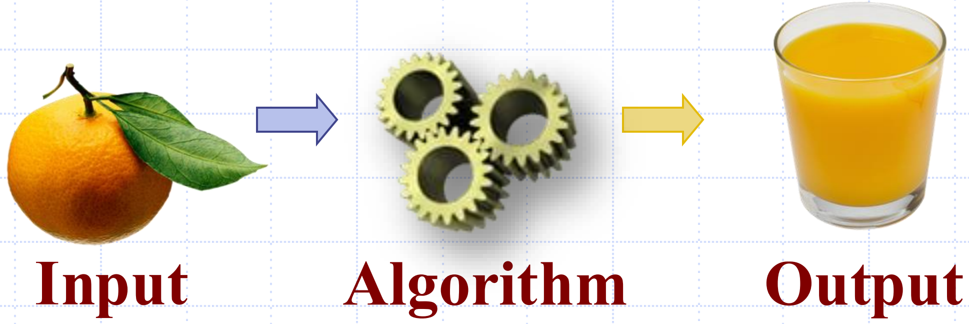
Analysis of Algorithms



Algorithm

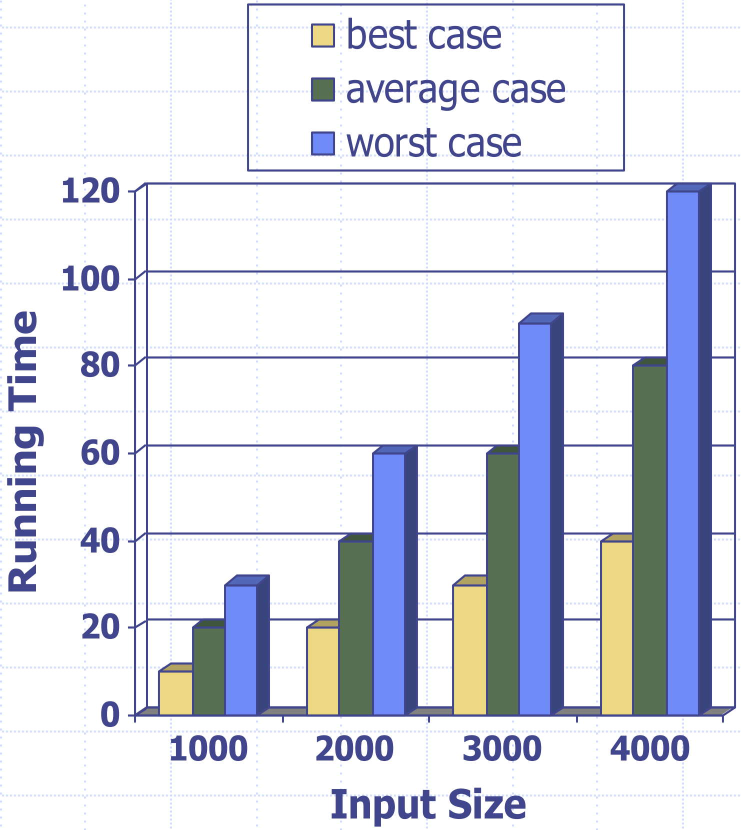
- step by step procedure for performing some task in a finite amount of time

