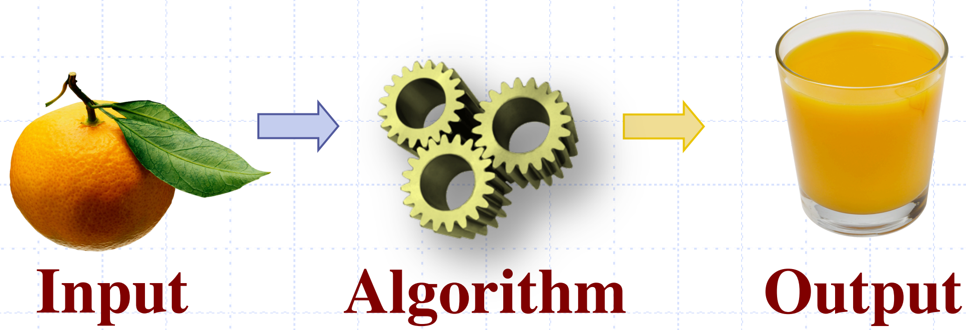
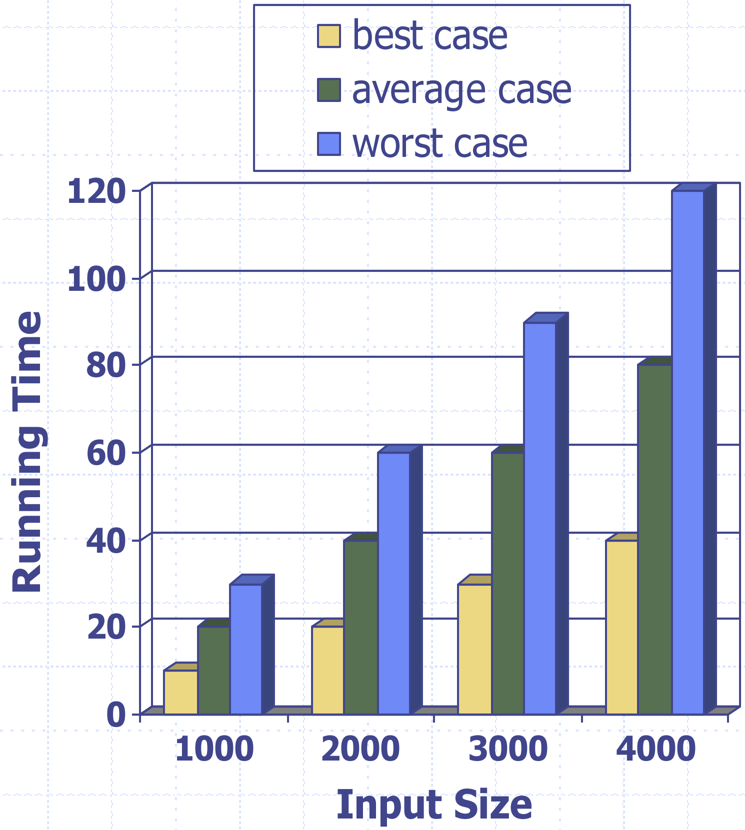
Analysis of Algorithms



