Computer Science 1000: Part #2

Algorithms and Programming

Problems, Algorithms, and Programs

Programming in Python I: Basics

Algorithms and Pseudocode

Example Algorithms

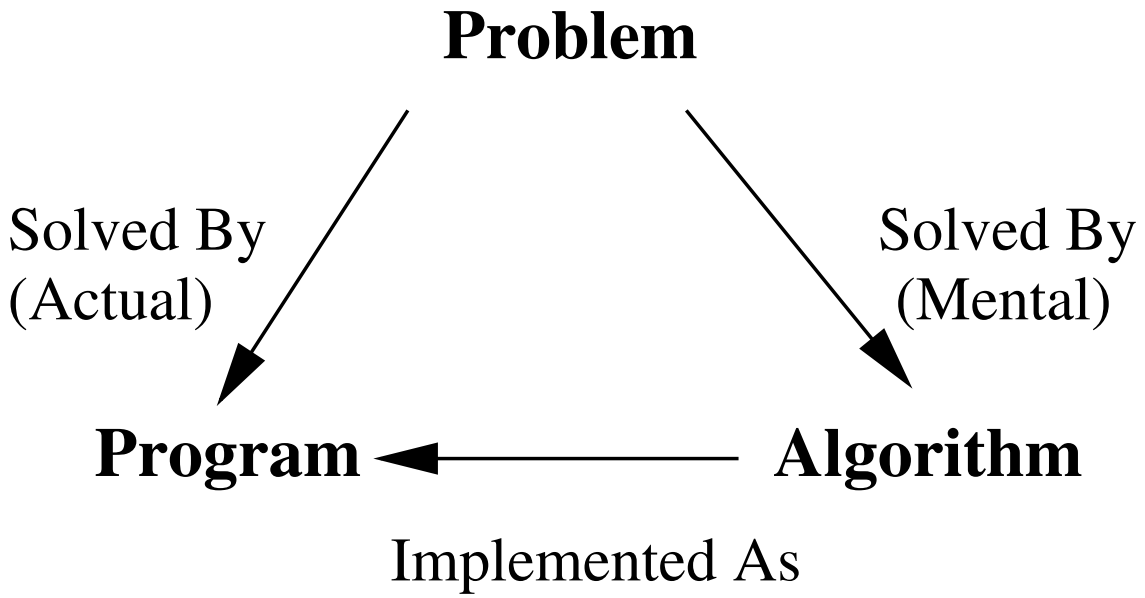
Programming in Python II: Functions

. . . To Recap . . .

- The fundamental task of Computer Science is the design and development of algorithms for solving important problems.
- An **algorithm** is a well-ordered sequence of unambiguous and effectively computable operations that produces a result and halts in a finite amount of time.

IF WE CAN SPECIFY AN ALGORITHM
TO SOLVE A PROBLEM,
WE CAN AUTOMATE ITS SOLUTION!!!

Problems, Algorithms, and Programs



Problem: A set of inputs and their associated outputs.

Algorithm: A sequence of instructions that solves a problem, *i.e.*, computes the output for a given input.

Program: A sequence of instructions *in some computer language* that solves a problem.

Example: Finding the Area of a Circle

Problem:

Input: A radius r.
Output: The area of a circle with radius r.

Algorithm:

Get the value of radius r
Set the value of area to $\pi \times r^2$
print area

Program:

```
r = int(input("r?  "))
area = 3.14159 * r * r
print("Area is ", area)
```

Example: Summing a List

Problem:

Input: A list L of n numbers.
Output: The sum of the numbers in L.

Algorithm:

Get the values for list L and n
Set the values of INDEX to 1 and SUM to 0
While (INDEX $\leq$ n) do
    Set the value of SUM to SUM + L_{INDEX}
    Set the value of INDEX to INDEX + 1
print the value of SUM

Example: Summing a List (Cont'd)

Program #1:

```
sum = 0
L = []
n = int(input("n?  "))
for i in range(1, n + 1):
    L.append(int(input("L-item #", \
                 i, "?  ")))
for i in range(1, n + 1):
    sum = sum + L[i]
print("Sum is ", sum)
```

Example: Summing a List (Cont'd)

Program #2:

```
sum = 0
n = int(input("n?  "))
for i in range(1, n + 1):
    curL = int(input("L-item #", \
                     i, "?  "))
    sum = sum + curL
print("Sum is ", sum)
```

Example: Finding the Maximum Value in a List

Problem #1:

Input: A list L with n elements.
Output: The largest-valued element in L.

Problem #2:

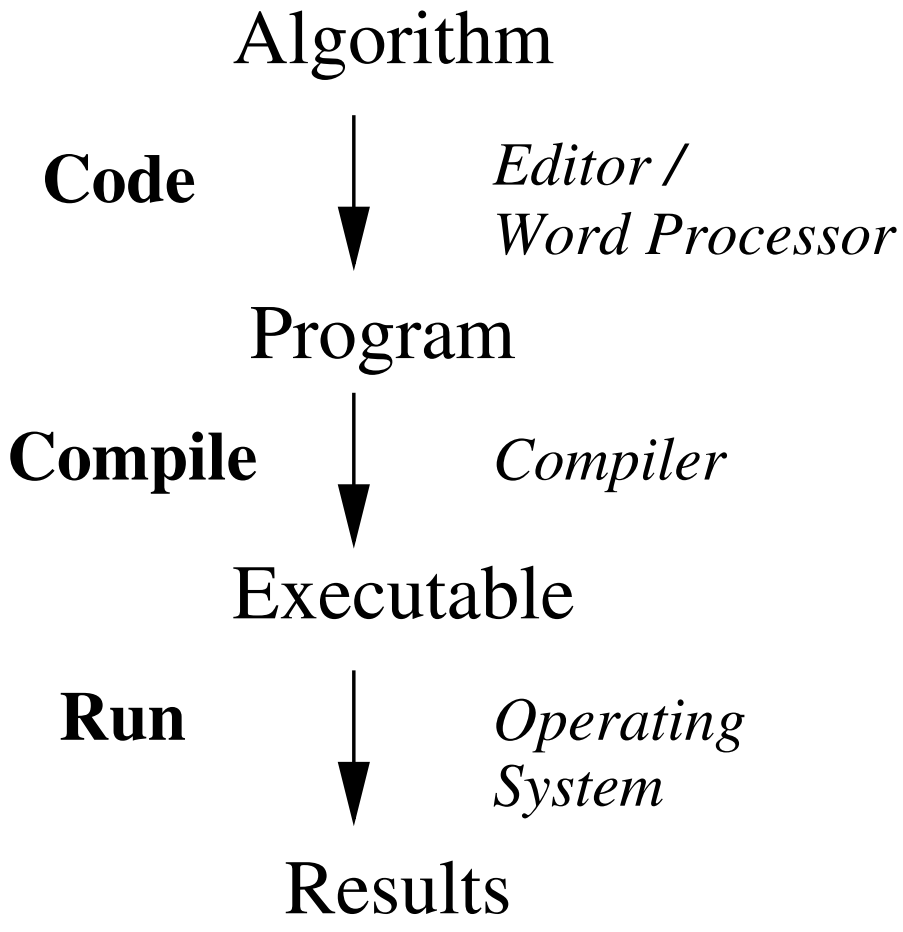
Input: A list L with n elements.
Output: The position of the largest-valued element in L.

