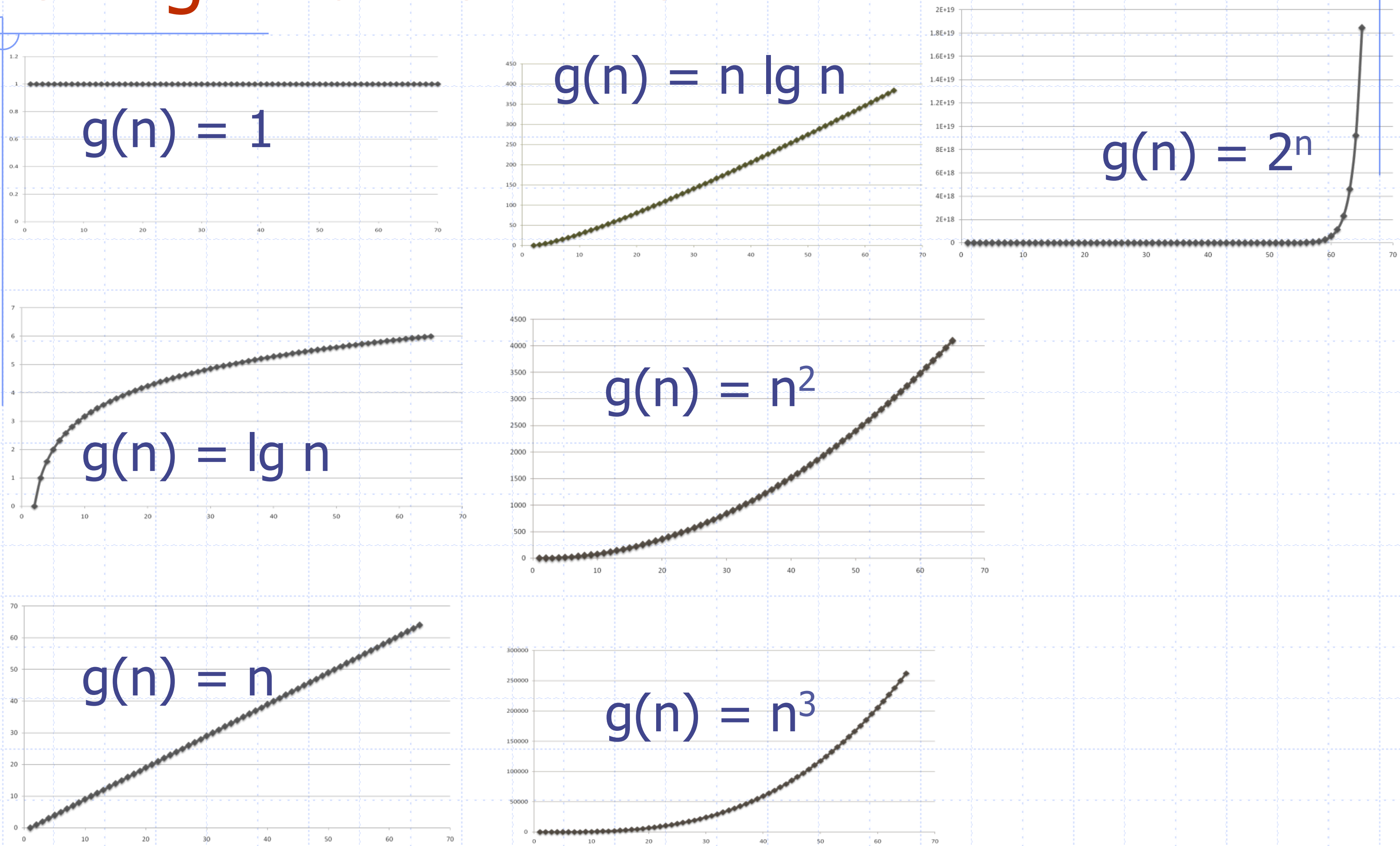
Seven Important Functions

- 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



Functions Graphed Using “Normal” Scale

Slide by Matt Stallmann included with permission.