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

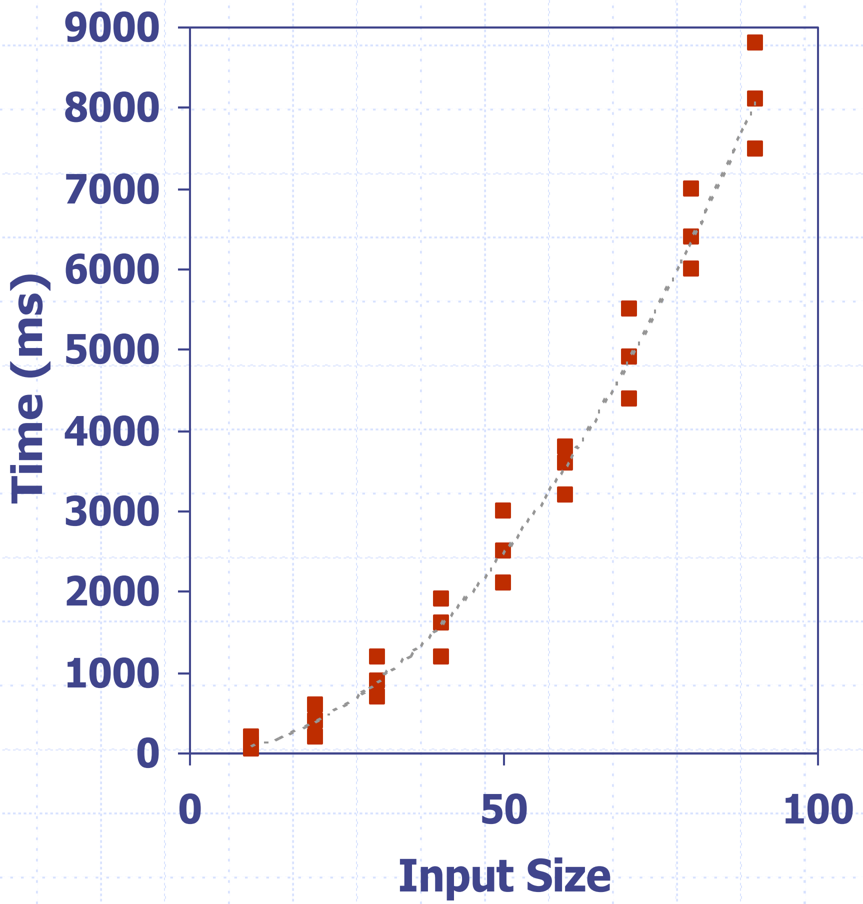


Experimental Studies

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

```
from time import time
start_time = time( )
run algorithm
end_time = time( )
elapsed = end_time − start_time
```

- 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

Pseudocode Examples

 # code en python

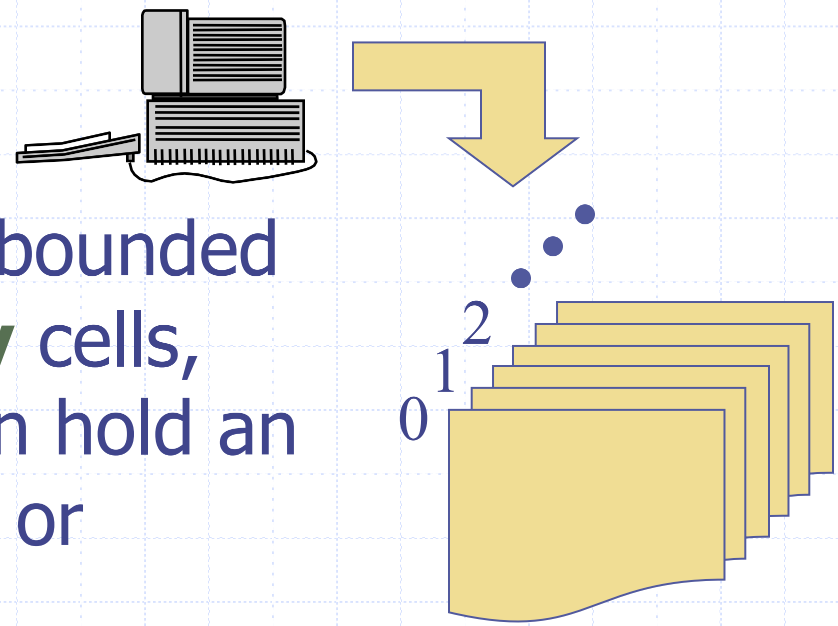
```
def recherche_binaire_iterative( data, cible ):
    min = 0
    max = len( data ) - 1
    while min <= max:
            milieu = ( min + max ) // 2
            if cible == data[milieu]:
                return True
            elif cible < data[milieu]:
                max = milieu - 1
            else:
                min = milieu + 1
    return False
```

 # pseudo code

```
recherche_binaire_iterative( data, cible )
    min = 0
    max = n – 1
    while min < max do
        milieu = ( min + max ) / 2
        if cible = data[milieu]
            return true
        else if cible < data[milieu]
            max = milieu – 1
        else min = milieu + 1
    return false
```

The Random Access Machine (RAM) Model

- A **CPU**



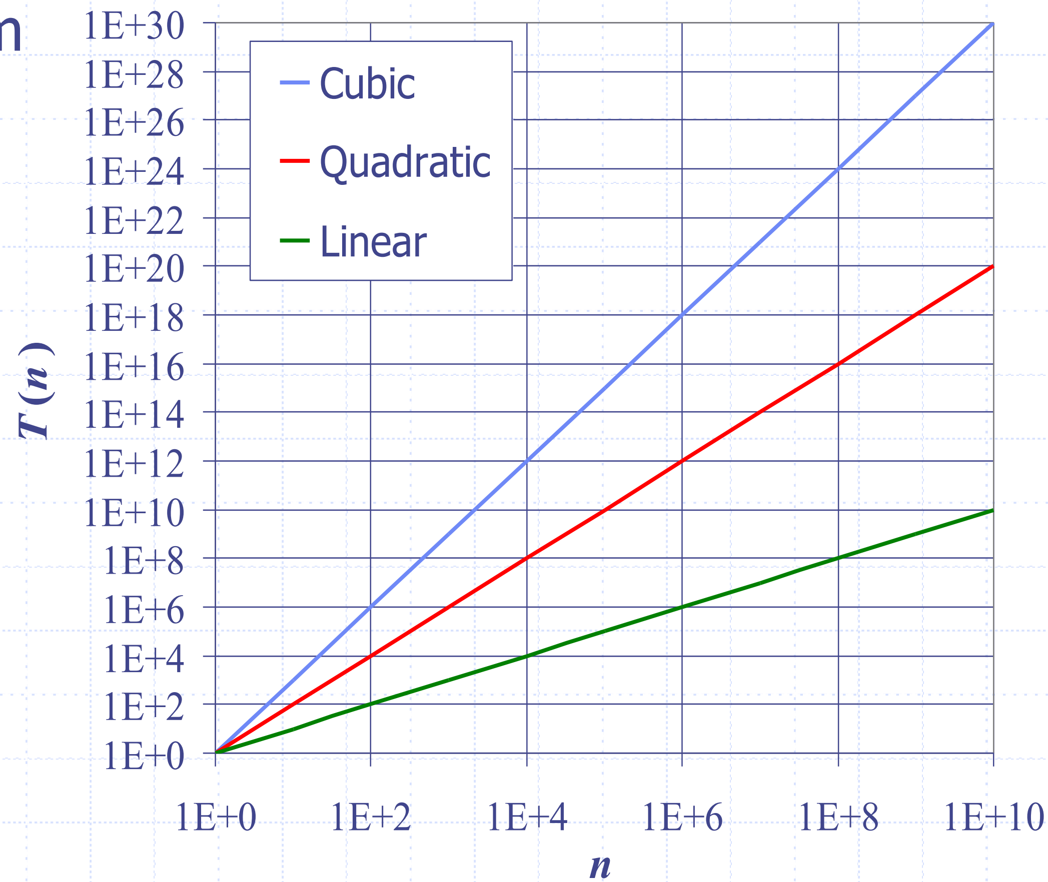
- An potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character

