Big Idea 3: Algorithms and Programming Part 2

Programming

Programs can be developed for creative expression, to satisfy personal curiosity, to create new knowledge, or to solve problems (to help people, organizations, or society). Advances in computing have generated and increased creativity in other fields. For example, new computer tools are being used to track dolphins and to decode their vocalization. Understanding complex dolphin behavior would not have been possible without the ability to data mine the acoustic dolphin database (searching through large databases of information to pinpoint relevant data).

Sometimes programs that are developed for personal use can be adapted to serve a larger audience and purpose. In 2004, Mark Zuckerberg created Facebook, a local platform for Harvard students to use to connect with each other. Before adapting his program to apply to a larger audience, it totaled about 1 million users by the end of 2004. Only eight years later, that number had increased to over 1 billion active users. This widespread distribution was made possible by changes to the program in its development.

Programs Can Be Flexible

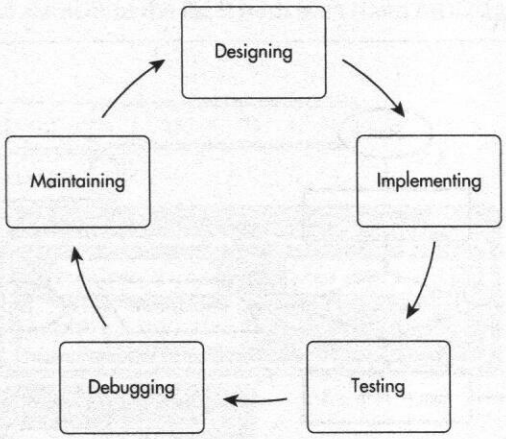
When it was targeted to a smaller, local audience simply to satisfy Zuckerberg's personal curiosity and enjoyment, Facebook was held to much different standards in its development. When it was made for billions of people to use, changes had to be made to accommodate so many people and to target it toward a wide market of people that might use it—making it appealing to a wider audience.

The technologies and programs that are applied to Facebook could also be applied to other fields. As well as inspiring the age of social media, the data-mining programs used to understand and process the data that are "harvested" from Facebook are the same type of programs that allow researchers to make sense of data gathered from the acoustic dolphin database.

Programming Design Steps

The first step in programming is planning and identifying programmer and user concerns that can affect the solution. Consultation and communication with program users are important aspects of program development to solve problems.

When designing a large program, an iterative process helps with correctly coding. Checking for errors in small chunks can make isolating errors more efficient. Once a small chunk of the program is error free, it can be combined with already-checked portions of the program to help create a larger, correct program. This process of designing, implementing, testing, debugging, and maintaining programs is repeated until the entire program is ready to be released.



Program Documentation

Program documentation is helpful in all stages of program development. Documentation does not slow down run speed and is necessary when collaborating or programming alone. Documentation is useful during initial program development and when modifications are made. In this book, the symbol // has been used to indicate documentation in programs.

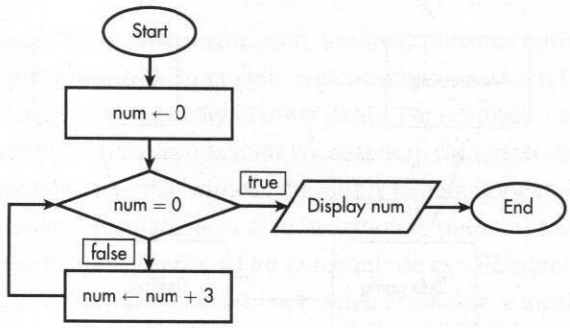
Flowcharts

A flowchart is a way to represent an algorithm visually. The flowcharts below use the following building blocks.

