Where are we?

Chapter 1: The Big Picture
Chapter 2: Binary Values and Number Systems
Chapter 3: Date Representation
Chapter 4. Gates and Circuits
Chapter 5. Computing Components
Chapter 6. Low-Level Programming Languages and Pseudocode
Chapter 7. Problem Solving and Algorithms
Chapter 8. Abstract Data Types and Subproblems
Chapter 9. Object-Oriented Design and High-Level Programming languages
Chapter 10. Operating Systems
Chapter 11. File Systems and Directories
Chapter 12. Information Systems
Chapter 13. Artificial Intelligence
Chapter 14. Simulation, Graphics, Gaming, and Other Applications
Chapter 15. Networks
Chapter 16. The World Wide Web
Chapter 17. Computer Security
Chapter 18. Limitations and Computing
Chapter Goals

- Describe the computer problem-solving process and relate it to Polya’s How to Solve It list
- Distinguish between a simple type and a composite type
- Describe two composite data-structuring mechanisms
- Recognize a recursive problem and write a recursive algorithm to solve it
- Distinguish between an unsorted array and a sorted array
- Distinguish between a selection sort and an insertion sort
Chapter Goals

- Describe the Quicksort algorithm
- Apply the selection sort, the bubble sort, insertion sort, and Quicksort to an array of items by hand
- Apply the binary search algorithm
- Demonstrate an understanding of the algorithms in this chapter by hand-simulating them with a sequence of items
Problem Solving

- Problem solving
  - The act of finding a solution to a perplexing, distressing, vexing, or unsettled question

- How do you define problem solving?
Problem Solving

- **How to Solve It**
  - A New Aspect of Mathematical Method by George Polya (1945)
  - "How to solve it list" written within the context of mathematical problems
- But list is quite general

We can use it to solve computer related problems!
Problem Solving

How do you solve problems?
1. Understand the problem
2. Devise a plan
3. Carry out the plan
4. Look back
Ask questions!
- What do I know about the problem?
- What is the information that I have to process in order to find the solution?
- What does the solution look like?
- What sort of special cases exist?
- How will I recognize that I have found the solution?
Strategies

- Look for familiar things! *Never reinvent the wheel!*
  - Similar problems come up again and again in different guises
  - A good programmer recognizes a task or subtask that has been solved before and plugs in the solution
  - Can you think of two similar problems?
Divide and Conquer!

- Break up a large problem into smaller units and solve each smaller problem
- Applies the concept of abstraction
- The divide-and-conquer approach can be applied over and over again until each subtask is manageable
Computer Problem-Solving

1. Analysis and Specification Phase
   Analyze
   Specification

2. Algorithm Development Phase
   Develop algorithm
   Test algorithm

3. Implementation Phase
   Code
   Test

4. Maintenance Phase
   Use
   Maintain
Phase Interactions

Should we add another arrow?

(What happens if the problem is revised?)
Top-Down Design

- Process continues for as many levels as it takes to make every step concrete
- Name of (sub)problem at one level becomes a module at next lower level
Algorithms

- **Algorithm**
  - A set of unambiguous instructions for solving a problem or subproblem in a finite amount of time using a finite amount of data

- **Abstract Step**
  - An algorithmic step containing unspecified details

- **Concrete Step**
  - An algorithm step in which all details are specified
Summary of Methodology

- Analyze the Problem
  - Understand the problem!!
  - Develop a plan of attack

- List the Main Tasks (becomes Main Module)
  - Restate problem as a list of tasks (modules)
  - Give each task a name

- Write the Remaining Modules
  - Restate each abstract module as a list of tasks
  - Give each task a name

- Re-sequence and Revise as Necessary
  - Process ends when all steps (modules) are concrete
Control Structures

- An instruction that determines the order in which other instructions in a program are executed
- Can you name the ones we defined in the functionality of pseudocode?
Selection Statements

Flow of control of if statement
Algorithm with Selection

Problem: Write the appropriate dress for a given temperature.