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

Seven Important Functions

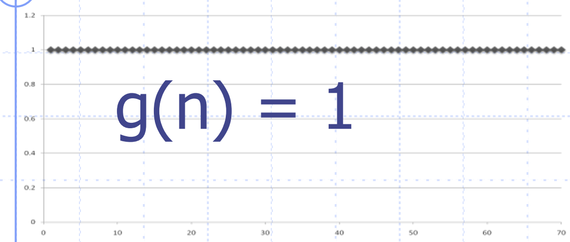
- Seven functions that often appear in algorithm analysis:
  - Constant, $f(n) = c$
  - Logarithmic, $f(n) = \log n$
  - Linear, $f(n) = n$
  - N-Log-N, $f(n) = n \log n$
  - Quadratic, $f(n) = n^2$
  - Cubic, $f(n) = n^3$
  - Exponential, $f(n) = 2^n$

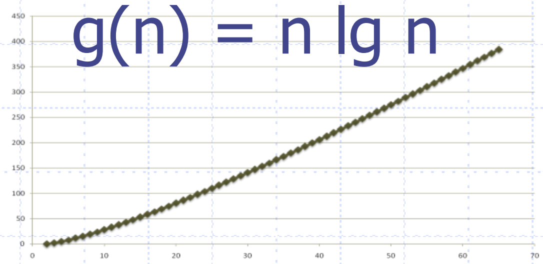
- In a log-log chart, the slope of the line corresponds to the growth rate

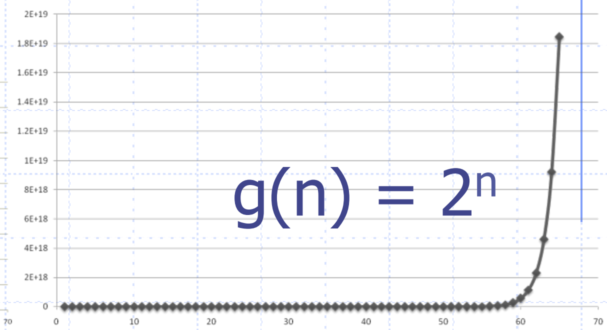


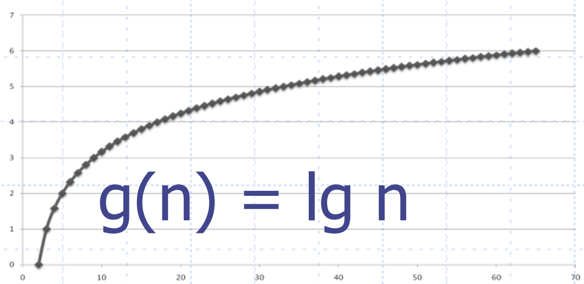
Functions Graphed Using “Normal” Scale

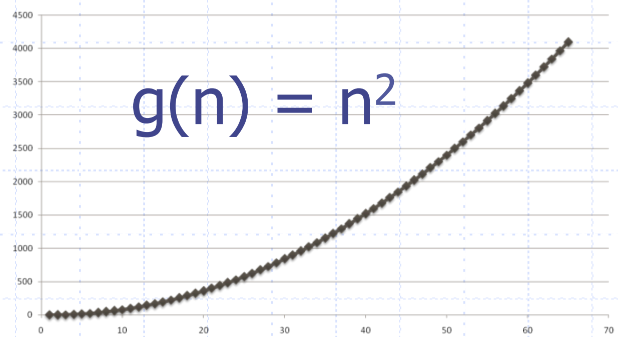
Slide by Matt Stallmann included with permission.