Block	Explanation
Oval	The start or end of the algorithm
Rectangle	One or more processing steps, such as a statement that assigns a value to a variable
Diamond	A conditional or decision step, where execution proceeds to the side labeled `true` if the condition is true and to the side labeled `false` otherwise
Parallelogram	Displays a message

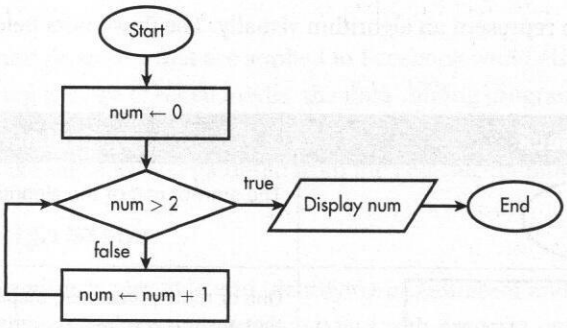
Selection statements determine which parts of an algorithm are executed based on a condition being true or false.

Example Forty-Eight



Step 1: Start the program.
Step 2: Set num = 0.
Step 3: If *n* equal to 0, evaluates to true.
Step 4: Display 0.
Step 5: End the program.

Example Forty-Nine



Step 1: Start the program.
Step 2: Set num = 0.
Step 3: If 0 greater than 2, evaluates to false.
Step 4: Set num equal to 0 + 1, which equals 1.
Step 5: Is 1 greater than 2, evaluates to false.
Step 6: Set num equal to 1 + 1, which equals 2.
Step 7: Is 2 greater than 2, evaluates to false.
Step 8: Set num equal to 2 + 1, which equals 3.
Step 9: Is 3 greater than 2, evaluates to true.
Step 10: Display 3.
Step 11: End the program.

Example Fifty

Conditional statements or "If" statements affect the sequential flow of control by executing different statements based on the value of a Boolean expression. In this example, `list2`

will be appended only if the element in the list is both even (item MOD 2 = 0) and odd (item MOD 2 = 1).

```
list1 ← 1, 1, 35, 6, 76, 4, 98
list2 ← _
FOR EACH item IN list1
    IF item MOD 2 = 0 AND item MOD 2 = 1
        APPEND list2, item
FOR EACH item IN list2
    DISPLAY item
```

Step 1 puts the following numbers [1, 1, 35, 6, 76, 4, 98] into a data structure called `list1`.
In step 2, the data structure called `list2` is set to empty.
Step 3 is a FOR EACH loop that will loop through each item contained in `list1`.
Step 4 means that if the item is both even AND odd, it will add the number to the end of `list2`.

- Item MOD 2 = 0 will evaluate to true if the number is even.
- Item MOD 2 = 1 will evaluate to true if the number is odd.

Step 5 shows that since a number can never be both even AND odd, no numbers will be appended to the end of `list2`.
Step 6 means that since `list2` remains empty, there is nothing to DISPLAY.

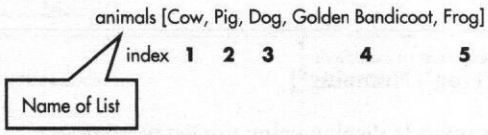
Lists

Lists are an organized and formatted way of storing and retrieving data. Each element in a list can be accessed by its index.

Unlike some common programming languages, indexes start at 1 on the AP exam, not 0. Trying to access an index that does not exist will result in an index out of bounds error.

Example Fifty-One

The data structure is filled with the strings called Cow, Pig, Dog, Golden Bandicoot, and Frog. Each string can be accessed using an index.



animals[0] = error. Index out of bounds.
animals[1] = Cow
animals[2] = Pig
animals[3] = Dog
animals[4] = Golden Bandicoot
animals[5] = Frog
animals[6] = error. Index out of bounds.

Example Fifty-Two

scores ← [11, 35, 6, 75, 37]
scores[1] = 11
scores[2] = 35
scores[3] = 6
scores[4] = 75
scores[5] = 37

Abstractions Provided on Your AP Exam for Lists

For all list operations, if a list index is less than 1 or greater than the length of the list, an error message is produced and the program terminates.

Text: `list[i]` Block: `list` [`i`]	Refers to the element of `list` at index `i`. The first element of list is at index 1.
Text: `list[i] ← list[j]` Block: `list` [`i`] ← `list` [`j`]	Assigns the value of `list[j]` to `list[i]`.
Text: `list ← [value1, value2, value3]` Block: `list ←` [`value1, value2, value3`]	Assigns `value1`, `value2`, and `value3` to `list[1]`, `list[2]`, and `list[3]`, respectively.