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

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

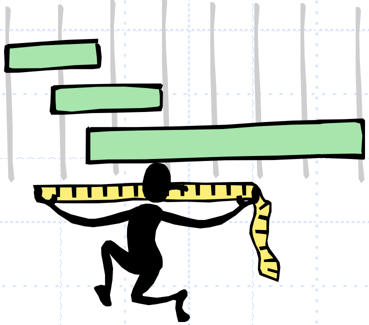
Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currentMax \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if $currentMax < A[i]$ **then**
    $currentMax \leftarrow A[i]$
return $currentMax$

Estimating Running Time



- Algorithm arrayMax executes $7n - 2$ primitive operations in the worst case, $5n$ in the best case. Define:

  a = Time taken by the fastest primitive operation

  b = Time taken by the slowest primitive operation

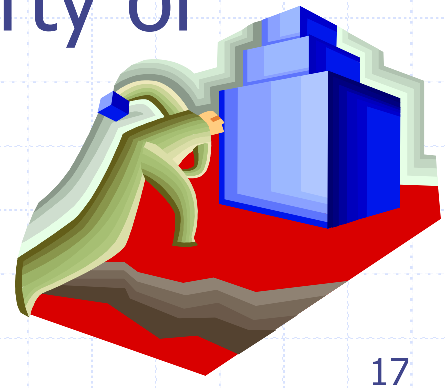
- Let $T(n)$ be worst-case time of arrayMax. Then

$$a(5n) \leq T(n) \leq b(7n - 2)$$

- 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



Slide by Matt Stallmann included with permission.

Why Growth Rate Matters

if runtime is...	time for n + 1	time for 2 n	time for 4 n
$c \lg n$	$c \lg (n + 1)$	$c (\lg n + 1)$	$c(\lg n + 2)$
$c n$	$c (n + 1)$	$2c n$	$4c n$
$c n \lg n$	$\sim c n \lg n + c n$	$2c n \lg n + 2cn$	$4c n \lg n + 4cn$
$c n^2$	$\sim c n^2 + 2c n$	**$4c n^2$**	$16c n^2$
$c n^3$	$\sim c n^3 + 3c n^2$	$8c n^3$	$64c n^3$
$c 2^n$	$c 2^{n+1}$	$c 2^{2n}$	$c 2^{4n}$

runtime quadruples when problem size doubles

Analyzing Recursive Algorithms

- Use a function, T(*n*), to derive a **recurrence relation** that characterizes the running time of the algorithm in terms of smaller values of *n*.

Algorithm recursiveMax(A, n):

  Input: An array A storing $n \geq 1$ integers.

  Output: The maximum element in A.