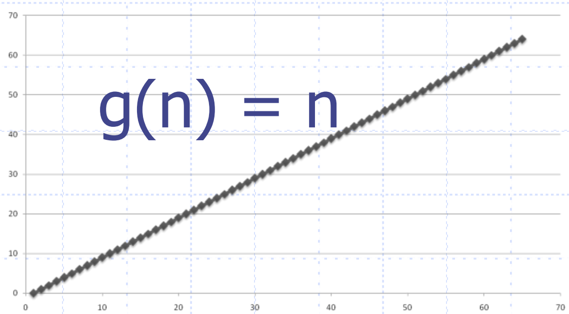


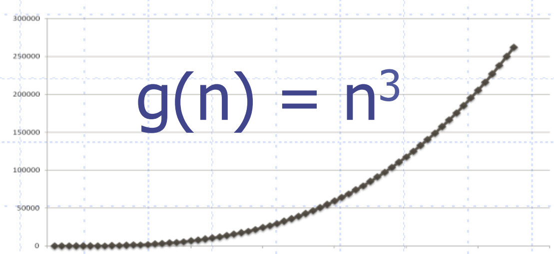












Primitive Operations

- 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

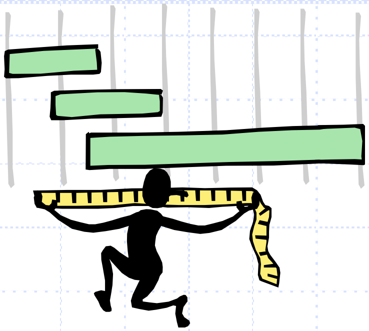
Counting Primitive Operations

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

```
1 def find_max(data):
2   """Return the maximum element from a nonempty Python list."""
3   biggest = data[0]              # The initial value to beat
4   for val in data:               # For each value:
5     if val > biggest             # if it is greater than the best so far,
6       biggest = val              # we have found a new best (so far)
7   return biggest                 # When loop ends, biggest is the max
```

- Step 1: 2 ops, 3: 2 ops, 4: 2n ops, 5: 2n ops, 6: 0 to n ops, 7: 1 op

Estimating Running Time



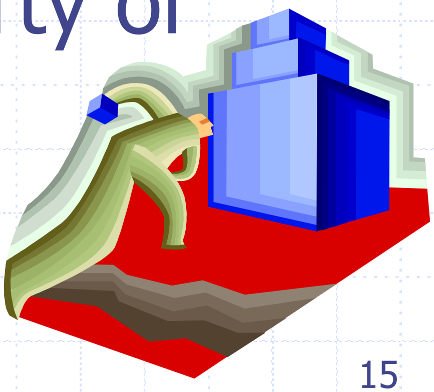
- Algorithm **find_max** executes $5n + 5$ primitive operations in the worst case, $4n + 5$ in the best case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of **find_max.** Then

$$a\,(4n + 5) \le T(n) \le b(5n + 5)$$

- 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 **find_max**



Slide by Matt Stallmann included with permission.