- “goodness” of an algorithm

Google images

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

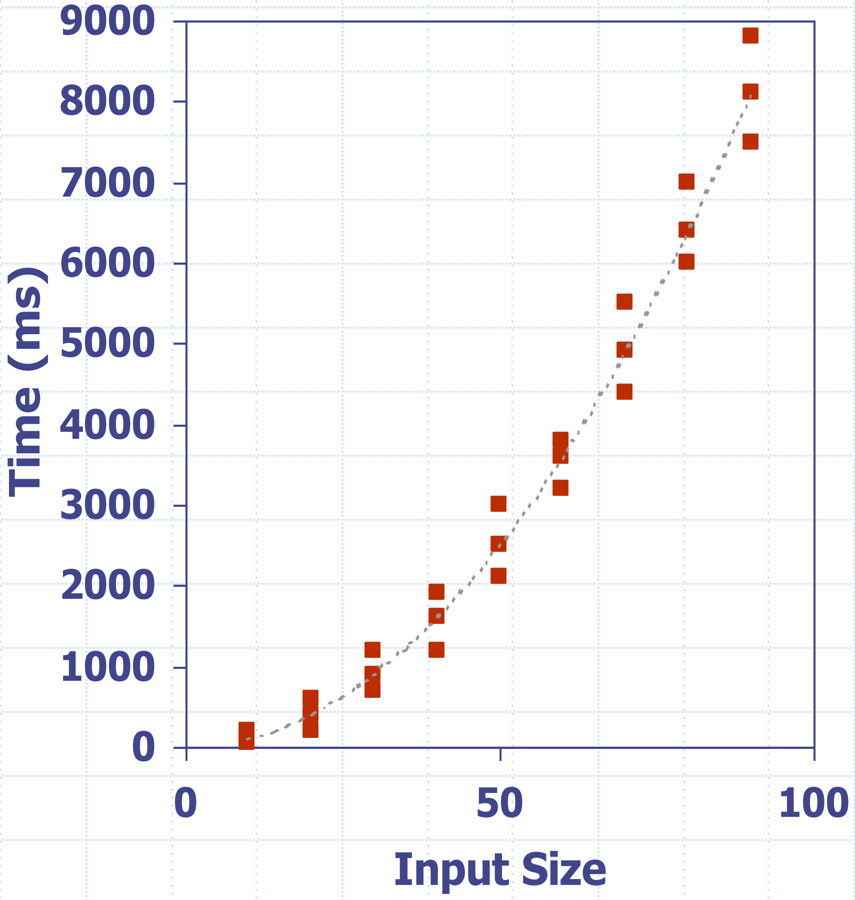


Experimental Studies

- 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

Running Time - Worst Case Input

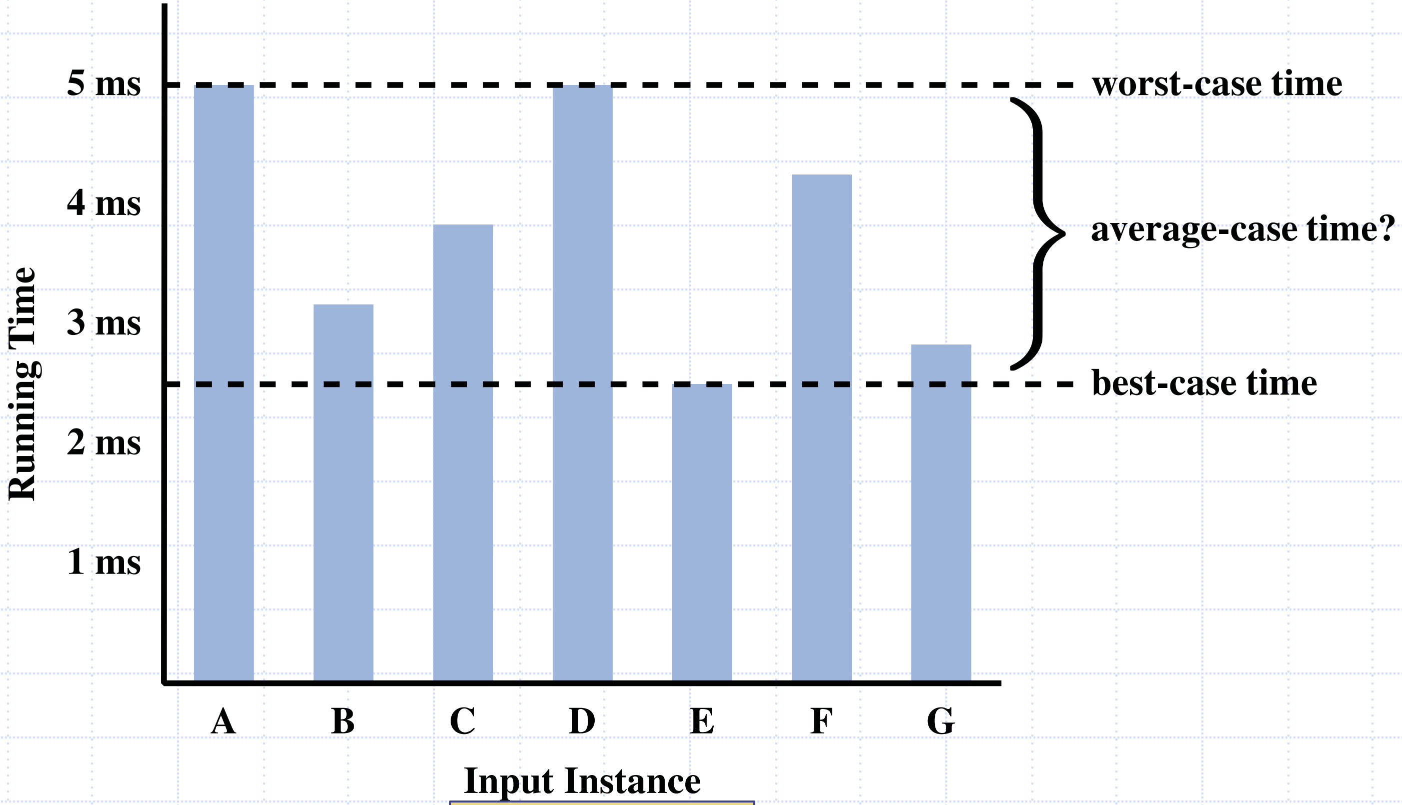
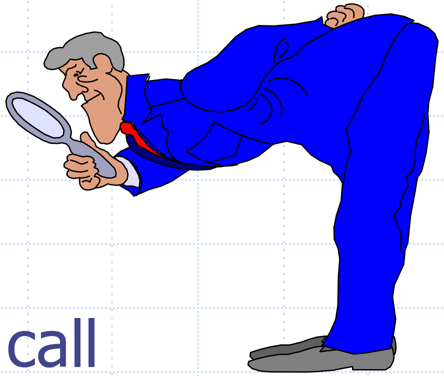


Figure 3.2

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 - Finding the maximum value in an array

- Algorithm find_max(A)
  - Input: an array of numbers - A
  - Output: the largest element in the array A
- current_max $\leftarrow$ A(0)
- for j $\leftarrow$ 1 to length of A
- do
  - if A(j) > current_max then
    - current_max $\leftarrow$ A(j)
  - end if
- end for
- return current_max