Text: `INSERT(list, i, value)` Block: `INSERT` [`list, i, value`]	Any values in `list` at indexes greater than or equal to `i` are shifted to the right. The length of list is increased by 1, and `value` is placed at index `i` in `list`.
Text: `APPEND(list, value)` Block: `APPEND` [`list, value`]	The length of `list` is increased by 1, and `value` is placed at the end of `list`.

Example Fifty-Three

words ← ["The", "Little", "Frog", "Jumping"]

What will the following commands display using the list words?

DISPLAY(words[2]) = Little
DISPLAY(words[5]) = Index Out Of Bounds
DISPLAY(words[1]) = The

Example Fifty-Four

```
words ← "The", "little", "Frog", "Jumping"
INSERT words, 3, "Green"
```

The data structure words will now contain the following after the INSERT method is used.
words["The", "Little", "Green", "Frog", "Jumping"]

Example Fifty-Five

```
words ← "The", "little", "Frog", "Jumping"
INSERT words, 2, "Green"
APPEND words, "Fox"
```

The data structure words will now contain the following after the INSERT and APPEND methods are used.
words["The", "Green", "Little", "Frog", "Jumping", "Fox"]

Example Fifty-Six

```
words ← ["The"]
INSERT(words, 1, "Green")
APPEND(words, "Fox")
APPEND(words, "Pig")
APPEND(words, "Rhino")
INSERT(words, 1, "Elephant")
```

The data structure words will now contain the following after the INSERT and APPEND methods are used.
words["Elephant", "Green", "The", "Fox", "Pig", "Rhino"]

Text:
`REMOVE(list, i)`
Block:
`REMOVE list`
	Removes the item at index `i` in `list` and shifts to the left any values at indexes greater than `i`. The length of `list` is decreased by 1.

Text:
`LENGTH(list)`
Block:
`LENGTH list`
	Evaluates to the number of elements in `list`.

Example Fifty-Seven

Line 1: words ← ["Elephant", "Green", "The", "Fox", "Pig", "Rhino", "Fox"]
Line 2: DISPLAY(LENGTH(words)) // answer 7
Line 3: REMOVE(words, 1)
Line 4: DISPLAY(LENGTH(words)) // answer 6

Traversing a List

Traversing a list means that you are accessing all the elements of the list one by one.

Example Fifty-Eight

```
scores ← 96, 93, 90, 100, 92, 90
For EACH item IN scores
    DISPLAY item
```

Output: 96 93 90 100 92 90

Example Fifty-Nine

```
scores ← 96, 93, 90, 100, 92, 90
n ← 0
REPEAT LENGTH scores TIMES
    DISPLAY LENGTH scores − n
    n ← n + 1
```

scores[length] = 6
n = 0
DISPLAY (scores[6-0]) // displays 90

scores[length] = 6
n = 1
DISPLAY (scores[6-1]) // displays 92

scores[length] = 6
n = 2
DISPLAY (scores[6-2]) // displays 100

scores[length] = 6
n = 3
DISPLAY (scores[6-3]) // displays 90

scores[length] = 6
n = 4
DISPLAY (scores[6-4]) // displays 93

scores[length] = 6
n = 5
DISPLAY (scores[1]) // displays 96

Total Output: 90 92 100 90 93 96

Procedures

A procedure is a set of code that is referred to by name and can be called (invoked) at any point in a program simply by utilizing the procedure's name. In some languages, a procedure could be called a *method* or *subroutine*.

Example Sixty

```
Line 1:  PROCEDURE doubling(list)
Line 2:  {
Line 3:    count ← 1
Line 3:    REPEAT LENGTH(list) TIMES
Line 4:    {
Line 5:      list[count] ← list[count] * 2
Line 6:      count ← count + 1
Line 6:    }
Line 7:  }
```

The purpose of this procedure is to double every value in the list

In step 1, the procedure "doubling" takes in a list as input.
Step 2 sets the value of the count to 1.
In step 3, this loop will repeat for the number of elements contained in the list.
Step 4 sets the element of list with the index of count equal to the value of the element times 2.
Step 5 adds 1 to the count.