Problem #3:

Input: A list L with n elements.
Output: All positions at which the largest-valued element in L occurs.

MANY POSSIBLE VARIANTS OF A PROBLEM;
MAKE SURE YOU SOLVE THE ONE YOU NEED TO.

Problems, Algorithms, and Programs: The Big Picture



Problems, Algorithms, and Programs: The Big Picture (Cont'd)

- In programming, there are many choices:
  - There are many variants of a problem
  - There are many algorithms solving a specific problem.
  - There are many programs implementing a specific algorithm.
- There is no universal best choice of algorithm and program for a problem, but rather several more or less appropriate choices given operational constraints, e.g., runtime / memory efficiency, exact vs. approximate solutions.
- Figuring out both the set of possible choices and the most appropriate of these choices for a particular situation is a big part of both programming and Computer Science.

Programming Languages: An Overview

- A programming language is defined by the valid statements in that language (**syntax**) and what those statements do (**semantics**).
- A programming language can be **compiled** (whole program translated into machine language) or **interpreted** (individual program-statements translated as needed).
- Machine-independence achieved formally by standards, e.g., ANSI, IEEE, and implemented in practice by intermediate languages, e.g., **bytecode**.
- Machine-independence is often violated, e.g., may exploit particular machines and/or modify language features; additional incompatible variants may arise as language evolves over time, e.g., Python 2.x vs. Python 3.x.

Programming Languages: An Overview (Cont'd)

Two reasons why there are many programming languages:

1. Languages are designed for different tasks, e.g.,
   - Scientific computation (FORTRAN)
   - Business applications (COBOL)
   - Web-page creation (HTML)
   - Database creation (SQL)
2. Languages are designed for different ways of thinking about programming, e.g.,
   - Procedural programming (FORTRAN, COBOL, C)
   - Object-oriented programming (OOP) (C++, Java)
   - Logic Programming (Prolog)
   - Script-based programming (Javascript, Ruby)

The Python Programming Language: Overview

- Created by Guido van Rossum in 1991 as an easy-to-learn general-purpose programming language.
- Procedural scripting language that allows but does not require OOP ("as OOP as you wanna be").
- Key design principles:
  - Control structure indicated by indentation.
  - Powerful built-in data types.
  - Any variable can refer to any type of data, and this type can change as a program executes.
- Primarily interpreted but can be compiled for speed.
- General machine-independence achieved by bytecode; however, Python 3.x not directly backward-compatible with Python 2.x.

Programming in Python I: A First Example Program

```
1.  # Example program; adapted from
2.  # Online Python Supplement, Figure 1.2
3.
4.  speed = input("Enter speed (mph):  ")
5.  speed = int(speed)
6.  distance = input("Enter distance (miles):  ")
7.  distance = float(distance)
8.
9.  time = distance / speed
10.
11. print("At", speed, "mph, it will take")
12. print(time, "hours to travel", \
13.       distance, "miles.")
```

Programming in Python I: A First Example Program (Cont'd)

- Python programs are stored in files with extension `.py`, e.g., `example1.py`.
- When this program is executed using a Python interpreter and the user enters the boldfaced values, this is printed:

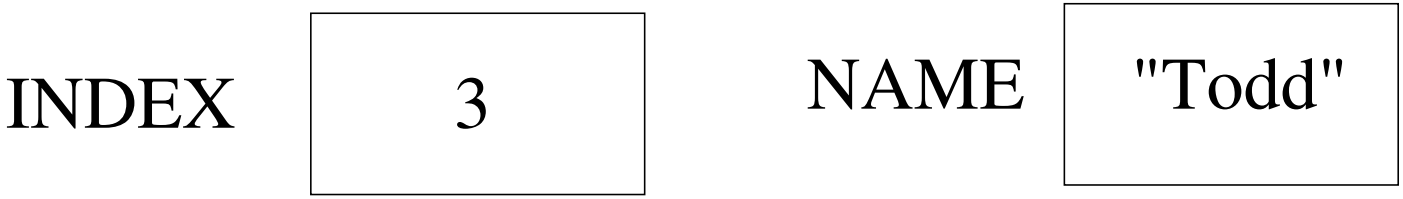
```
Enter speed (mph):  58
Enter distance (miles):  657.5
At 58 mph it will take
11.3362068966 hours to travel 657.5 miles.
```

Programming in Python I: A First Example Program (Cont'd)

- Line numbers not necessary; are given here to allow easy reference to program lines.
- Lines beginning with hash (#) are **comments** (Lines 1-2); a **prologue comment** at the top of the program gives a program's purpose and creation / modification history.
- Comment and blank lines (Lines 3, 8, and 10) are ignored.
- Each line is a program statement; multiline statements are linked by end-of-line backslashes (\) (Lines 12-13).
- No variable-type declaration statements; this is handled by **assignment statements** (Lines 4-7 and 9).
- This program also has basic **I/O statements** (Lines 4, 6, and 11-13); **control statements** will be shown later.

Programming in Python I: Assignment Statements

- General form: *variable = expression*, e.g.,
  - `index = 1`
  - `myDistanceRate = curDistanceRate * 1.75`
  - `name = "Todd Wareham"`
  - `callList = ["Bob", "Sue", "Anne"]`
- A **variable** is a stored data location that can hold a data value, e.g., variables INDEX and NAME with values 3 and "Todd", respectively.:



Programming in Python I: Assignment Statements (Cont'd)

- In Python, an assignment statement sets the value of *variable* to the value of *expression*.
  - If *variable* did not already exist, it is created.
  - If *variable* did already exist, its previous value is replaced. Note that the data-type of this previous value need not be that created by *expression*.
- Variable names (also called **identifiers**) can be arbitrary sequences of letters, numbers and underscore symbols (_) such that (1) the first symbol is a letter and (2) the sequence is not already used in the Python language.
- Python is case-sensitive wrt letter capitalization, e.g., `myList` is a different variable than `mylist`.

Programming in Python I: Assignment Statements (Cont'd)

- By convention, constants use only upper-case letters and numbers, e.g., `PI, TYPE1COLOR`.
  - Though constants should not change value, they are still technically variables, e.g.,

```
...
PI = 3.1415927
...
PI = -1
...
```

    It is up to programmers to make sure that such changes do not happen.
- Underscores reserved for Python system constants.

Programming in Python I: Assignment Statements (Cont'd)

- The `int` and `float` data-types
  - Encode "arbitrary" integers, e.g., `-1001, 0, 57`, and floating-point numbers, e.g. `-100.2, 3.1415927`.
  - Support basic arithmetic operations (`+, -, *, /`); also have floor-division (`//`) and remainder (`%`) operations, e.g.,

    `7 / 2` $\Longrightarrow$ 3.5
    `7 // 2` $\Longrightarrow$ 3
    `7 % 2` $\Longrightarrow$ 1

    Behaviour of `/` incompatible with Python 2.x.
  - Many additional math functions and constants available in the `math` module, e.g., `abs(x)`, `pow(base, exponent)`, `sqrt(x)`, `pi`.