Pseudocode – Finding if an element is present in an array

- Algorithm FindEle(A, e)
  - Input
  - Output

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

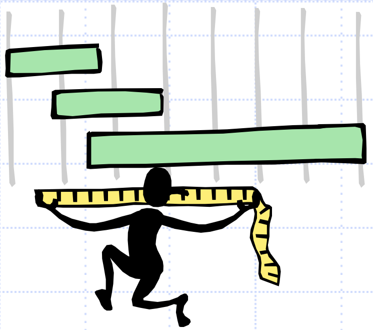
Counting Primitive Operations

- 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



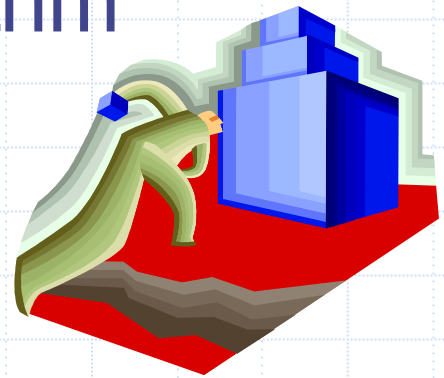
- Algorithm **find_max** executes $5n + 5$ primitive operations in the worst case, $4n + 5$ in the best case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of **find_max.** Then

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

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

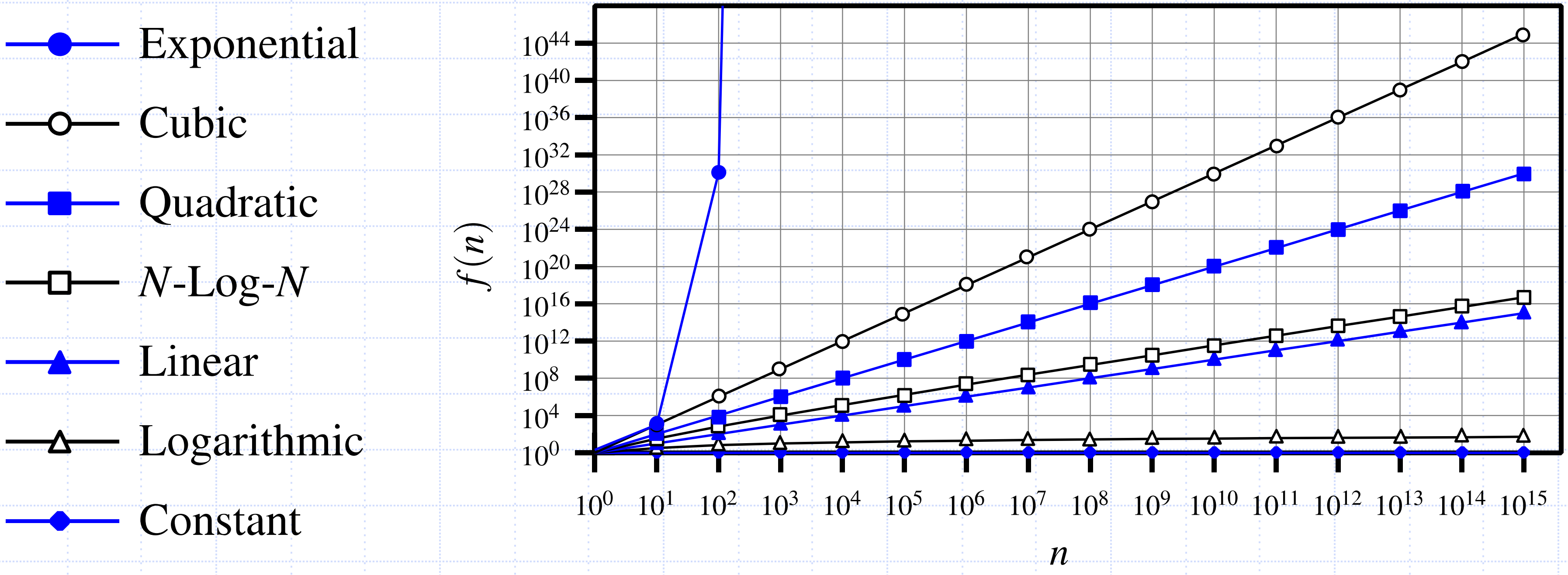
Growth Rate of Running Time

- 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**



Seven Important Functions

- Seven functions that often appear in algorithm analysis:



Slide by Matt Stallmann included with permission.