A call to the procedure `doubling` passing the list numbers 2, 5, 11, 6] will result in the following.

numbers ←[2, 5, 11, 6]
DISPLAY(numbers[1]) // will display 2
DISPLAY(numbers[2]) // will display 5
DISPLAY(numbers[3]) // will display 11
DISPLAY(numbers[4]) // will display 6

doubling (numbers) // Calls the procedure doubling

DISPLAY(numbers[1]) // will display 4
DISPLAY(numbers[2]) // will display 10
DISPLAY(numbers[3]) // will display 22
DISPLAY(numbers[4]) // will display 12

Example Sixty-One

```
Line 1:  PROCEDURE keepPositive(alist, bList)
Line 2:  {
Line 3:    FOR EACH item IN aList
Line 4:    {
Line 5:      IF(item < 0)
Line 6:      APPEND(bList, item)
Line 7:    }
```

The purpose of the above procedure is to append all negative numbers in aList into bList.
Line 3 iterates through aList.
If the element in aList is less than 0, line 5 causes the element to be appended into bList.

A call to keepPositive(aList, bList) where aList[2, −5, −11, 6] and bList[ ] would result in bList containing the numbers [−5, −11].

Example Sixty-Two

Calling a Procedure

Calls the procedure addThree(num)

```
Line 1:  num ← 7

Line 2:  DISPLAY(addThree(num))

Line 3:  PROCEDURE addThree(num)

Line 4:  RETURN num + 3
```

Output

10

Example Sixty-Three

Finding Total Text-Based Coding
Standard algorithm used on AP exam

Calls the procedure findTotal(scores)

```
Line 1:  scores ← [90, 89, 98, 100, 90]

Line 2:  total ← findTotal(scores)

Line 3:  DISPLAY(total)

Line 4:

Line 5:  PROCEDURE findTotal(scores)

Line 6:  {

Line 7:  sum = 0

Line 8:  FOR EACH item IN scores

Line 9:  {

Line 10: sum ← sum + item

Line 11: }

Line 12: RETURN sum

Line 13: }
```

Output

467

Example Sixty-Four

Finding Average Text-Based Coding
Standard algorithm used on AP exam

```
Line 1:  scores ← [90, 90, 100, 100, 95]
Line 2:  ave ← findAverage(scores)
Line 3:  DISPLAY(ave)
Line 4:
Line 5:  PROCEDURE findAverage(list)
Line 6:  {
Line 7:  sum ← 0
Line 8:  count ← 1
Line 9:  n ← LENGTH(list)
Line 10: REPEAT n TIMES
Line 11: {
Line 12: sum ← sum + list(count)
Line 13: count ← count + 1
Line 14: }
Line 15: RETURN(sum/n)
Line 16: }
```

Output

95

Lines 7 and 8 initialize the variable sum to 0 and the variable count to 1.
Line 9 sets the variable *n* equal to the number of items in the list.
Line 10 repeats the next segment *n* times (for every item in the list).
Line 12 adds the item in position count from the list to the sum.
Line 13 is an increment, count by 1 (so that the next time you loop through, you will add the item in the next spot in the list).
Line 15 takes the sum of all the items and divides by *n*, which is the number of items in the list.

Code can be written in different ways and still achieve the same results. For example, the code for finding average could also be written using a For Each loop and achieve the same results with different syntax. Compare the coding in examples sixty-four and sixty-five.