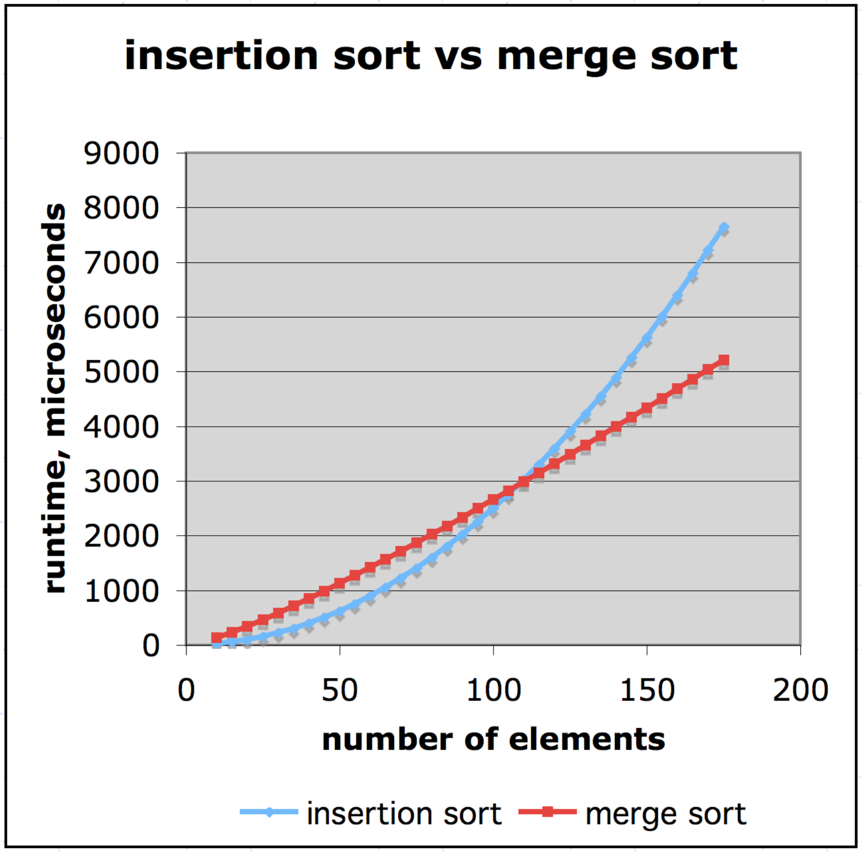
Why Growth Rate Matters

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

Slide by Matt Stallmann included with permission.

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

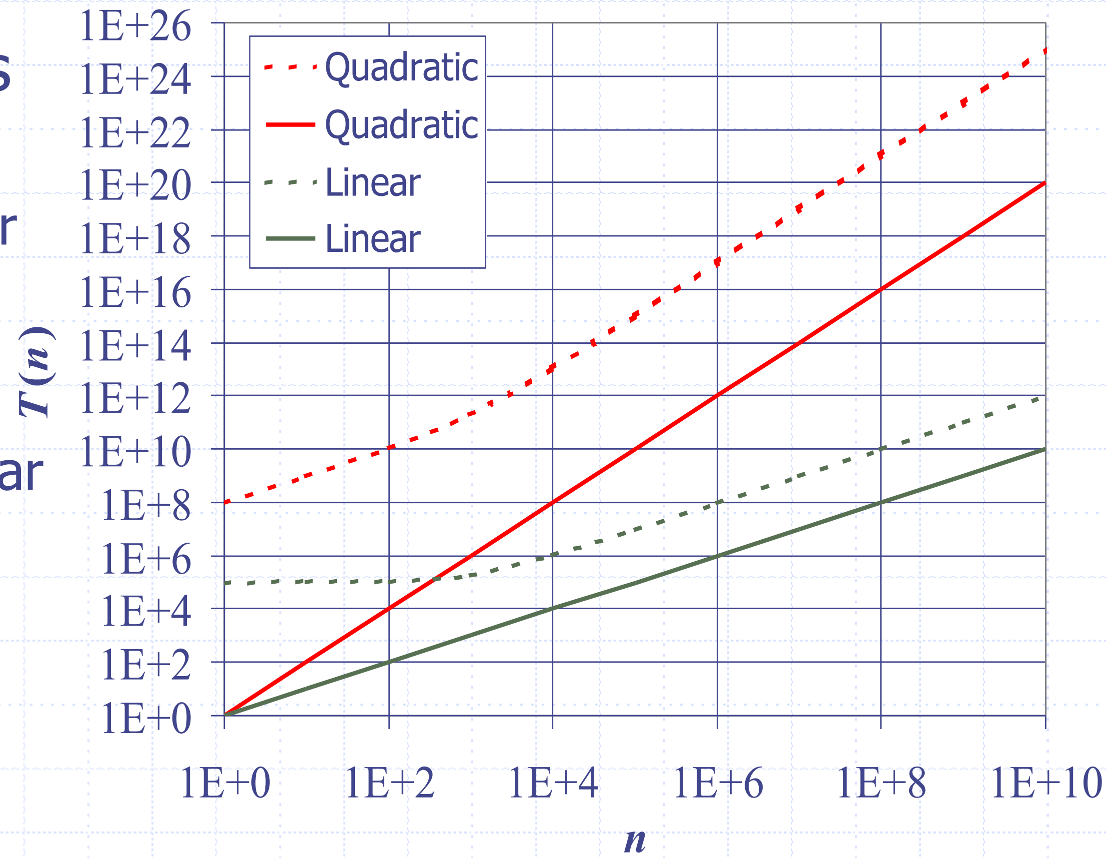
while

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
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function