Programming in Python I: Assignment Statements (Cont'd)

```
radius = input("Enter radius:  ")
radius = float(radius)
area = 3.1415927 * radius * radius
print("Circle Area = ", area)
```

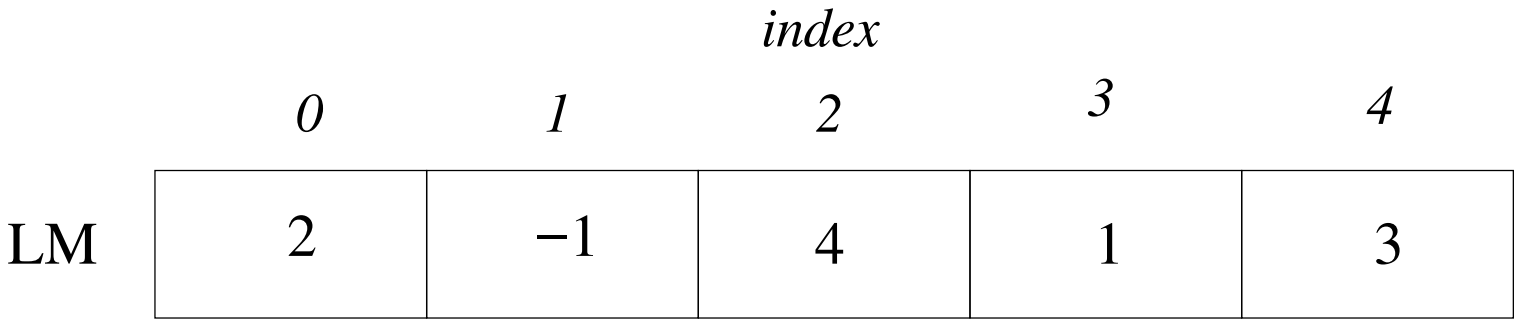
```
import math

radius = input("Enter radius:  ")
radius = float(radius)
area = math.pi * math.pow(radius, 2)
print("Circle Area = ", area)
```

Programming in Python I: Assignment Statements (Cont'd)

- The `str` data-type
  - Encodes "arbitrary" character strings, e.g., `"657.5"`, `"Todd Wareham"`.
  - Supports many operations, e.g.,
    - Concatenation (+) (`"Todd" + " " + "Wareham"` $\Longrightarrow$ `"Todd Wareham"`)
    - Lower-casing ("Todd".lower() $\Longrightarrow$ "todd")
    - Upper-casing ("Todd".upper() $\Longrightarrow$ "TODD")
- Convert between data types using **type casting** functions, e.g., `float("657.5")` $\Longrightarrow$ `657.5`, `int(657.5)` $\Longrightarrow$ `657`, `str(58)` $\Longrightarrow$ `"58"`.

Programming in Python I: Assignment Statements (Cont'd)

- The `list` data-type
  - A **list** is a sequence of some number of stored data locations, e.g., a list LM with 5 data locations:



  - Each data location in a list has its own index, and that location is accessed using that index, e.g., LM[1] and LM[3] have values -1 and 1, respectively.

Programming in Python I: Assignment Statements (Cont'd)

- The `list` data-type (Cont'd)
  - In Python, lists can encode values of arbitrary data types, e.g., `[22, 5, 13, 57, -1]`, `["Bob", "Sue", "Anne"]`, `[1, "Bob", 3]`.
  - Supports many operations, e.g.,
    - Number of values in list (`len(L)`)
    - Append x to right end of list (`L.append(x)`)
    - List sorting (`L.sort()`)
    - Get list maximum value (`max(L)`)

Programming in Python I: I/O Statements

- Keyboard input done via `input(string)`.
  - Prints `string` on screen, waits for user to enter input followed by a key return, and then returns this input-string.
  - Input-string can be converted as necessary by type-casting functions, e.g., `float(radius)`.
- Screen output done via `print(plist)`.
  - Comma-separated items in `plist` converted to strings as necessary and concatenated, and resulting string printed.
  - By default, each `print`-statement prints one line; can override this by making `end = " ")` the last item.
  - Can include escape characters to modify printout, e.g., `\t` (tab), `\n` (newline),
- Above I/O incompatible with Python 2.x.

Programming in Python I: I/O Statements (Cont'd)

The statements

```
print("Here is \t a weird")
print("way \n of printing ", end = " ")
print("this message.")
```

print(out)

```
Here is       a weird
way
of printing this message.
```

Programming in Python I: A First Example Program Redux

```
1.  # Example program; adapted from
2.  # Online Python Supplement, Figure 1.2
3.
4.  speed = input("Enter speed (mph):  ")
5.  speed = int(speed)
6.  distance = input("Enter distance (miles):  ")
7.  distance = float(distance)
8.
9.  time = distance / speed
10,
11. print("At", speed, "mph, it will take")
12. print(time, "hours to travel", \
13.       distance, "miles.")
```

Programming in Python I: Control Statements

- Sequential Statements (**Statement Block**):
  - A set of statements with the same indentation.
  - All Python programs seen so far are purely sequential.
- Conditional Statements:
  - General form:

```
if (CONDITION1):
    ⟨ CONDITION1 Block⟩
elif (CONDITION2):
    ⟨ CONDITION2 Block⟩
        ...
else:
    ⟨ ELSE Block⟩
```

  - `elif` and `else` blocks are optional.

Programming in Python I: Control Statements (Cont'd)

Conditions typically based on variable-comparisons, possibly connected together by logical operators.

`x == y`	x equal to y
`x != y`	x not equal to y
`x < y`	x less than y
`x <= y`	x less than or equal to y
`x > y`	x greater than y
`x >= y`	x greater than or equal to y
`E1 and E2`	logical AND of $E1$ and $E2$
`E1 or E2`	logical OR of $E1$ and $E2$
`not E1`	logical NOT of $E1$

Programming in Python I: Control Statements (Cont'd)

```
if ((number % 2) == 0):
    print("number is even")
```

```
if ((number >= 1) and (number <= 10)):
    print("number in range")
```

```
if (1 <= number <= 10):
    print("number in range")
```

```
if not (1 <= number <= 10):
    print("number not in range")
```

Programming in Python I: Control Statements (Cont'd)

```
if ((number % 2) == 0):
    print("number is even")
else:
    print("number is odd")
```

```
if (number < 10):
    print("number less than 10")
elif (number == 10):
    print("number equal to 10")
else:
    print("number greater than 10")
```

Programming in Python I: Control Statements (Cont'd)

- Conditional Looping Statement:
  - General form:

    `while (`**CONDITION**`):`
    ⟨ **Loop Block** ⟩

  - Executes **Loop Block** as long as **CONDITION** is True.
- Iterated Looping Statement:
  - General form:

    `for x in` **LIST**`:`
    ⟨ **Loop Block** ⟩

  - Executes **Loop Block** for each item x in LIST.