 if $n = 1$ **then**

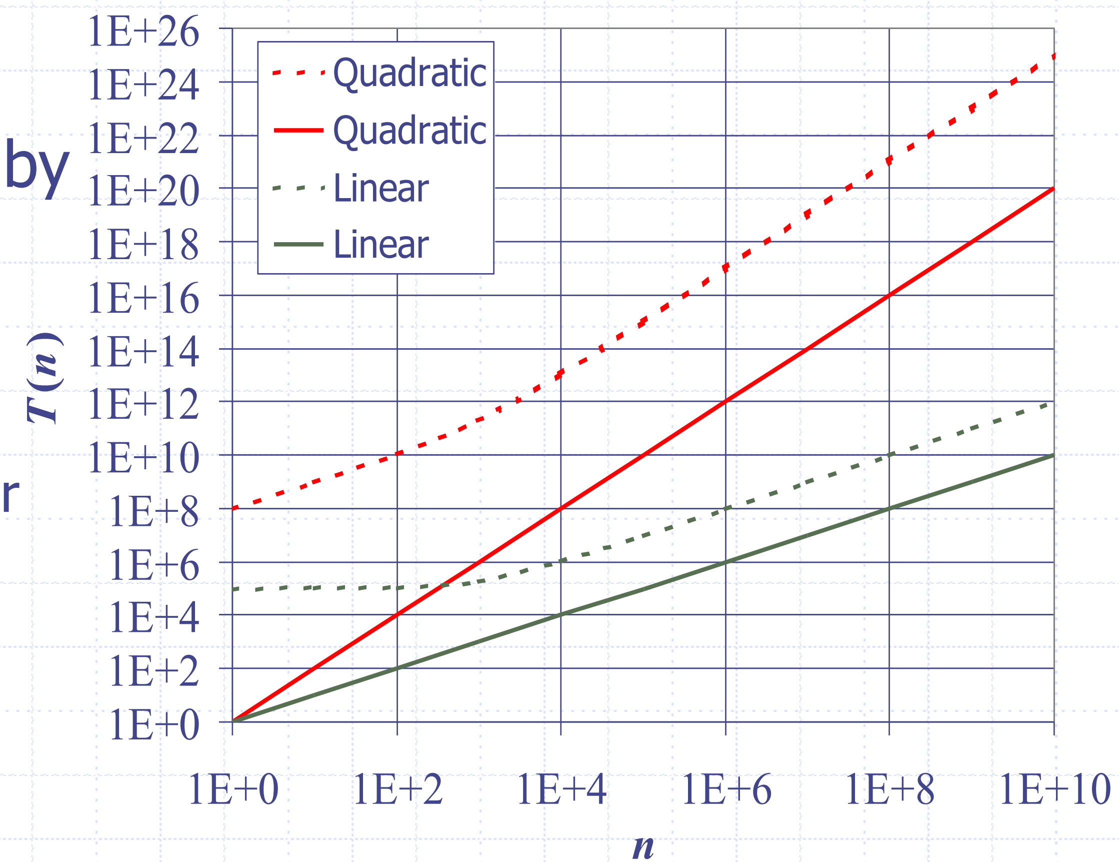
  return $A[0]$

 return $\max\{\text{recursiveMax}(A, n-1),\ A[n-1]\}$

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1) + 7 & \text{otherwise,} \end{cases}$$

Constant Factors

- The growth rate is minimally affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function