Write "Enter temperature"
Read temperature
Determine Dress

Which statements are concrete?
Which statements are abstract?
Algorithm with Selection

Determine Dress

IF (temperature > 90)
    Write “Texas weather: wear shorts”
ELSE IF (temperature > 70)
    Write “Ideal weather: short sleeves are fine”
ELSE IF (temperature > 50)
    Write “A little chilly: wear a light jacket”
ELSE IF (temperature > 32)
    Write “Philadelphia weather: wear a heavy coat”
ELSE
    Write “Stay inside”
Looping Statements

Flow of control of while statement
A count-controlled loop

Set sum to 0
Set count to 1
While (count <= limit)
    Read number
    Set sum to sum + number
    Increment count
Write "Sum is " + sum

Why is it called a count-controlled loop?
Looping Statements

An event-controlled loop

Set sum to 0
Set allPositive to true
WHILE (allPositive)
    Read number
    IF (number > 0)
        Set sum to sum + number
    ELSE
        Set allPositive to false
Write "Sum is " + sum

Why is it called an event-controlled loop?
Looping Statements

Calculate Square Root

Read in square
Calculate the square root
Write out square and the square root

Are there any abstract steps?
Looping Statements

Calculate Square Root

Set epsilon to 1
WHILE (epsilon > 0.001)
  Calculate new guess
  Set epsilon to abs(square - guess * guess)

Are there any abstract steps?
Looping Statements

Calculate New Guess

Set newGuess to
\((\text{guess} + (\text{square/guess})) / 2.0\)

Are there any abstract steps?
Looping Statements

Read in square
Set guess to square/4
Set epsilon to 1
WHILE  (epsilon > 0.001)
    Calculate new guess
    Set epsilon to abs(square - guess * guess)
Write out square and the guess
Composite Data Types

- **Records**
  - A named heterogeneous collection of items in which individual items are accessed by name
  - For example, we could bundle name, age and hourly wage items into a record named Employee

- **Arrays**
  - A named homogeneous collection of items in which an individual item is accessed by its position (index) within the collection
Composite Data Types

Employee

- name
- age
- hourly/Wage

Following algorithm, stores values into the fields of record:

```java
Employee employee  // Declare and Employee variable
Set employee.name to "Frank Jones"
Set employee.age to 32
Set employee.hourlyWage to 27.50
```
Arrays

- As data is being read into an array, a counter is updated so that we always know how many data items were stored.
- If the array is called list, we are working with:
  - `list[0] to list[length-1]` or
  - `list[0]...list[length-1]`
Arrays

numbers[0] = 1066
Discussion:
Unsorted versus Sorted Arrays

Find the largest or smallest item
Find an item
Add an item
An Unsorted Array

data[0]...data[length-1] is of interest
Example: Array

Fill array numbers with *limit* values

```
integer data[20]
Write “How many values?”
Read length
Set index to 0
WHILE (index < length)
    Read data[index]
    Set index to index + 1
```
Sequential Search of an Unsorted Array

- A sequential search examines each item in turn and compares it to the one we are searching.
- If it matches, we have found the item. If not, we look at the next item in the array.
- We stop either when we have found the item or when we have looked at all the items and not found a match.
- Thus, a loop with two ending conditions.
Sequential Search Algorithm

Set Position to 0
Set found to FALSE
WHILE (position < length AND NOT found )
    IF (numbers [position] equals searchitem)
        Set found to TRUE
    ELSE
        Set position to position + 1
Booleans

- **Boolean Operators**
  - A *Boolean variable* is a location in memory that can contain either true or false.
  - Boolean operator **AND** returns TRUE if both operands are true and FALSE otherwise.
  - Boolean operator **OR** returns TRUE if either operand is true and FALSE otherwise.
  - Boolean operator **NOT** returns TRUE if its operand is false and FALSE if its operand is true.
Sorted Arrays

- The values stored in an array have unique keys of a type for which the relational operators are defined.
- Sorting rearranges the elements into either ascending or descending order within the array.
- A sorted array is one in which the elements are in order.
Sequential Search in a Sorted Array

- If items in an array are sorted, we can stop looking when we pass the place where the item would be if it were present in the array.