Programming in Python I: Control Statements (Cont'd)

Print the numbers between 1 and 100 inclusive:

```
number = 1
while (number <= 100):
    print(number)
    number = number + 1
```

```
for number in range(1, 101):
    print(number)
```

Programming in Python I: Control Statements (Cont'd)

Sum the numbers in a -1-terminated list:

```
sum = 0
number = int(input("Enter number:  "))
while (number != -1):
    sum = num + number
    number = int(input("Enter number:  "))
print("Sum is ", sum)
```

Programming in Python I: Control Statements (Cont'd)

Find the maximum value in a -1-terminated list:

```
maxValue = -99
number = int(input("Enter number:  "))
while (number != -1):
   if (number > maxValue):
      maxValue = number
   number = int(input("Enter number:  "))
print("Maximum value is ", maxValue)
```

Programming in Python I: Control Statements (Cont'd)

Store the values in a -1-terminated list in L:

```
L = []
number = int(input("Enter number:  "))
while (number != -1):
    L.append(number)
    number = int(input("Enter number:  "))
```

Programming in Python I: Control Statements (Cont'd)

Print the values in list L (one per line) [Version #1]:

```
for i in range(0, len(L)):
    print(L[i])
```

Print the values in list L (one per line) [Versionj #2]:

```
for number in L:
    print(number)
```

Algorithms and Pseudocode

- Natural language not precise enough to express algorithms as it often relies on context and background knowledge; programming languages too precise for abstract thinking needed during algorithm development.
- **Pseudocode** = programming-language-like statement of algorithm that does not run on a computer.
- No standard way of writing pseudocode – anything goes, as long as the result satisfies the definition of an algorithm.
- In our pseudocode, we will have variables and lists as in Python; however, in accordance with our textbook, lists will start at index 1 rather than 0 and list-location indices will be denoted by subscripts, e.g., LM_2, LM_{INDEX}.

Algorithms and Pseudocode: Sequential Operations

Assignment: Set the value of a variable to something, e.g.,

Set the value of INDEX to INDEX + 1
Add 1 to INDEX
INDEX = INDEX + 1

Input: Get variable values from the outside world, e.g.,

Get the values for X and list L
Read in the values for X and list L

Output: Send some message (such as computed variable values) to the outside world, e.g.,

Print the value of SUM
Print the message "not found"
Return the value of SUM

Algorithms and Pseudocode: Conditional Operations

IF-THEN: Do something if some condition is true, e.g.,

```
if (L_INDEX is X) then
    Set the value of FOUND to "YES"
    Print the message "found"
```

IF-THEN-ELSE: Do first thing if some condition is true and if not do second thing, e.g.,

```
if (L_INDEX is X) then
    Set the value of FOUND to "YES"
    Print the message "found"
else
    Set the value of INDEX to INDEX + 1
```

Algorithms and Pseudocode: Conditional Operations (Cont'd)

Multiple IF-THEN-ELSE: Do something depending on which of a set of conditions is true and if none of the conditions are true do something else, e.g.,

```
if (L_INDEX is "apple" or "orange") then
    NUMFRUIT = NUMFRUIT + 1
else if (L_INDEX is "potato") then
    NUMVEG = NUMVEG + 1
else if (L_INDEX is "thyme") then
    NUMHERB = NUMHERB + 1
else
    print "unrecognized produce item"
```

Algorithms and Pseudocode: Iterative Operations

Conditional Iteration: Repeat something as long as a condition is true, e.g.,

Set the values of INDEX to 1 and SUM to 0
while (INDEX $\leq$ n) do
    Set the value of SUM to SUM + L_{INDEX}
    Set the value of INDEX to INDEX + 1

Counted Iteration: Repeat something as long as a condition is true, e.g.,

Set the value of SUM to 0
for INDEX = 1 to n do
    Set the value of SUM to SUM + L_{INDEX}

Example Algorithms: Overview

- Every algorithm has at least one underlying intuition which is subsequently refined into an algorithm proper.
- Let's look at how some classic algorithms are derived from their underlying intuitions.
- These algorithms are for the following problems:
  1. List Search
  2. List Maximum Value
  3. List Sorting
  4. Bin Packing

The List Search Problem

LIST SEARCH
Input: A list L with n elements and a value X.
Output: The position of the element with value X in L if such an element exists and -1 otherwise.

- Has many applications, e.g., looking up a person's telephone number, charging some amount to a credit card, finding out if anyone won the Lotto Max jackpot this week.
- For simplicity, assume X and all list-elements are numbers; however, our algorithms work for any values that can be ordered, e.g., words, names.

Sequential List Search: Intuition

"Well, if I don't know anything else about the list except that it has n elements, I suppose I'll have to look at each element in the list and see if it is equal to the target-value. If I find such an element, I can stop and save that element's position; otherwise, I return -1 after I've looked at all elements in the list. Sounds like a lot of work. Bummer."

Sequential List Search: Algorithm (Version 0)

Get values for X, list L, and n
Set the value of INDEX to 1 and FOUND to "NO"
While (FOUND is "NO") and (INDEX $\leq$ n) do
    If L_{INDEX} is X then
        Set the value of FOUND to "YES"
        Print the message “found”
    Else
        Set the value of INDEX to INDEX + 1
If (FOUND is "NO") then
    Print the message “not found”

Sequential List Search: Algorithm (Version 1)

Get values for X, list L, and n
Set the value of FOUNDPOS to -1 and INDEX to 1
while (FOUNDPOS is -1) and (INDEX $\leq$ n) do
    if (L_{INDEX} is X) then
        Set the value of FOUNDPOS to INDEX
    else
        Set the value of INDEX to INDEX + 1
return FOUNDPOS

Sequential List Search: Algorithm (Version 2)

Get values for X, list L, and n
FOUNDPOS = -1
INDEX = 1
while (FOUNDPOS is -1) and (INDEX $\leq$ n) do
    if (L_{INDEX} is X) then
        FOUNDPOS = INDEX
    else
        INDEX = INDEX + 1
return FOUNDPOS

Binary List Search: Intuition

"Hmmm ... Suppose this time I know L is sorted. Whenever I look at L_{INDEX} where INDEX is the middle of the list and L_{INDEX} is not equal to the target-value, as L is sorted, I know that the target-value must be either above or below INDEX in the list (depending on whether the target-value is greater or less than L_{INDEX}). I can keep repeating this in a loop until I either find the target-value or run out of list to search. This should finish way faster because each time I halve the size of the list I'm looking at. Cool!"

Binary List Search: Algorithm (Version 0)

```
Get values for X, list L, and n
Set the current list to all of L
while we haven't found X in list L and
  there's still a current list to search do
      if X isn't the middle element of the current list then
            if X > middle element then
                  set current list to upper part of current list
            else
                  set current list to lower part of current list
```

Binary List Search: Algorithm (Version 1)

```
Get values for X, list L, and n
FOUNDPOS = -1
LEFT = 1
RIGHT = n
while (FOUNDPOS is -1) and (LEFT ≤ RIGHT) do
    FOUNDPOS = (LEFT + RIGHT) / 2
    if (L_FOUNDPOS is not equal to X) then
        if (X > L_FOUNDPOS) then
            LEFT = FOUNDPOS + 1
        else
            RIGHT = FOUNDPOS - 1
        FOUNDPOS = -1
return FOUNDPOS
```

The List Maximum Value Problem

LIST MAXIMUM VALUE
Input: A list L with n elements.
Output: The position of the largest-valued element in L.