Example Sixty-Five

Finding Average Text-Based Coding (Using a For Each Loop)
Standard algorithm used on AP exam

```
Line 1: PROCEDURE findAverage(list)
Line 2: {
Line 3: sum ← 0
Line 4: FOR EACH item IN list
Line 5: {
Line 6:   sum ← sum + item
Line 7: }
Line 8: RETURN(sum/LENGTH(list))
Line 9: }
```

Example Sixty-Six

Finding Maximum Number in a Text-Based Coding List
Standard algorithm used on AP exam

```
Line 1:   PROCEDURE findMaximum(list)
Line 2:   {
Line 3:   max ← list[1]
Line 4:   n ← LENGTH(list)
Line 5:   count ← 1
Line 6:   REPEAT n TIMES
Line 7:   {
Line 8:     IF(list[count] > max)
Line 9:       max = list[count]
Line 10:    count ← count + 1
Line 11:  }
Line 12:  RETURN(max)
Line 13:  }
```

Line 3 initializes the variable max to the first element in the list.
Line 4 initializes the variable *n* to the number of items in the list.
Line 5 initializes the count to 1.
Line 6 repeats the loop *n* number of times.
In lines 7 to 9, if the current value is greater than the maximum, then the current value is set to the maximum.
Line 10 increases the value of the count.
Line 12 returns the maximum value in the data structure.

Code can be written in different ways and still achieve the same results. For example, the code for finding a maximum number in a list could also be written using a For Each loop and achieve the same results with a different syntax. Compare the coding in examples sixty-six and sixty-seven.

Example Sixty-Seven

Finding Maximum Number in a Text-Based Coding List (Using a For Each Loop)
Standard algorithm used on AP exam

```
Line 1:   PROCEDURE findMaximum(list)
Line 2:   {
Line 3:   max ← list[1]
Line 4:   FOR EACH item IN list
Line 5:   {
Line 6:     IF(item > max)
Line 7:       max = item
Line 8:   }
Line 9:   RETURN(max)
Line 10:  }
```

Example Sixty-Eight

DANGER AP exam trick question!!! DANGER
The AP exam loves to trick students with an added ELSE statement.

```
Line 1:  PROCEDURE findMaximum(list)
Line 2:  {
Line 3:  max ← list[1]
Line 4:  FOR EACH item IN list
Line 5:  {
Line 6:     IF(item > max)
Line 7:     max = item
Line 8:     ELSE
Line 9:     max = 0
Line 8:  }
Line 9:  RETURN(max)
Line 10: }
```

Using the above code with the list [1, 1, 35, 6] will return 0, not the expected maximum number!

Maximum	Item	Return
1		
0	1	
1	1	
35	35	
0	6	
		0

Example Sixty-Nine

DANGER AP exam trick question!!! DANGER
The AP exam loves to trick students with the initial value of the max.

```
Line 1:  PROCEDURE findMaximum(list)
Line 2:  {
Line 3:  max ← 0
Line 4:  FOR EACH item IN list
Line 5:  {
Line 6:    IF(item > max)
Line 7:    max = item
Line 8:  }
Line 9:  RETURN(max)
Line 10: }
```

Using the above code with the list [−1, −1, −35, −6] will return 0, not the expected maximum number!

Maximum	Item	Return
0		
0	−1	
0	−1	
0	−35	
0	−6	
		0

Example Seventy

Finding Minimum Number in a Text-Based Coding List
Standard algorithm used on AP exam

```
Line 1:   PROCEDURE findMinimum(list)
Line 2:   {
Line 3:   min ← list[1]
Line 4:   n ← LENGTH(list)
Line 5:   count ← 1
Line 6:   REPEAT n TIMES
Line 7:   {
Line 8:     IF(list[count] < min)
Line 9:      min = list[count]
Line 10:    count ← count + 1
Line 11:  }
Line 12:  RETURN(min)
Line 13:  }
```

Line 3 initializes the variable "min" to the first element in the list.

Line 4 initializes the variable *n* to the number of items in the list.
Line 5 initializes the count to 1.
Line 6 repeats the loop *n* number of times.
Lines 7 to 9 means that if the current value is greater than the minimum, the current value is set to the minimum.
Line 10 increases the value of the count.
Line 12 returns the minimum value in the data structure.

Code can be written in different ways and still achieve the same results. For example, the code for finding a minimum number in a list could also be written using a For Each loop and achieve the same results using different syntax. Compare the coding in examples seventy and seventy-one.