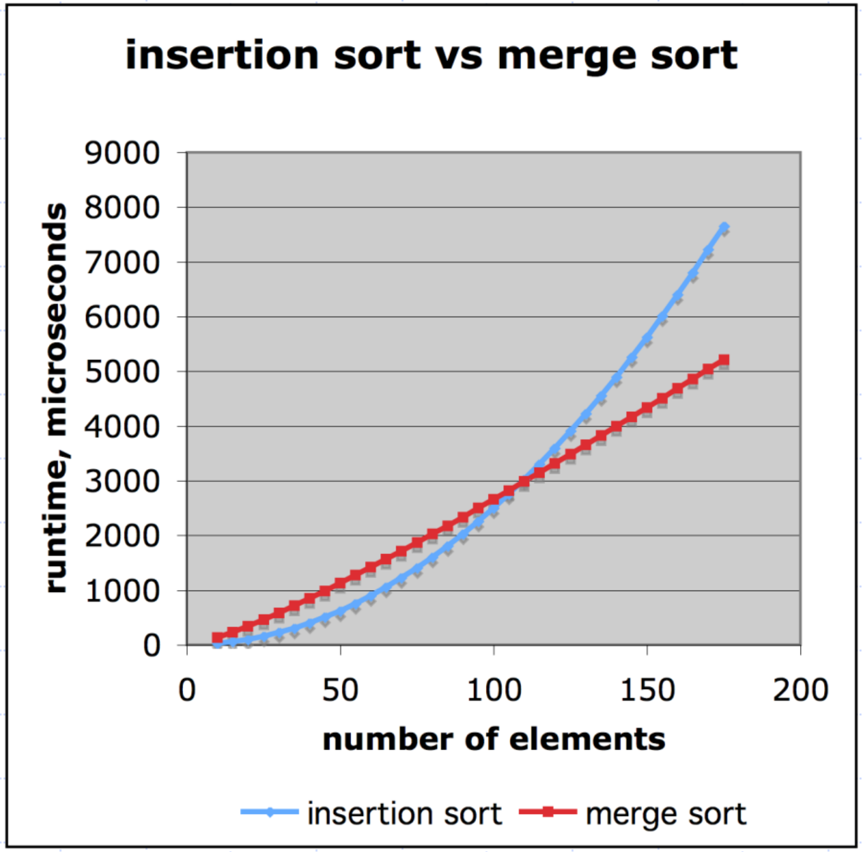
Why Growth Rate Matters

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

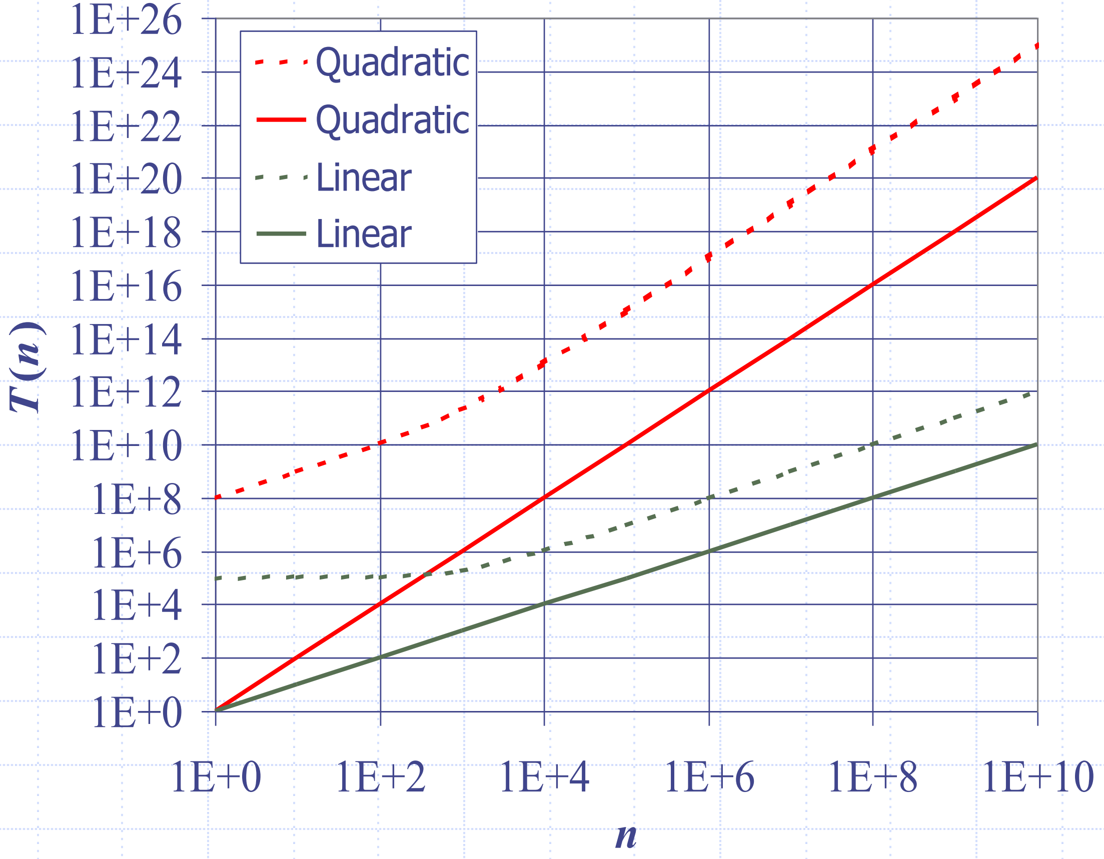
while

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

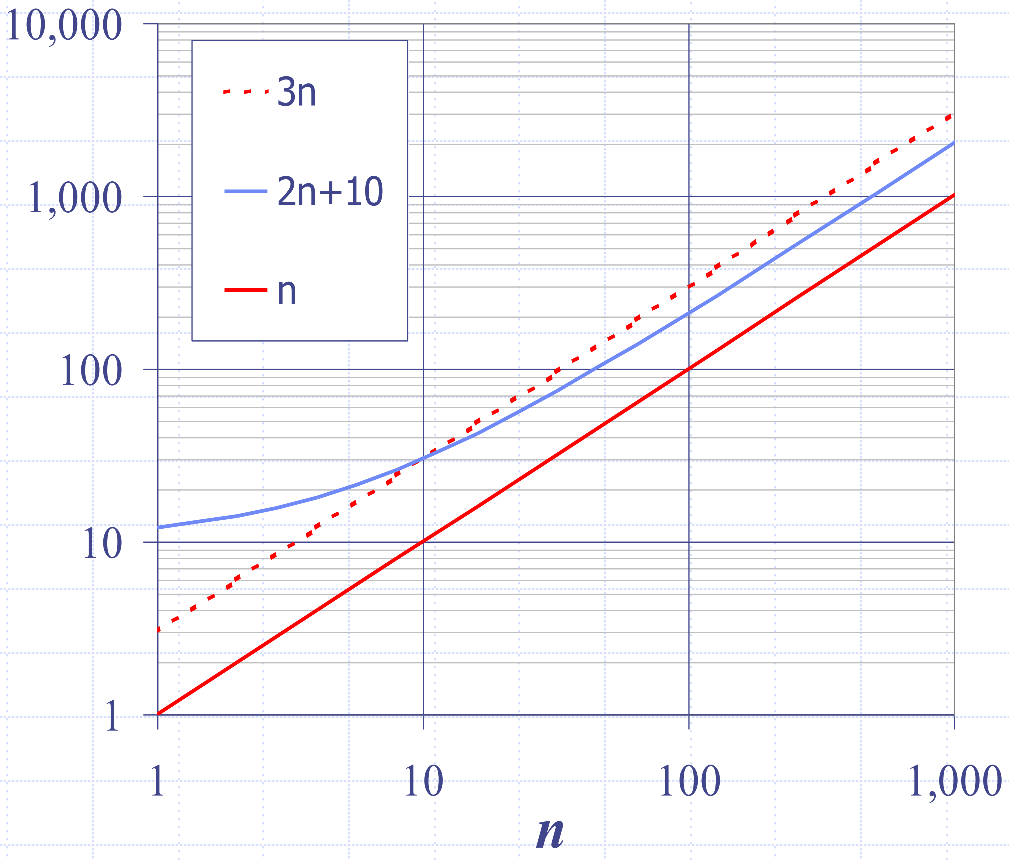


Big-Oh Notation

- 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