- Has many applications, e.g., looking for the employer that pays the highest salary for a particular job; is also a useful building block in more complex algorithms, e.g., list sorting.
- For simplicity, assume all list-elements are numbers; however, our algorithm work for any values that can be ordered, e.g., words, names.
- Can be readily adapted to find smallest list values.

List Maximum Search: Intuition

"Well, if I don't know anything else about the list except that it has n elements, I suppose I'll have to look at each element in the list and keep track as I go of which element is the largest I've found so far. After I've gone through this list, the largest I found by then is the largest in the list. Again, sounds like a lot of work. Bummer."

List Maximum Search: Algorithm (Version 0)

Get values for list L and n
Set the values of LARGEST to L_1, FOUNDPOS to 1, and
  INDEX to 2
While (INDEX $\leq$ n) do
    If $L_{INDEX} >$ LARGEST then
        Set the value of LARGEST to L_{INDEX}
        Set the value of FOUNDPOS to INDEX
    Set the value of INDEX to INDEX + 1
Print the value of FOUNDPOS

List Maximum Search: Algorithm (Version 1)

Get values for list L and n
FOUNDPOS = 1
INDEX = 2
while (INDEX $\leq$ n) do
    if $L_{INDEX} > L_{FOUNDPOS}$ then
        FOUNDPOS = INDEX
    INDEX = INDEX + 1
return FOUNDPOS

List Maximum Search: Algorithm (Version 2)

Get values for list L and n
FOUNDPOS = 1
for INDEX = 2 to n do
    If $L_{INDEX} > L_{FOUNDPOS}$ then
        FOUNDPOS = INDEX
return FOUNDPOS

The List Sorting Problem

LIST SORTING
Input: A list L with n elements.
Output: The version of L sorted in ascending value order.

- Has many applications, e.g., generating lists of employees by name or salary; also enables algorithms that require sorted list, e,g„ binary list search.
- For simplicity, assume all list-elements are numbers; however, our algorithms work for any values that can be ordered, e.g., words, names.
- Can be readily adapted to sort in descending value order.

Selection Sort: Intuition

"The last element in a sorted list is the largest in the list, the second-last element is the largest among the remaining elements in the list, and so on. Perhaps we could use a find-list-maximum algorithm in a loop!"

Selection Sort: Algorithm (Version 0)

Get values for list L and n
Set the marker for the unsorted section at the end of L
While the unsorted section is not empty do
    Find largest element in unsorted section of list
    Swap this largest element with the last element in the unsorted part of the list
    Move the marker for the unsorted section left one position

Selection Sort: Algorithm (Version 1)

Get values for list L and n
ENDUNSORTED = n
While (ENDUNSORTED $>$ 1) do
    FOUNDPOS = 1
    for INDEX = 2 to ENDUNSORTED do
        If $L_{INDEX} > L_{FOUNDPOS}$ then
            FOUNDPOS = INDEX
    TMP = $L_{ENDUNSORTED}$
    $L_{ENDUNSORTED} = L_{FOUNDPOS}$
    $L_{FOUNDPOS}$ = TMP
    ENDUNSORTED = ENDUNSORTED - 1

Selection Sort: Algorithm (Version 2)

Get values for list L and n
for ENDUNSORTED = n downto 2 do
    FOUNDPOS = 1
    for INDEX = 2 to ENDUNSORTED do
        If $L_{INDEX} > L_{FOUNDPOS}$ then
            FOUNDPOS = INDEX
    TMP = $L_{ENDUNSORTED}$
    $L_{ENDUNSORTED}$ = $L_{FOUNDPOS}$
    $L_{FOUNDPOS}$ = TMP

Bubble Sort: Intuition

"In order for a list to be unsorted, there must be at least one pair of adjacent list-elements L_{INDEX} and $L_{INDEX+1}$ such that $L_{INDEX} > L_{INDEX+1}$. Suppose we kept going through the list, swapping bad adjacent list-element pairs until there weren't any more such pairs?"

Bubble Sort: Algorithm (Version 0)

```
Get values of list L
while list is not sorted do
    traverse L, swapping bad adjacent list-element pairs
      as necessary
    if no swaps occurred then
        the list is sorted
```

Bubble Sort: Algorithm (Version 1)

Get values of list L and n
SORTED = "NO"
while (SORTED is "NO") do
    NUMSWAP = 0
    for INDEX = 2 to n do
        if ($L_{INDEX-1} > L_{INDEX}$) then
            TMP = $L_{INDEX-1}$
            $L_{INDEX-1}$ = L_{INDEX}
            L_{INDEX} = TMP
            NUMSWAP = NUMSWAP + 1
    if (NUMSWAP is 0) then
        SORTED = "YES"

Bubble Sort: Algorithm (Version 2)

Get values of list L and n
SORTED = "NO"
while (SORTED is "NO") do
    SORTED = "YES"
    for INDEX = 2 to n do
        if ($L_{INDEX-1} > L_{INDEX}$) then
            TMP = $L_{INDEX-1}$
            $L_{INDEX-1}$ = L_{INDEX}
            L_{INDEX} = TMP
            SORTED = "NO"

The Bin Packing Problem

BIN PACKING
Input: A bin size B and a list L of n item sizes, each $\leq B$.
Output: The smallest number of bins of size B that can hold all of the items in L.

- Has many applications, e.g., minimizing order packaging for online retailers.
- Has a set of candidate solutions (packings of the items of L into bins), each with their own cost (number of bins in a packing), and requires a candidate solution that optimizes that cost; hence, this is an **optimization problem**.

Bin Packing: Intuitions

Intuition #1: "If I have at most n items, I'll need at most n bins. How about I try all possible ways of dividing the items in L among n or less bins, and then check each packing to make sure that no bin has items that are too big for that bin?"

Intuition #2: "Intuition #1 sounds way too hard. How about I just do it like Doug at Sobey's – take each item in L in turn and add it to the current bin, and if that item is too large, make a new bin and add it to that one?"

Assessing Algorithm Efficiency (Take I)

How many list-item comparisons (as a function of list-length n) do each of these algorithms require in the best and worst case?

Algorithm	Best	Worst
Sequential List Search	1	n
Binary List Search	1	$\log_2 n$
List Maximum Search	n	n
Selection Sort	$\approx n^2$	$\approx n^2$
Bubble Sort	$n-1$	$\approx n^2$

Given one wants to do m searches in a list of length n, for what values of m does it make sense to sort a list?

Does this change if we know that a given list is almost sorted, e.g., has $k << n$ unsorted items at the front of the list?