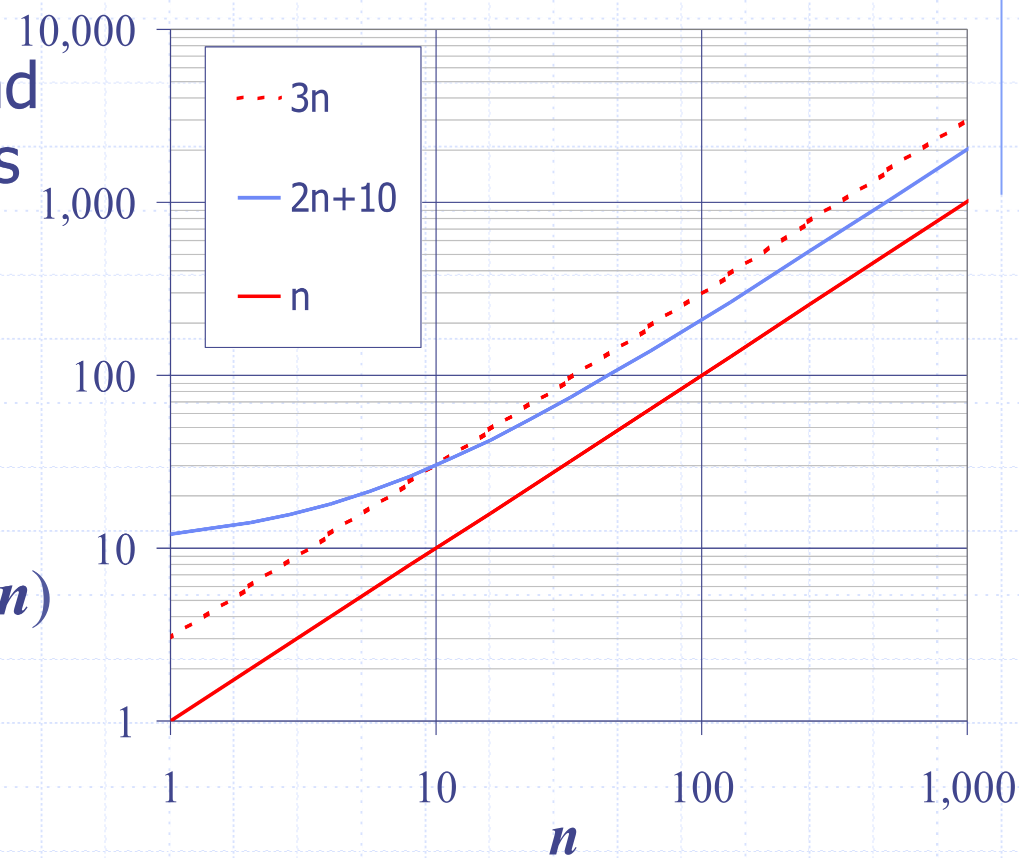


Big-Oh Notation

- 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

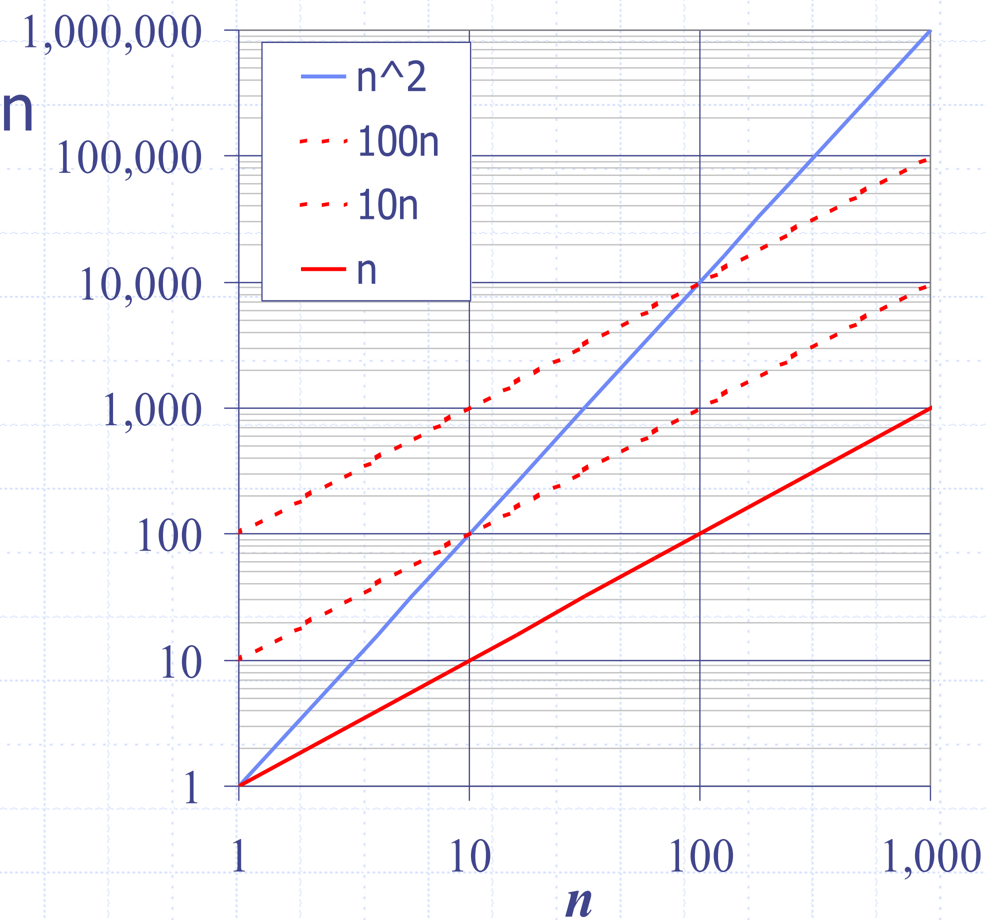
  $f(n) \le cg(n)$ for $n \ge n_0$

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

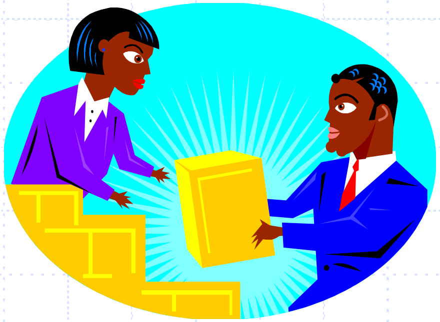


Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - 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 $7n - 2 \leq cn$ 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 $3n^3 + 20n^2 + 5 \leq cn^3$ 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 $n_0 = 2$

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

Big-Oh Rules

- If is $f(n)$ 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 $O(3n)$"