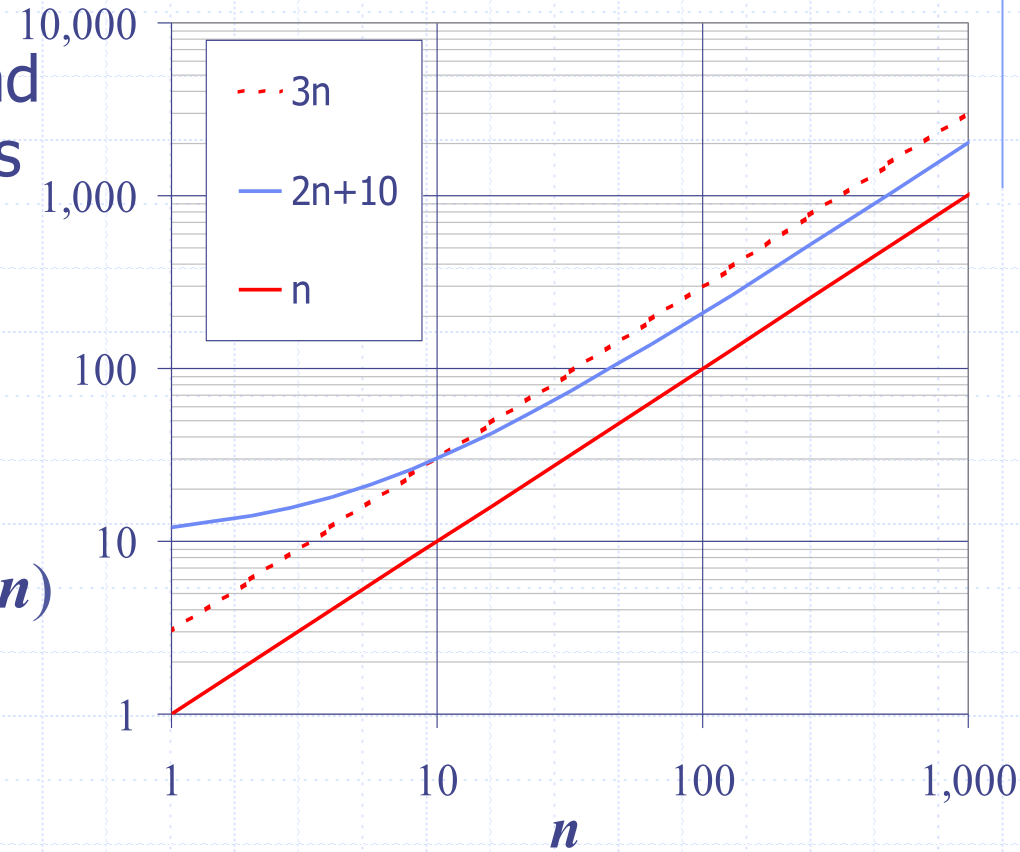


Big-Oh Notation

- 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

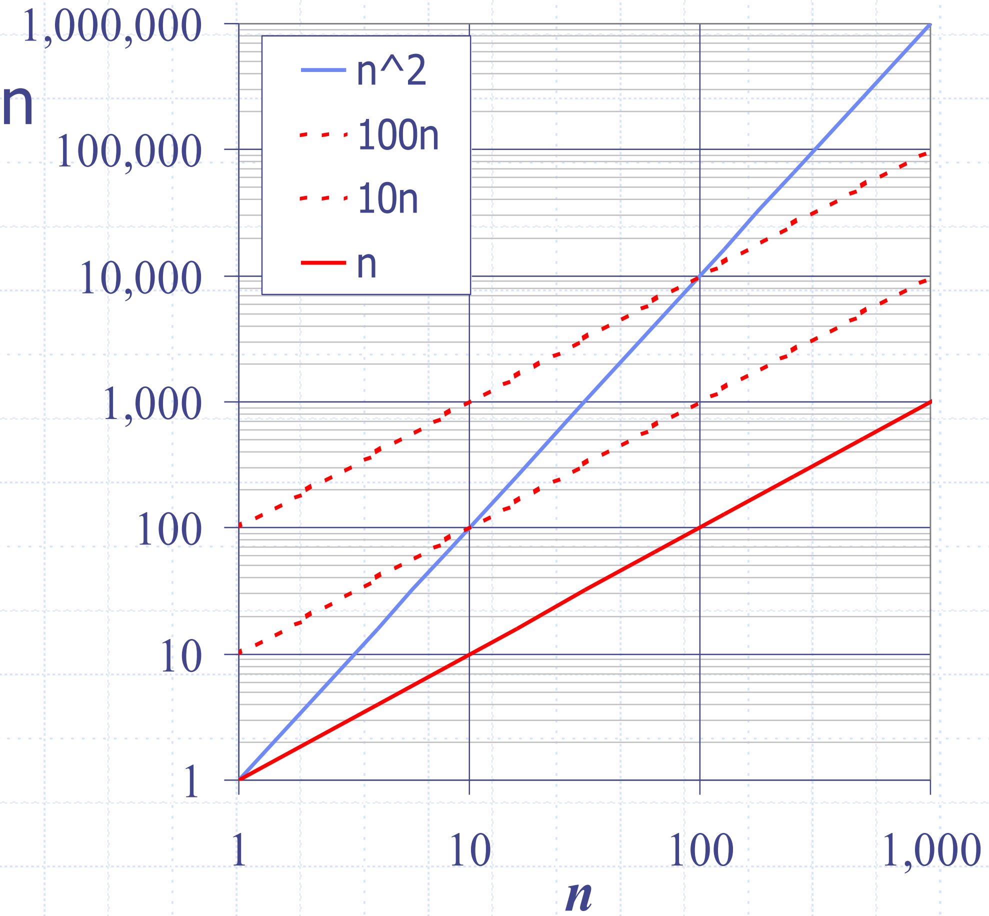
  $$f(n) \leq cg(n) \text{ for } n \geq n_0$$