Programming in Python II: Functions

- Compartmentalize data, tasks, and algorithms in programs with **functions**; allow implementation of divide-and-conquer-style programming.
- General form:

```
def funcName():
    ⟨ Function Block ⟩

def funcName(parameterList):
    ⟨ Function Block ⟩

def funcName(parameterList):
    ⟨ Function Block ⟩
    return value
```

Programming in Python II: Functions (Cont'd)

- A variable defined inside a function is a **local variable**; otherwise, it is a **global variable**.
- If a local variable has the same name as a global variable, the local variable is used inside the function.
- What does this print?

```
def myFunc1():
    one = -1
    print(one, two)

one = 1
two = 2
print(one, two)
myFunc1()
print(one, two)
```

Programming in Python II: Functions (Cont'd)

- The parameters in a function's parameter-list match up with and get their values from the arguments in the argument-list of a function call in numerical order, not by parameter / argument name.
- What does this print?

```
def myFunc2(one, two, three):
    print(one, two, three)

one = 1
two = 2
three = 3
print(one, two, three)
myFunc2(two, three, one)
print(one, two, three)
```

Programming in Python II: Functions (Cont'd)

- The value returned by a function can be captured by an assignment statement which has that function as the expression.
- What does this print?

```
def myFunc3(one, two, three):
    sum = (one + two) - three
    return sum

one = 1
two = 2
three = 3
result = myFunc3(two, three, one)
print(result)
```

Programming in Python II: Functions (Cont'd)

- Eliminate global variables with **main functions**.
- What does this print?

```
def myFunc4(one, two, three):
    sum = (one + two) - three
    return sum

def main():
   one = 1
   two = 2
   three = 3
   result = myFunc4(two, three, one)
   print(result)

main()
```

Programming in Python II: Functions (Cont'd)

Functions useful in all stages of software development:

1. Planning (View complex problem as set of simple subtasks)
2. Coding (Code individual subtasks independently)
3. Testing (Test individual subtasks independently)
4. Modifying (Restrict changes to individual subtasks)
5. Reading (Understand complex problem as set of simple subtasks)

Programming in Python II: Functions (Cont'd)

Reading in and printing a -1-terminated list (Version #1):

```
L = []
number = int(input("Enter number:  "))
while (number != -1):
    L.append(number)
    number = int(input("Enter number:  "))
for number in L:
    print(number)
```

Programming in Python II: Functions (Cont'd)

Reading in and printing a -1-terminated list (Version #2):

```
def readList():
    L = []
    number = int(input("Enter number:  "))
    while (number != -1):
        L.append(number)
        number = int(input("Enter number:  "))

def printList():
    for number in L:
        print(number)

readList()
printList()
```

Programming in Python II: Functions (Cont'd)

Reading in and printing a -1-terminated list (Version #3):

```
def readList():
    number = int(input("Enter number:  "))
    while (number != -1):
        L.append(number)
        number = int(input("Enter number:  "))

def printList():
    for number in L:
        print(number)

L = []
readList()
printList()
```

Programming in Python II: Functions (Cont'd)

Reading in and printing a -1-terminated list (Version #4):

```
def readList():
    L = []
    number = int(input("Enter number:  "))
    while (number != -1):
        L.append(number)
        number = int(input("Enter number:  "))
    return L

def printList(L):
    for number in L:
        print(number)

L = readList()
printList(L)
```

Programming in Python II: Functions (Cont'd)

```
def readList():
    L = []
    number = int(input("Enter number:  "))
    while (number != -1):
        L.append(number)
        number = int(input("Enter number:  "))
    return L

def printList(L):
    for number in L:
        print(number)

def main():
    L = readList()
    printList(L)

main()
```

Programming in Python II: Functions (Cont'd)

Sort the values in list L (Selection Sort) (Function Version #1):

```
def sortList(L):
    endUnSort = len(L) - 1
    while (endUnSort > 0):
        maxPos = 0
        for ind in range(1, endUnSort + 1):
            if (L[ind] > L[maxPos]):
                maxPos = ind
        tmp = L[endUnSort]
        L[endUnSort] = L[maxPos]
        L[maxPos] = tmp
        endUnSort = endUnSort - 1
    return L
```

Programming in Python II: Functions (Cont'd)

Sort the values in list L (Selection Sort) (Function Version #2):

```
def sortList(L):
    endUnSort = len(L) - 1
    while (endUnSort > 0):
        maxPos = getListMaxPos(L, endUnSort)
        swap(L, maxPos, endUnsort)
        endUnSort = endUnSort - 1
    return L
```

Programming in Python II: Functions (Cont'd)

Compute unique values in sorted list L (Function):

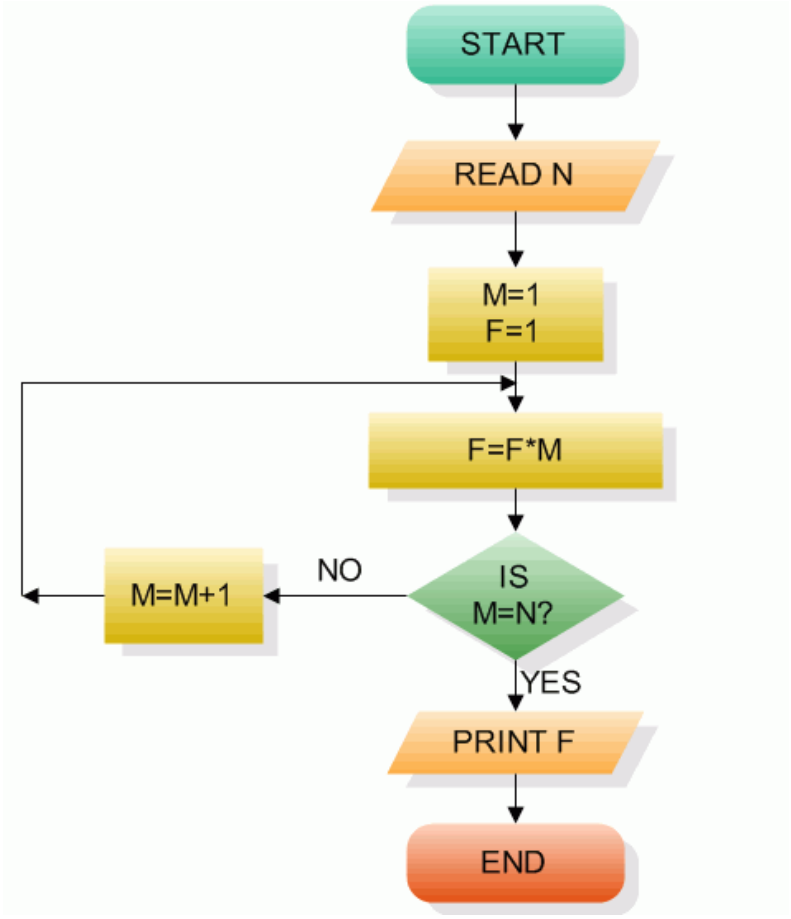
```
def getUniqueList(L):
    LUnique = []
    curValue = L[0]
    for ind in range(1, len(L)):
        if (L[ind] != curValue):
            LUnique.append(curValue)
            curValue = L[ind]
    LUnique.append(curValue)
    return LUnique
```