Asymptotic Algorithm Analysis

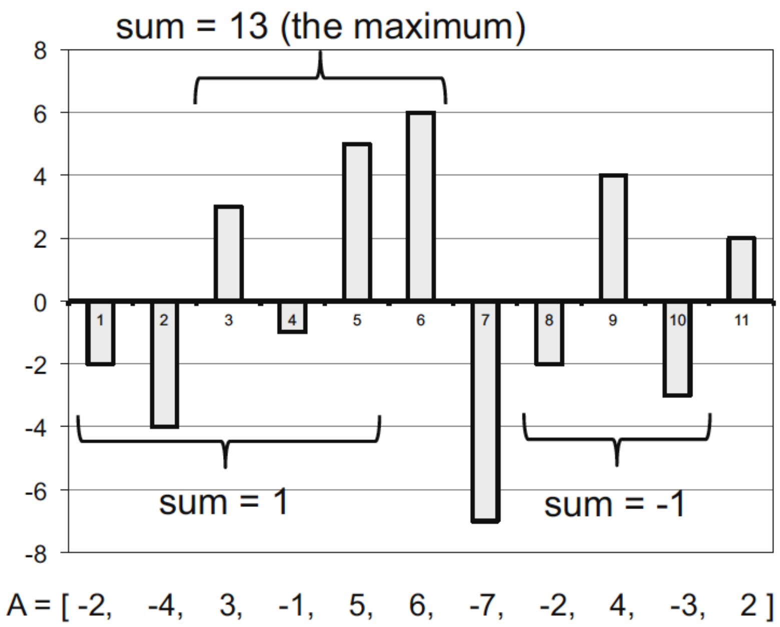
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We say that 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

A Case Study in Algorithm Analysis

- Given an array of *n* integers, find the subarray, A[j:k] that maximizes the sum

$$s_{j,k} = a_j + a_{j+1} + \cdots + a_k = \sum_{i=j}^{k} a_i.$$

- In addition to being an interview question for testing the thinking skills of job candidates, this **maximum subarray problem** also has applications in pattern analysis in digitized images.



A First (Slow) Solution

Compute the maximum of every possible subarray summation of the array *A* separately.

Algorithm MaxsubSlow(A):
  Input: An n-element array A of numbers, indexed from 1 to n.
  Output: The maximum subarray sum of array A.

```
m ← 0      // the maximum found so far
for j ← 1 to n do
    for k ← j to n do
        s ← 0      // the next partial sum we are computing
        for i ← j to k do
            s ← s + A[i]
        if s > m then
            m ← s
return m
```

- The outer loop, for index j, will iterate n times, its inner loop, for index k, will iterate at most n times, and the inner-most loop, for index i, will iterate at most n times.
- Thus, the running time of the MaxsubSlow algorithm is $O(n^3)$.

An Improved Algorithm

- A more efficient way to calculate these summations is to consider **prefix sums**

$$S_t = a_1 + a_2 + \cdots + a_t = \sum_{i=1}^{t} a_i$$

- If we are given all such prefix sums (and assuming $S_0=0$), we can compute any summation $s_{j,k}$ in constant time as

$$s_{j,k} = S_k - S_{j-1}$$

An Improved Algorithm, cont.

- Compute all the prefix sums
- Then compute all the subarray sums

Algorithm MaxsubFaster(A):