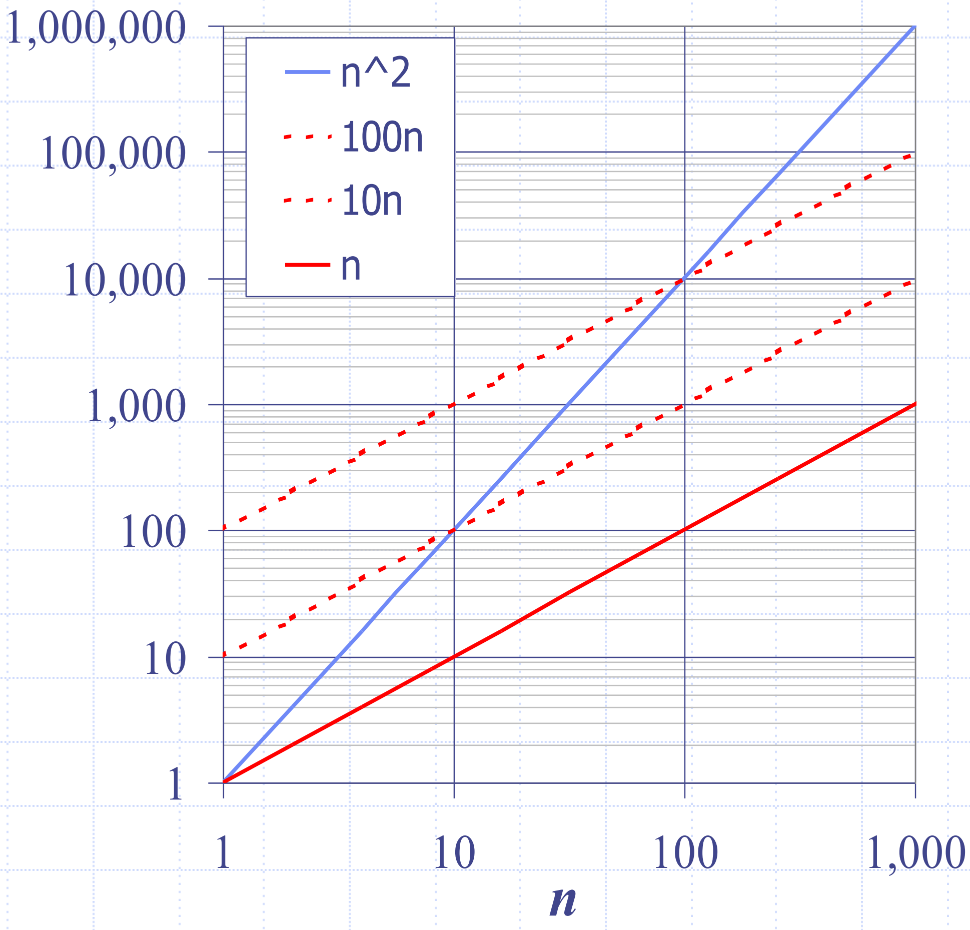
  $f(n) \leq cg(n)$ for $n \geq n_0$

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

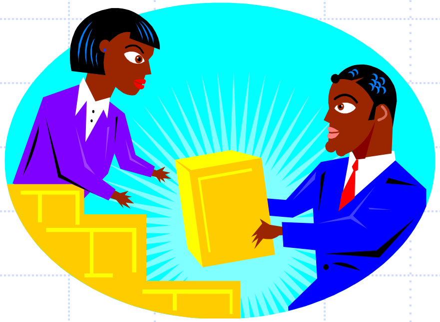


Big-Oh Example

- 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 $7n - 2 \leq cn$ 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 $f(n)$ 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