Programming in Python II: Functions (Cont'd)

Main function for unique-value list program:

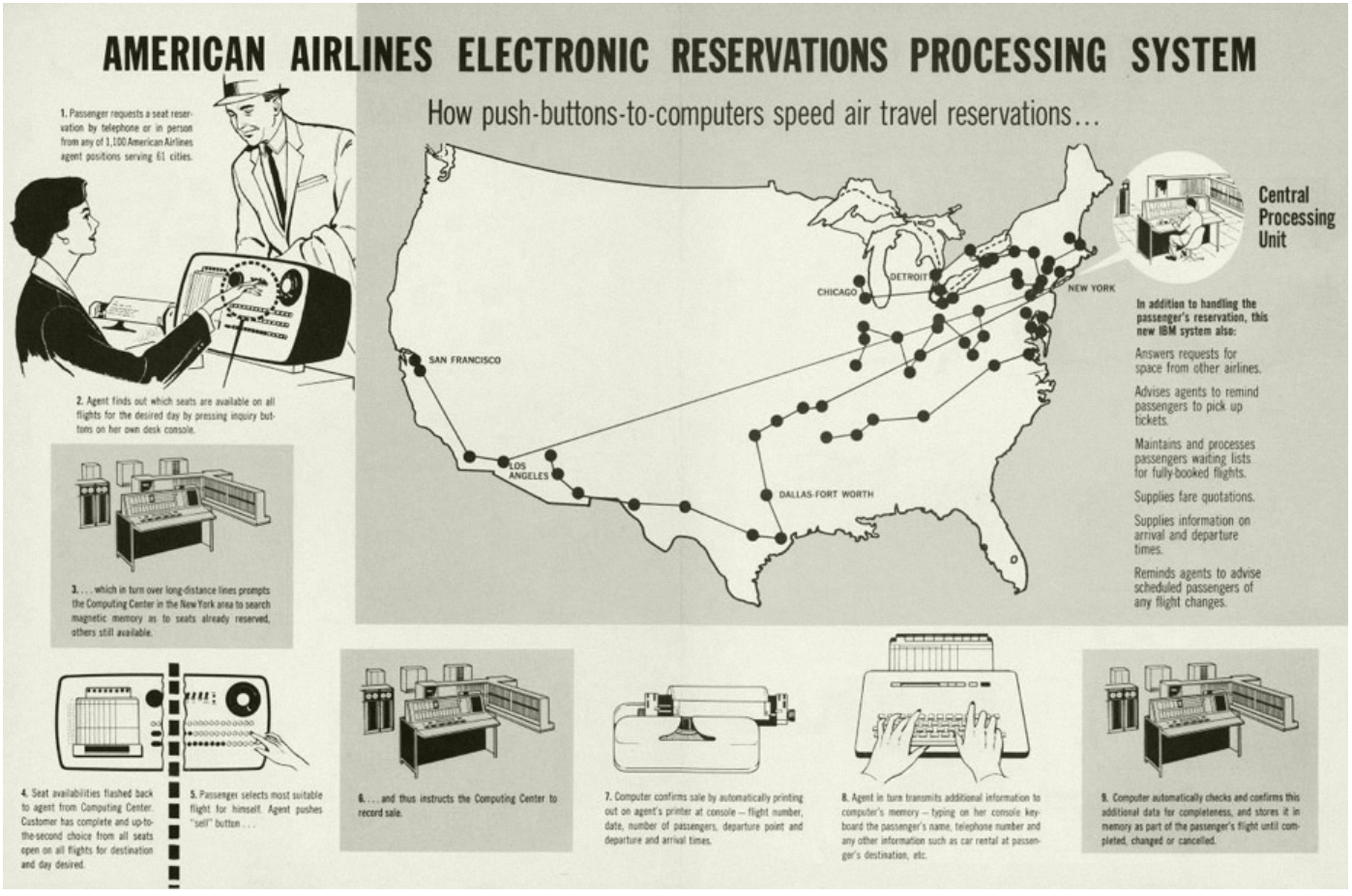
```
def main():
    L = readList()
    L = sortList(L)
    L = getUniqueList(L)
    printList(L)
```

Implementing Programming: The Software Crisis



- Act of programming made easier by compilers, languages, and operating systems; problem of developing algorithms remained.
- Special notations like flowcharts help with small- and medium-size programs; hope was that appropriate management would help with large ones.

Implementing Programming: The Software Crisis (Cont'd)



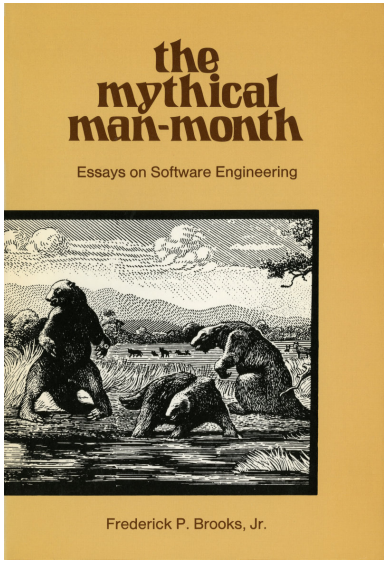
The SABRE Airline Reservation System (1964)

Implementing Programming: The Software Crisis (Cont'd)

IBM System/360 (1967)