Example Seventy-One

```
Line 1:  PROCEDURE findMinimum(list)
Line 2:  {
Line 3:  min ← list[1]
Line 4:  FOR EACH item IN list
Line 5:  {
Line 6:    IF(item < min)
Line 7:    min = item
Line 8:  }
Line 9:  RETURN(min)
Line 10: }
```

Example Seventy-Two

DANGER AP exam trick question!!! DANGER

The AP exam loves to trick students with an added ELSE statement.

```
Line 1:  PROCEDURE findMinimum(list)
Line 2:  {
Line 3:  min ← list[1]
Line 4:  FOR EACH item IN list
Line 5:  {
Line 6:    IF(item < min)
Line 7:      min = item
Line 8:    ELSE
Line 9:      min = 0
Line 8:  }
Line 9:  RETURN(min)
Line 10: }
```

Using the above code with the list [1, 1, 35, 6] will return 0, not the expected minimum number!

Minimum	Item	Return
1		
0	1	
0	1	
0	35	
0	6	
		0

Example Seventy-Three

DANGER AP exam trick question!!! DANGER

The AP exam loves to trick students with the initial value of the minimum.

```
Line 1:   PROCEDURE findMinimum(list)
Line 2:   {
Line 3:   min ← 0
Line 4:   FOR EACH item IN list
Line 5:   {
Line 6:     IF(item < min)
Line 7:       min = item
Line 8:   }
Line 9:   RETURN(min)
Line 10:  }
```

Using the above code with the list [1, 1, 35, 6] will return 0, not the expected minimum number!

Minimum	Item	Return
0		
0	1	
0	1	
0	35	
0	6	
		0

Example Seventy-Four

Searching for a Word in a Text-Based Coding List

Standard algorithm used on AP exam

```
Line 1:   PROCEDURE findWord(list, word)
Line 2:   {
Line 3:   index ← 1
Line 4:   FOR EACH item IN list
Line 5:   {
Line 6:     IF(item = word)
Line 7:     {
Line 8:     RETURN index
Line 9:     }
Line 10:  index ← index + 1
Line 11:  }
Line 12:  RETURN("Word not in list")
Line 13:  }
Line 14:  str ← [red, blue, purple]
```

```
Line 15: DISPLAY(findWord(str, blue) // would display 2
Line 16: DISPLAY(findWord(str, red) // would display 1
Line 17: DISPLAY(findWord(str, pink) // would display Word not in list
```

Line 3 initializes the variable index to equal 1 since that is where to begin the search for "word."
Line 4 is a For Each loop that will loop for every item in the list. Each iteration then looks at the next object in the list, calling it "item."
If the current item is the same as the word being searched for, lines 6 to 9 will return that given index.
Line 10 adds 1 to the index so the next iteration will check the element in the following index.
Line 12 means that if the procedure reaches this part of the program, the word is not in the list after checking every element of the list.

Example Seventy-Five

Searching for a Word in a Text-Based Coding Alternate Solution List
Standard algorithm used on AP exam

```
Line 1:  PROCEDURE findWord(list, word)
Line 2:  {
Line 3:  index ← 1
Line 4:  FOR EACH item IN list
Line 5:  {
Line 6:  IF(item = word)
Line 7:  {
Line 8:  RETURN index
Line 9:  }
Line 10: ELSE
Line 11: {
Line 12: index ← index + 1
Line 13: }
Line 14: }
Line 15: RETURN("Word not in list")
Line 16: }
Line 17: str ←[ red, blue, purple]
Line 18: DISPLAY(findWord(str, blue) // would display 2
Line 19: DISPLAY(findWord(str, red) // would display 1
Line 20: DISPLAY(findWord(str, pink) // would display Word not in list
```

Line 3 initializes the index equal to 1 because that is the starting point.
Lines 4 to 7 make a For Each loop to look at every item in the list.
If the item being looked for is the same as the word, line 8 will return the index currently being looked at.
Line 12 If the item being looked for is not the same as the word, the procedure will add 1 to the index.