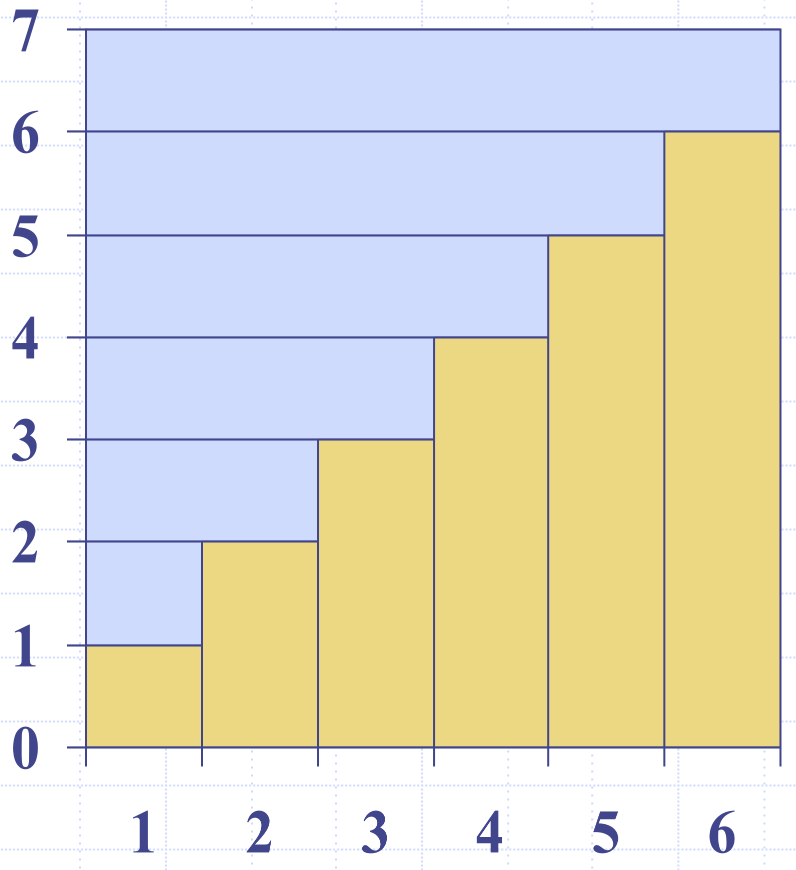
Prefix Averages (Quadratic)

- 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



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

Show that if $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n) + e(n)$ is $O(f(n)+g(n))$

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

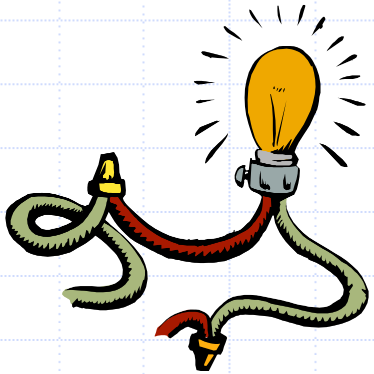
$$f(n) \geq c\, g(n) \text{ for } n \geq n_0$$

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

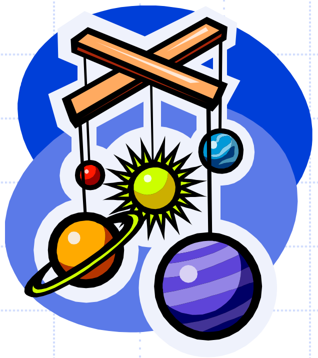


big-Oh
- f(n) is O(g(n)) if f(n) is asymptotically less than or equal to g(n)

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

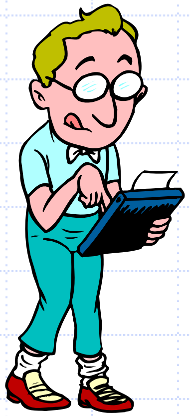
  let $c = 1$ and $n_0 = 1$

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

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

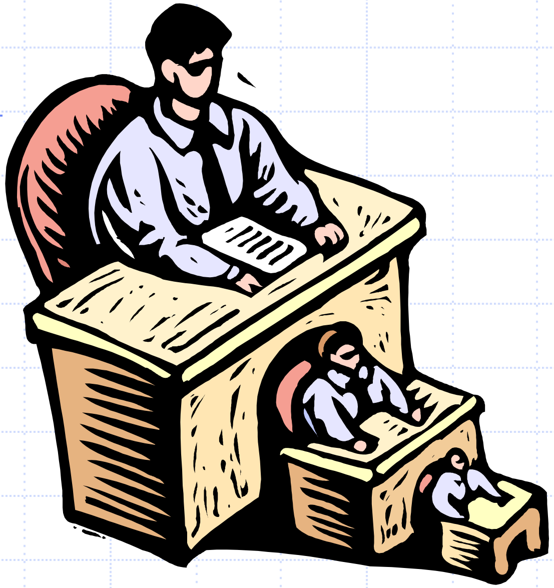
Math you need to Review



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

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

Recursion



The Recursion Pattern

- Recursion: when a method calls itself
- Classic example – the factorial function:

  $$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$$

- Recursive definition:

  $$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \textit{else} \end{cases}$$

- As a Python method:

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n−1)
```

Content of a Recursive Method

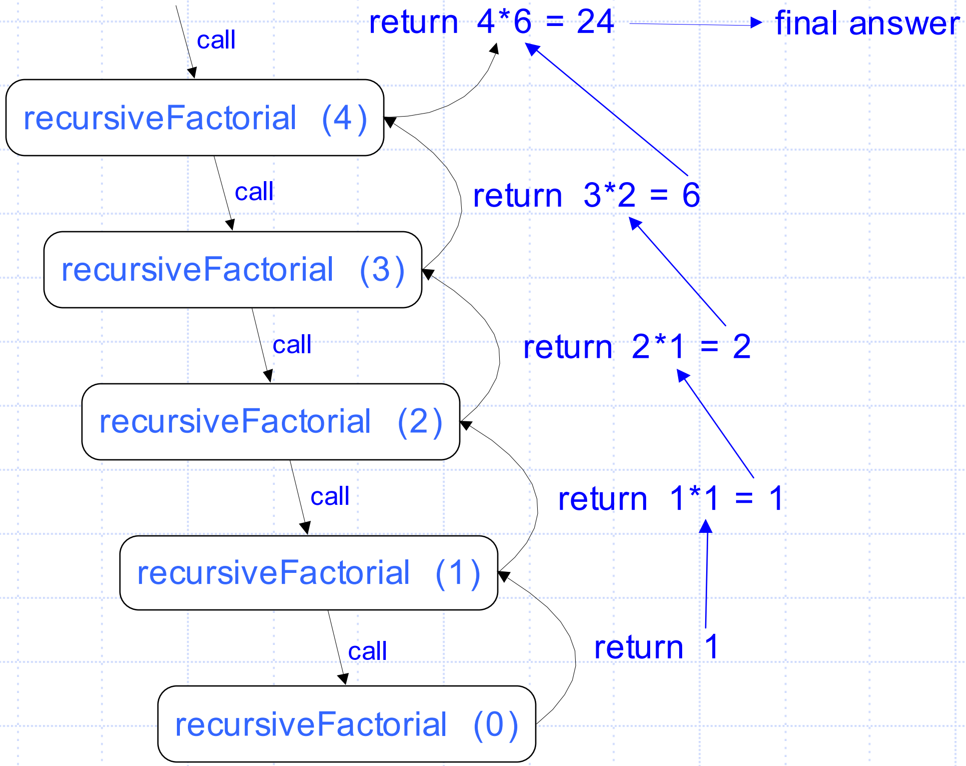
- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Not necessary - mutual recursion

Visualizing Recursion

- Recursion trace
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value

- Example



Linear Recursion

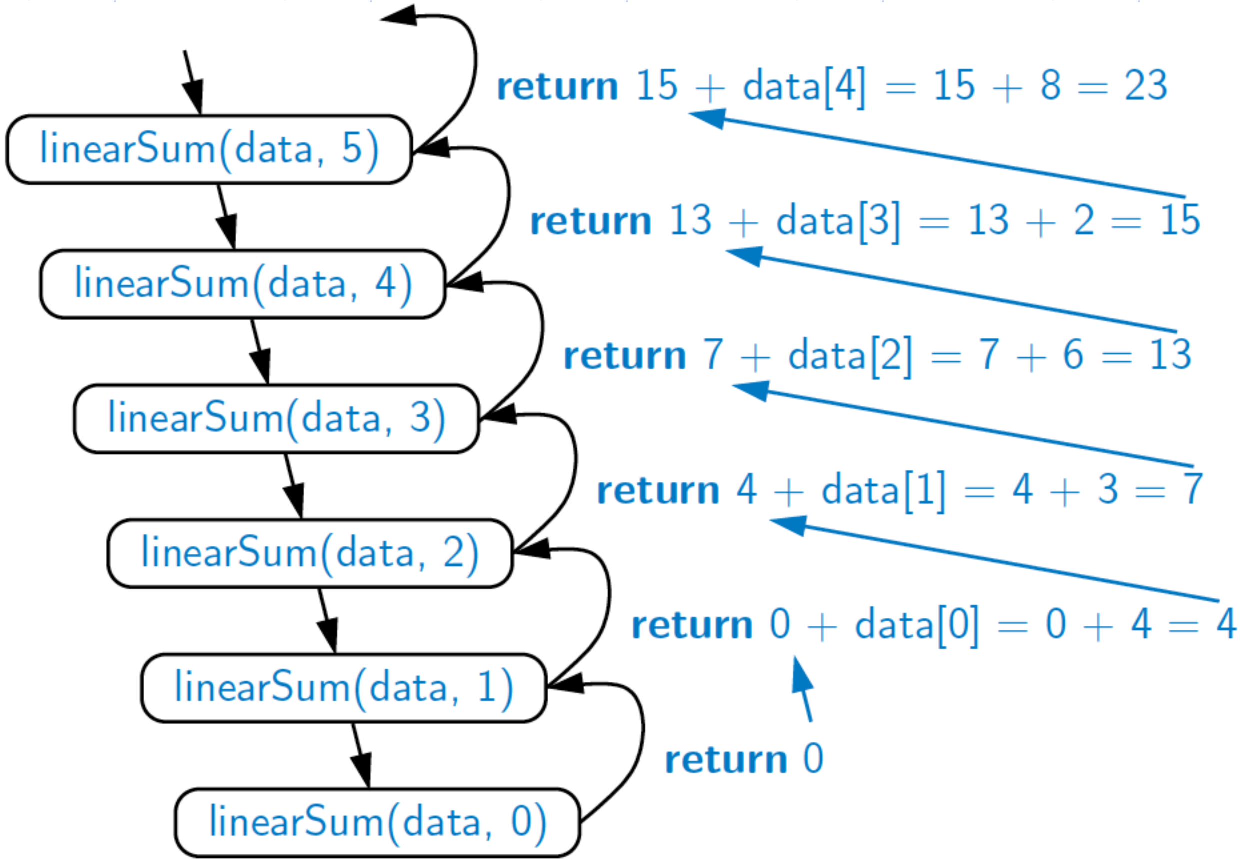
- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.

Example of Linear Recursion

Algorithm linearSum(A, n):
Input:
  Array, A, of integers
  Integer n such that
    $0 \leq n \leq |A|$
Output:
    Sum of the first n integers in A

if n = 0 then
  return 0
else
  return
linearSum(A, n - 1) + A[n - 1]

Recursion trace of linearSum(data, 5) called on array data = [4, 3, 6, 2, 8]



Binary Search

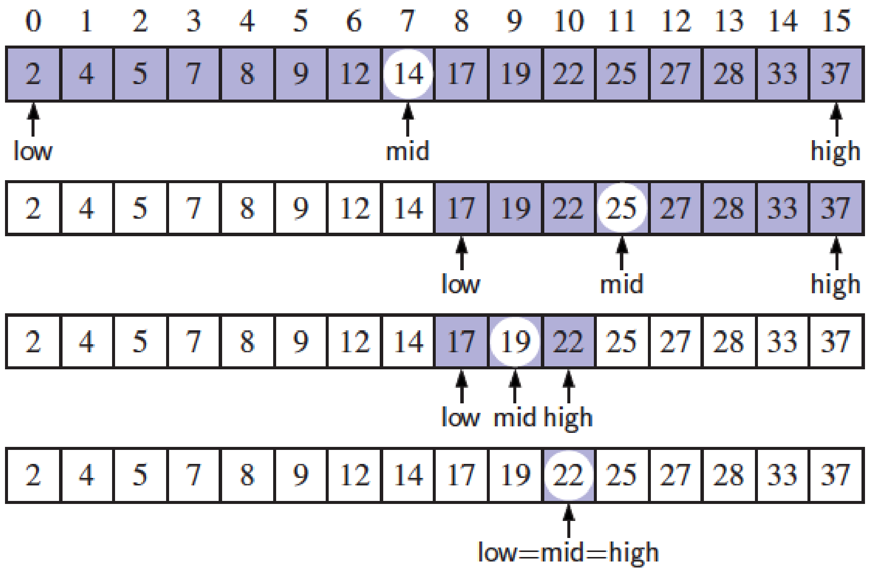
Search for an integer in an ordered list