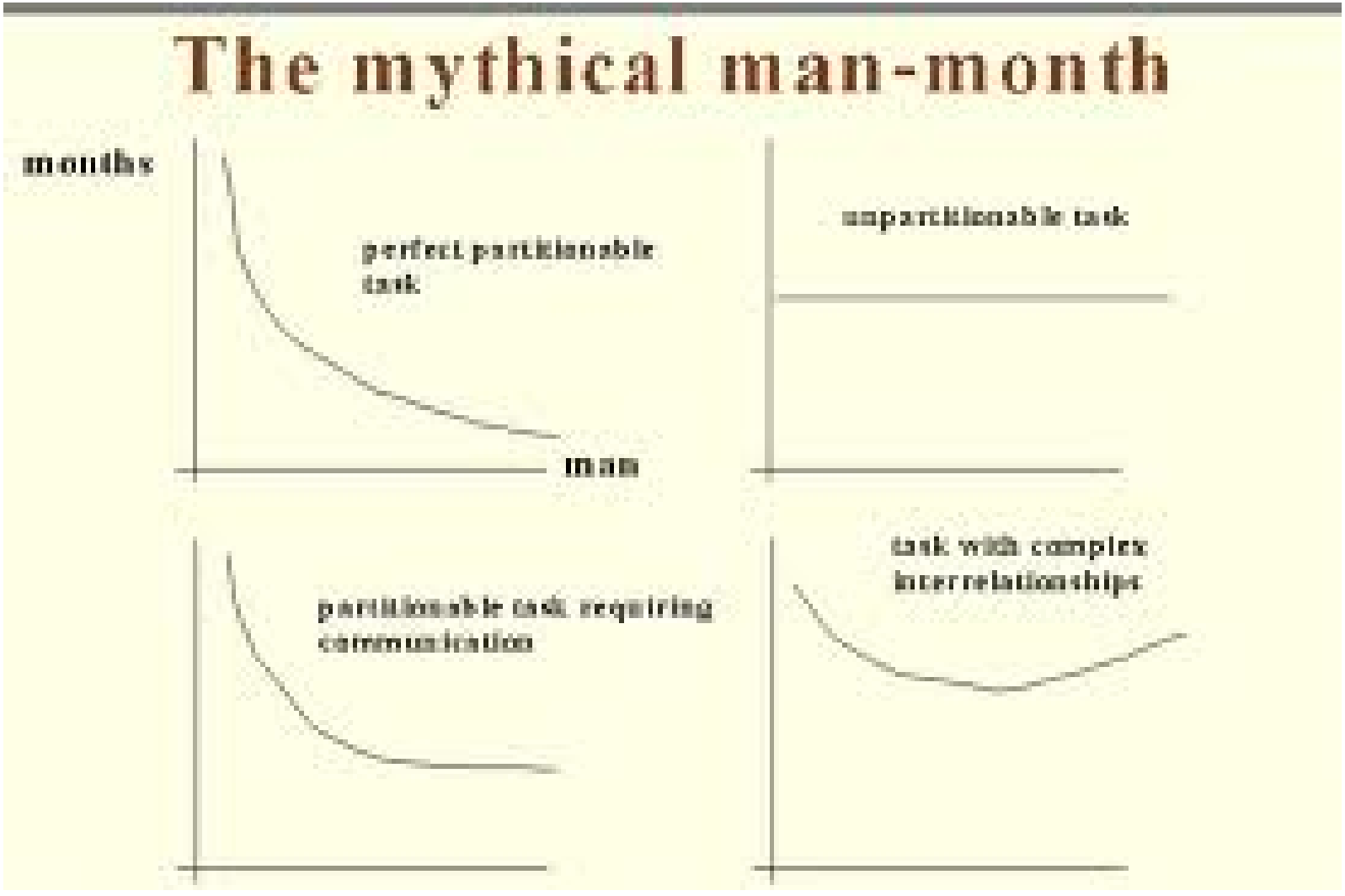
Implementing Programming: The Software Crisis (Cont'd)

Fred Brooks Jr.
(1931–)

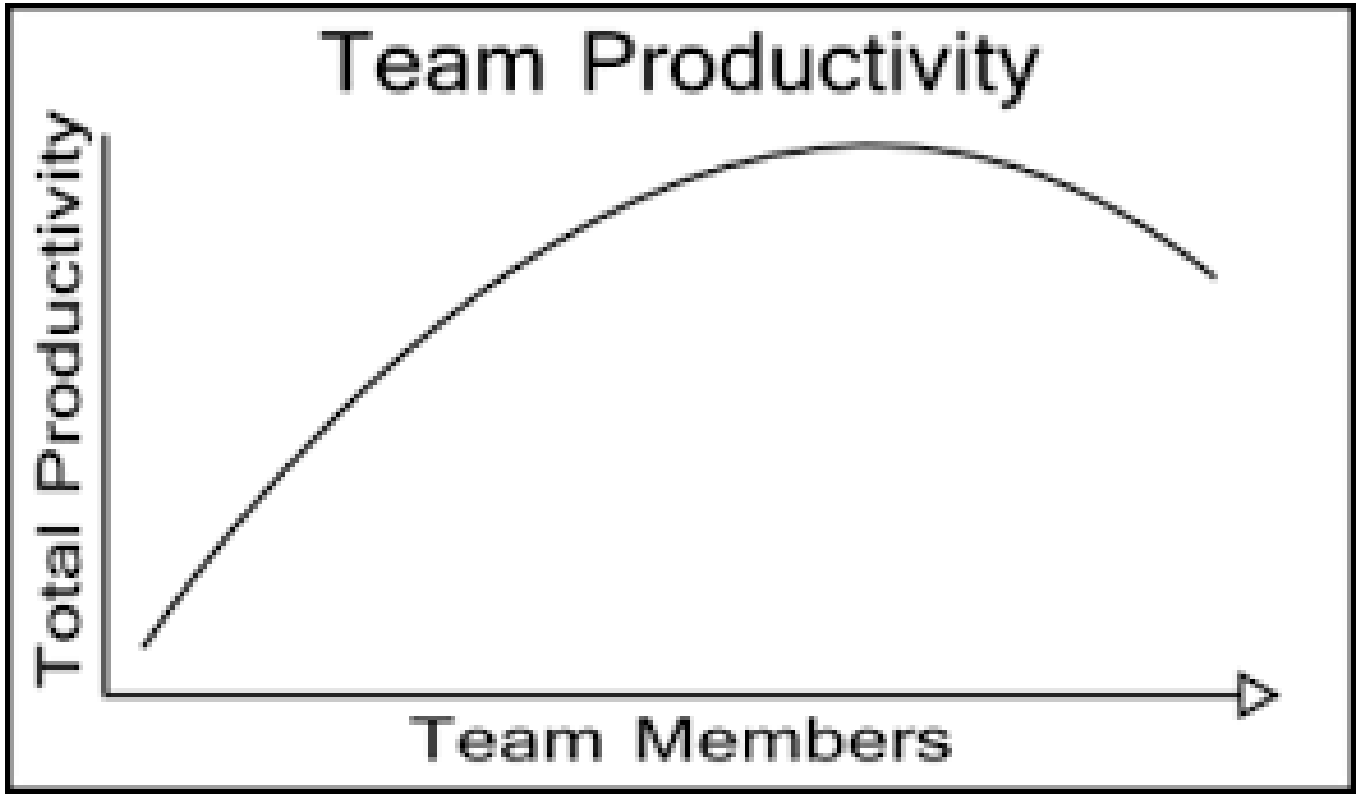


- OS/360 initially planned for 1965 costing $125M; limped to market in 1967 costing $500M, and virtually destroyed IBM's in-house programming division.
- Brooks discussed causes in *The Mythical Man Month*.

Implementing Programming: The Software Crisis (Cont'd)



Implementing Programming: The Software Crisis (Cont'd)



As both larger programs and larger teams have more complex internal relationships, adding more programmers to larger projects makes things *worse*.

Implementing Programming: The Software Crisis (Cont'd)

- Software Engineering born at 1968 NATO-sponsored conference; goal of SE is to develop efficient processes for creating and maintaining correct software systems.
- Many types of processes proposed, *e.g.*, design and management methodologies (Agile), automatic software derivation methods; however, "No Silver Bullet" (Brooks).

. . . And If You Liked This . . .

- MUN Computer Science courses on this area:
  - COMP 1001: Introduction to Programming
  - COMP 2001: Object-oriented Programming and HCI
  - COMP 2002: Data Structures and Algorithms
  - COMP 2005: Software Engineering
- MUN Computer Science professors teaching courses / doing research in in this area:
  - Ed Brown
  - Sharene Bungay
  - Adrian Fiech
  - Mark Hatcher
  - Amilcar Soares