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

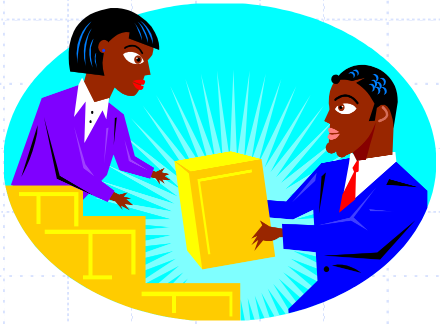


Big-Oh Example

- 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 $7n-2 \leq c \bullet n$ 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 $n_0 = 2$

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 $f(n)$ is 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

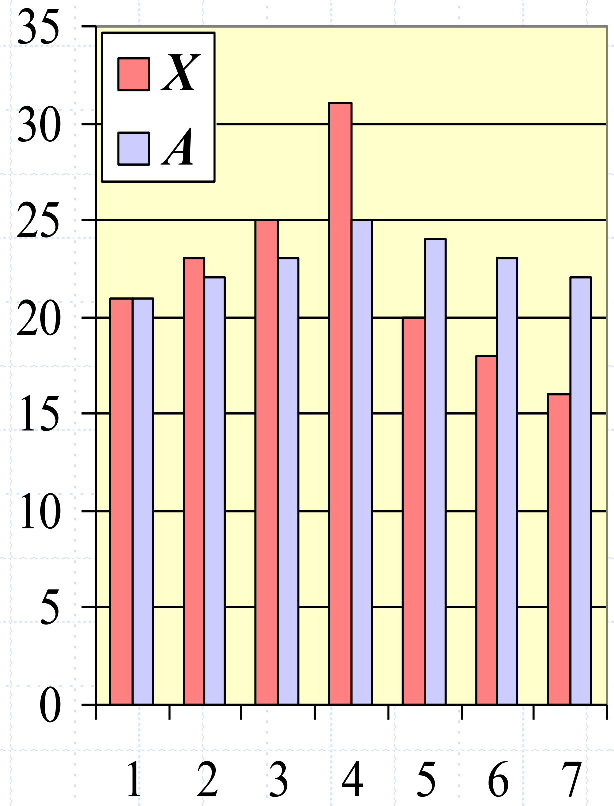
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We say that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

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

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



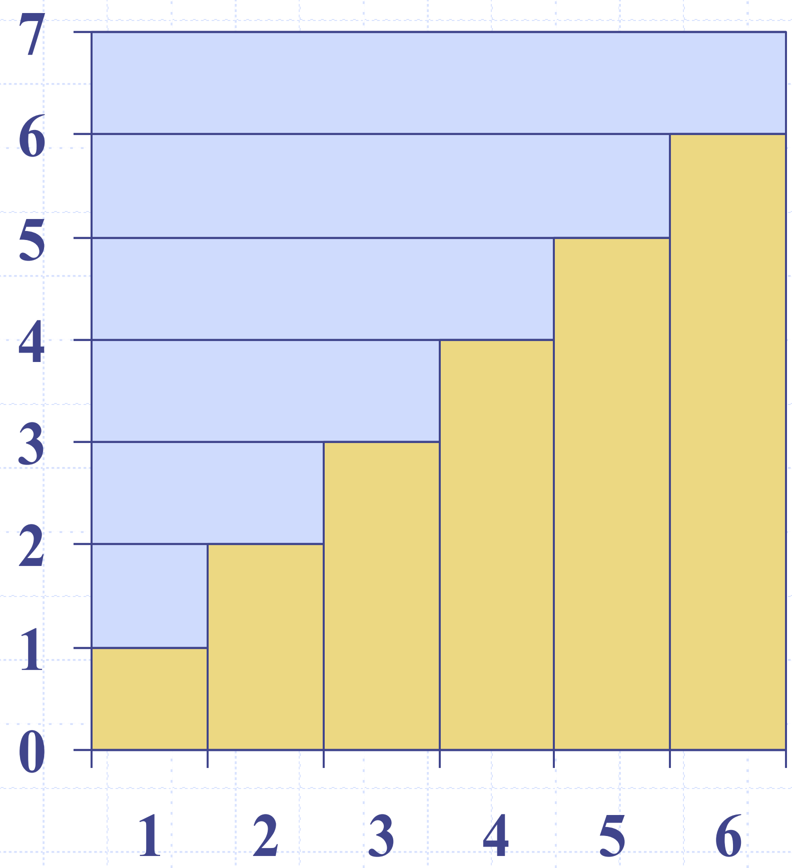
Prefix Averages (Quadratic)

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

```
def prefix_average1(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                  # create new list of n zeros
  for j in range(n):
    total = 0                  # begin computing S[0] + ... + S[j]
    for i in range(j + 1):
      total += S[i]
    A[j] = total / (j+1)       # record the average
  return A
```

Arithmetic Progression

- The running time of ***prefixAverage1*** 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 ***prefixAverage1*** runs in $O(n^2)$ time



Proving that 1+2+3+…+n is n(n+1)/2

We give three proofs here that the n-th Triangular number, 1+2+3+...+n is n(n+1)/2. The first is a visual one involving only the formula for the area of a rectangle. This is followed by two proofs using algebra. The first uses "..." notation and the second introduces you to the Sigma notation which makes the proof more precise.