```
def binary_search(data, target, low, high):
  """Return True if target is found in indicated portion of a Python list.

  The search only considers the portion from data[low] to data[high] inclusive.
  """
  if low > high:
    return False                       # interval is empty; no match
  else:
    mid = (low + high) // 2
    if target == data[mid]:            # found a match
      return True
    elif target < data[mid]:
      # recur on the portion left of the middle
      return binary_search(data, target, low, mid − 1)
    else:
      # recur on the portion right of the middle
      return binary_search(data, target, mid + 1, high)
```

Visualizing Binary Search

- We consider three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.



Analyzing Binary Search

- Runs in O(log n) time.
  - The remaining portion of the list is of size high – low + 1
  - After one comparison, this becomes one of the following:

$$(\mathrm{mid}-1)-\mathrm{low}+1=\left\lfloor\frac{\mathrm{low}+\mathrm{high}}{2}\right\rfloor-\mathrm{low}\leq\frac{\mathrm{high}-\mathrm{low}+1}{2}$$

$$\mathrm{high}-(\mathrm{mid}+1)+1=\mathrm{high}-\left\lfloor\frac{\mathrm{low}+\mathrm{high}}{2}\right\rfloor\leq\frac{\mathrm{high}-\mathrm{low}+1}{2}.$$

  - Thus, each recursive call divides the search region in half; hence, there can be at most log n levels

Reversing an Array

Algorithm reverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

```
if i < j then
        Swap A[i] and A[j]
        reverseArray(A, i + 1, j – 1)
return
```

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional parameters that are passed to the method.
- For example, we defined the array reversal method as reverseArray(A, i, j), not reverseArray(A)

```
def reverse(S, start, stop):
  """Reverse elements in implicit slice S[start:stop]."""
  if start < stop − 1:                            # if at least 2 elements:
    S[start], S[stop−1] = S[stop−1], S[start]     # swap first and last
    reverse(S, start+1, stop−1)                   # recur on rest
```

Binary Recursion

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.
- Fibonacci numbers are defined recursively:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1.$$

- Recursive algorithm (first attempt):

Algorithm BinaryFib(k):
 Input: Nonnegative integer k
 Output: The kth Fibonacci number F_k
 if $k = 1$ **then**
  return k
 else
  return BinaryFib($k - 1$) + BinaryFib(k □ 2)

Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!

A Better Fibonacci Algorithm

- Use linear recursion instead

```
Algorithm LinearFibonacci(k):
  Input: A nonnegative integer k
  Output: Pair of Fibonacci numbers (F_k , F_{k−1})
 if k = 1 then
     return (k, 0)
 else
     (i,  j)  =  LinearFibonacci(k – 1)
     return (i +j, i)
```

- LinearFibonacci makes $k-1$ recursive calls

Maximum recursive depth

- Infinite recursion
  - when the base case is never reached
  - programming error
    - fibonacci(int n)
      - return fibonacci(n) + fibonacci(n-1)
- StackOverflowError
  - limit on the number of recursive calls that can be made