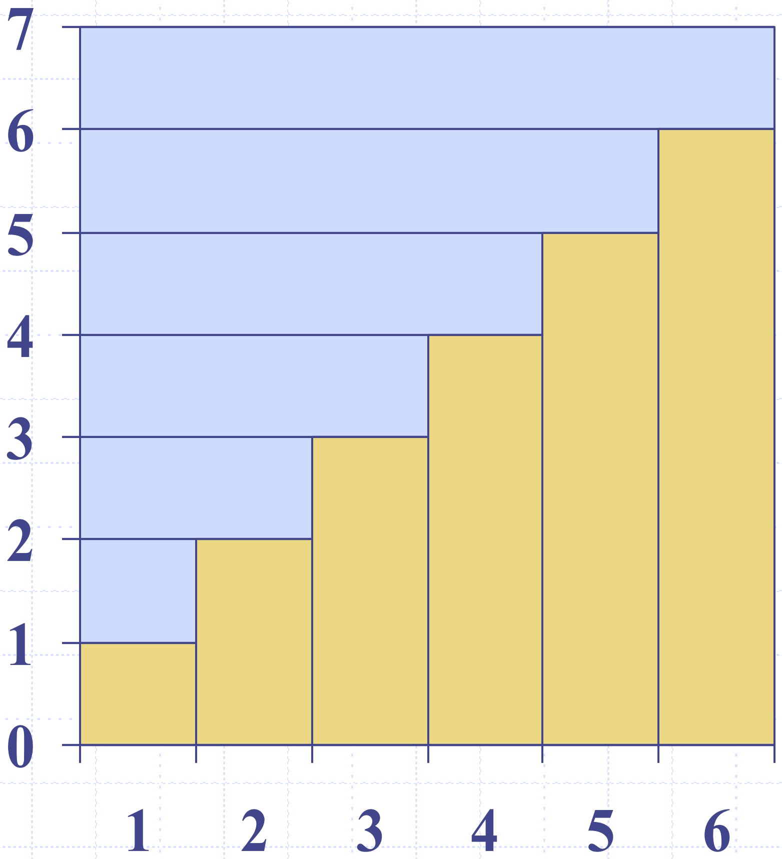
  Input: An n-element array A of numbers, indexed from 1 to n.

  Output: The maximum subarray sum of array A.

```
S_0 ← 0    // the initial prefix sum
for i ← 1 to n do
    S_i ← S_{i−1} + A[i]
m ← 0    // the maximum found so far
for j ← 1 to n do
    for k ← j to n do
        s = S_k − S_{j−1}
        if s > m then
            m ← s
return m
```

Arithmetic Progression

- The running time of MaxsubFaster 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 MaxsubFaster runs in $O(n^2)$ time



A Linear-Time Algorithm

- Instead of computing prefix sum $S_t = s_{1,t}$, let us compute a maximum suffix sum, M_t, which is the maximum of 0 and the maximum $s_{j,t}$ for j = 1,…, t.

$$M_t = \max\{0, \max_{j=1,\cdots,t}\{s_{j,t}\}\}$$

- if $M_t > 0$, then it is the summation value for a maximum subarray that ends at t, and if $M_t = 0$, then we can safely ignore any subarray that ends at t.
- if we know all the M_t values, for t = 1, 2, …, n, then the solution to the maximum subarray problem would simply be the maximum of all these values.

A Linear-Time Algorithm, cont.

- for $t \geq 2$, if we have a maximum subarray that ends at t, and it has a positive sum, then it is either A[t : t] or it is made up of the maximum subarray that ends at t – 1 plus A[t]. So we can define $M_0 = 0$ and

$$M_t = \max\{0,\ M_{t-1} + A[t]\}$$

- If this were not the case, then we could make a subarray of even larger sum by swapping out the one we chose to end at t – 1 with the maximum one that ends at t – 1, which would contradict the fact that we have the maximum subarray that ends at t.
- Also, if taking the value of maximum subarray that ends at t – 1 and adding A[t] makes this sum no longer be positive, then $M_t = 0$, for there is no subarray that ends at t with a positive summation.

A Linear-Time Algorithm, cont.

Algorithm MaxsubFastest(A):

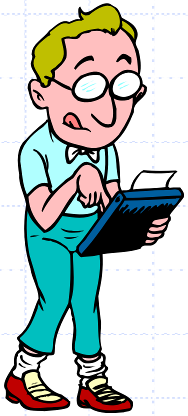
Input: An n-element array A of numbers, indexed from 1 to n.

Output: The maximum subarray sum of array A.

$M_0 \leftarrow 0$ // the initial prefix maximum
for $t \leftarrow 1$ **to** n **do**
$\quad M_t \leftarrow \max\{0,\ M_{t-1} + A[t]\}$
$m \leftarrow 0$ // the maximum found so far
for $t \leftarrow 1$ **to** n **do**
$\quad m \leftarrow \max\{m,\ M_t\}$
return m

- The MaxsubFastest algorithm consists of two loops, which each iterate exactly n times and take O(1) time in each iteration. Thus, the total running time of the MaxsubFastest algorithm is O(n).