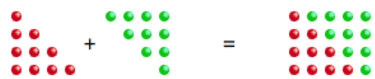
A visual proof that 1+2+3+...+n = n(n+1)/2

We can visualize the sum 1+2+3+...+n as a **triangle of dots**. Numbers which have such a pattern of dots are called **Triangle (or triangular) numbers**, written T(n), the sum of the integers from 1 to n :

n	1	2	3	4	5	6
T(n) as a sum	1	1+2	1+2+3	1+2+3+4	1..5	1..6
T(n) as a triangle					...	
T(n)=	**1**	**3**	**6**	**10**	**15**	**21**

For the proof, we will *count the number of dots in T(n)* but, instead of *summing the numbers 1, 2, 3, etc up to n* we will find the total using only *one multiplication and one division!*

To do this, we will fit **two copies of a triangle of dots together**, one red and an upside-down copy in green.
E.g. T(4)=1+2+3+4

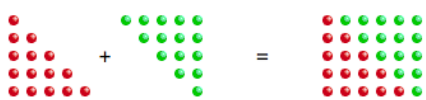


Notice that

- we get a **rectangle** which is has the same number of rows (4) but has one extra column (5)
- so the rectangle is **4 by 5**
- it therefore contains 4x5=20 balls
- but we took **two** copies of T(4) to get this
- so we must have 20/2 = 10 balls in T(4), which we can easily check.

This visual proof applies to any size of triangle number.

Here it is again on T(5):



So T(5) is half of a rectangle of dots 5 tall and 6 wide, i.e. half of 30 dots, so T(5)=15.

http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/runsums/triNbProof.html

Prefix Averages 2 (Looks Better)

- The following algorithm uses an internal Python function to simplify the code

```
def prefix_average2(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                              # create new list of n zeros
  for j in range(n):
    A[j] = sum(S[0:j+1]) / (j+1)           # record the average
  return A
```

- Algorithm ***prefixAverage2*** **still** runs in $O(n^2)$ time!

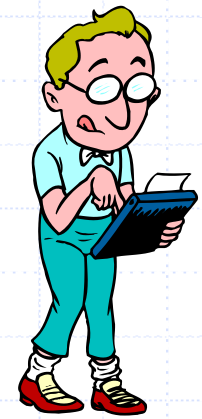
Prefix Averages 3 (Linear Time)

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

```
def prefix_average3(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                    # create new list of n zeros
  total = 0                      # compute prefix sum as S[0] + S[1] + ...
  for j in range(n):
    total += S[j]                # update prefix sum to include S[j]
    A[j] = total / (j+1)         # compute average based on current sum
  return A
```

- Algorithm ***prefixAverage3*** runs in $O(n)$ time

Math you need to Review



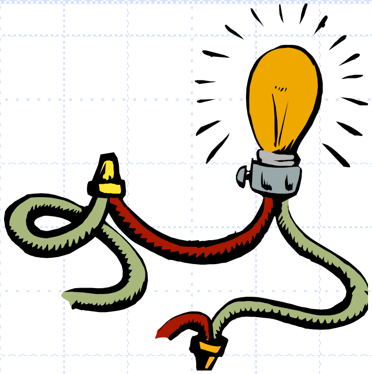
- Summations
- Logarithms and Exponents
- Proof techniques
- Basic probability

- **properties of logarithms:**
  $\log_b(xy) = \log_b x + \log_b y$
  $\log_b (x/y) = \log_b x - \log_b y$
  $\log_b xa = a\log_b x$
  $\log_b a = \log_x a/\log_x b$
- **properties of exponentials**:
  $a^{(b+c)} = a^b a^{c}$
  $a^{bc} = (a^b)^c$
  $a^b / a^c = a^{(b-c)}$
  $b = a^{\log_a b}$
  $b^c = a^{c*\log_a b}$

Relatives of Big-Oh

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

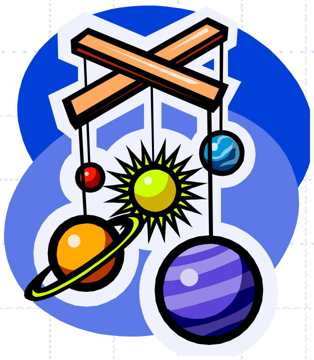
big-Omega

- 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)

Example Uses of the Relatives of Big-Oh



- **$5n^2$ is $O(n^2)$**

  $f(n)$ is $\mathbf{O}(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \bullet g(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

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

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

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

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

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

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

Définitions graphiques de **O**, **Ω** et **Θ**