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-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

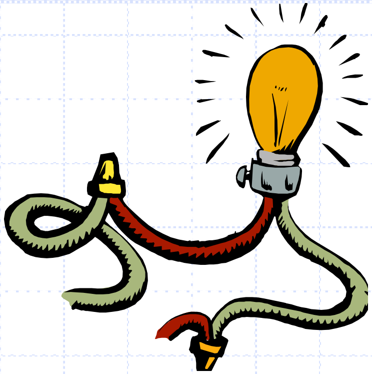
$$f(n) \geq c\, g(n) \text{ for } n \geq n_0$$

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-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)

Example Uses of the Relatives of Big-Oh

- **$5n^2$ 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 $\mathrm{f}(n) \geq c\, g(n)$ for $n \geq n_0$

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

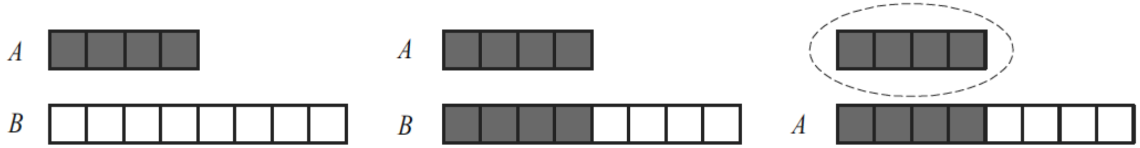
- **$5n^2$ is $\Theta(n^2)$**

  $f(n)$ 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 $\mathrm{f}(n) \leq c\, g(n)$ for $n \geq n_0$

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

Amortization

- The **amortized running time** of an operation within a series of operations is the worst-case running time of the series of operations divided by the number of operations.
- Example: A growable array, S. When needing to grow:
  a. Allocate a new array B of larger capacity.
  b. Copy A[i] to B[i], for i = 0, . . . , n − 1, where n is size of A.
  c. Let A = B, that is, we use B as the array now supporting A.



(a) (b) (c)

Growable Array Description

- Let add(e) be the operation that adds element e at the end of the array
- When the array is full, we replace the array with a larger one
- But how large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

```
Algorithm add(e)
  if t = A.length - 1 then
    B ← new array of
        size …
    for i ← 0 to n−1 do
      B[i] ← A[i]
    A ← B
  n ← n + 1
  A[n−1] ← e
```

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n add operations
- We assume that we start with an empty list represented by a growable array of size 1
- We call amortized time of an add operation the average time taken by an add operation over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- Over n add operations, we replace the array $k = n/c$ times, where c is a constant
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- Thus, the amortized time of an add operation is $O(n)$

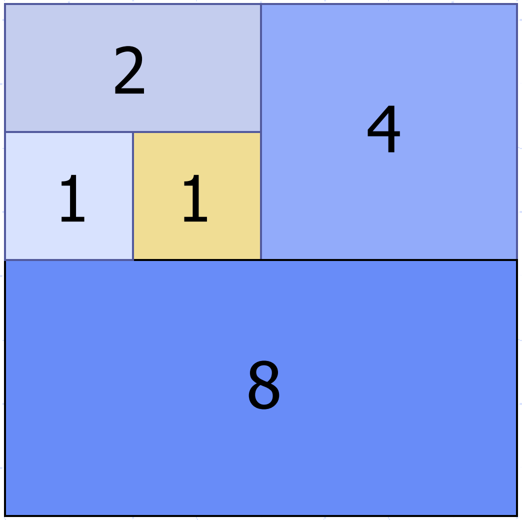
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of an add operation is $O(1)$

geometric series



Accounting Method Proof for the Doubling Strategy

- We view the computer as a coin-operated appliance that requires the payment of 1 **cyber-dollar** for a constant amount of computing time.
- We shall charge each add operation 3 cyber-dollars, that is, it will have an amortized O(1) amortized running time.
  - We over-charge each add operation not causing an overflow 2 cyber-dollars.
  - Think of the 2 cyber-dollars profited in an insertion that does not grow the array as being "stored" at the element inserted.
  - An overflow occurs when the array A has 2^i elements.
  - Thus, doubling the size of the array will require 2^i cyber-dollars.
  - These cyber-dollars are at the elements stored in cells 2^{i-1} through 2^i-1.