Is this better?
A Sorted Array

A sorted array of integers
A Sorted Array

Read in array of values
Write “Enter value for which to search”
Read searchItem
Set found to TRUE if searchItem is there
IF (found)
    Write “Item is found”
ELSE
    Write “Item is not found”
A Sorted Array

Set found to TRUE if searchItem is there
Set index to 0
Set found to FALSE
WHILE (index < length AND NOT found)
  IF (data[index] equals searchItem)
    Set found to TRUE
  ELSE IF (data[index] > searchItem)
    Set index to length
  ELSE
    Set index to index + 1
Binary Search

- **Sequential search**
  - Search begins at the beginning of the list and continues until the item is found or the entire list has been searched

- **Binary search (list must be sorted)**
  - Search begins at the middle and finds the item or eliminates half of the unexamined items; process is repeated on the half where the item might be

Say that again…
Set first to 0
Set last to length-1
Set found to FALSE
WHILE (first <= last AND NOT found)
    Set middle to (first + last)/ 2
    IF (item equals data[middle])
        Set found to TRUE
    ELSE
        IF (item < data[middle])
            Set last to middle – 1
        ELSE
            Set first to middle + 1
    RETURN found
## Binary Search

### Searching for cat

<table>
<thead>
<tr>
<th>First</th>
<th>Last</th>
<th>Middle</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>5</td>
<td>cat &lt; dog</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>2</td>
<td>cat &lt; chicken</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>cat &lt; ant</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>cat = cat</td>
</tr>
</tbody>
</table>

### Searching for fish

<table>
<thead>
<tr>
<th>First</th>
<th>Last</th>
<th>Middle</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>5</td>
<td>fish &gt; dog</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>8</td>
<td>fish &lt; horse</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>6</td>
<td>fish = fish</td>
</tr>
</tbody>
</table>

### Searching for zebra

<table>
<thead>
<tr>
<th>First</th>
<th>Last</th>
<th>Middle</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>5</td>
<td>zebra &gt; dog</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>8</td>
<td>zebra &gt; horse</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>9</td>
<td>zebra &gt; rat</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>zebra &gt; snake</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>first &gt; last</td>
<td>Return: false</td>
</tr>
</tbody>
</table>

---

**Figure 7.10** Trace of the binary search
## Binary Search

**Table 7.1 Average Number of Comparisons**

<table>
<thead>
<tr>
<th>Length</th>
<th>Sequential Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5.5</td>
<td>2.9</td>
</tr>
<tr>
<td>100</td>
<td>50.5</td>
<td>5.8</td>
</tr>
<tr>
<td>1000</td>
<td>500.5</td>
<td>9.0</td>
</tr>
<tr>
<td>10000</td>
<td>5000.5</td>
<td>12.0</td>
</tr>
</tbody>
</table>

*Is a binary search always better?*
Recursion

**Recursion**
The ability of a subprogram to call itself

**Base case**
The case to which we have an answer

**General case**
The case that expresses the solution in terms of a call to itself with a smaller version of the problem
Recursion

For example, the factorial of a number is defined as the number times the product of all the numbers between itself and 0:

\[ N! = N \times (N - 1)! \]

**Base case**

Factorial(0) = 1 (0! is 1)

**General Case**

Factorial(N) = N \times Factorial(N-1)
Recursion

Write “Enter n”
Read n
Set result to Factorial(n)
Write result + “ is the factorial of “ + n

Factorial(n)
IF (n equals 0)
  RETURN 1
ELSE
  RETURN n * Factorial(n-1)
Recursion

**BinarySearch (first, last)**

*IF (first > last)*

  RETURN FALSE

ELSE

  Set middle to (first + last)/ 2

  *IF (item equals data[middle])*

  RETURN TRUE

ELSE

  *IF (item < data[middle])*

  **BinarySearch (first, middle – 1)**

  ELSE

  **BinarySearch (middle + 1, last)**
Sorting

- **Sorting**
  - Arranging items in a collection so that there is an ordering on one (or more) of the fields in the items

- **Sort Key**
  - The field (or fields) on which the ordering is based

- **Sorting algorithms**
  - Algorithms that order the items in the collection based on the sort key

Why is sorting important?
Selection Sort

- Given a list of names, put them in alphabetical order
  - Find the name that comes first in the alphabet, and write it on a second sheet of paper
  - Cross out the name off the original list
  - Continue this cycle until all the names on the original list have been crossed out and written onto the second list, at which point the second list contains the same items but in sorted order
Selection Sort

- A slight adjustment to this manual approach does away with the need to duplicate space
  - As you cross a name off the original list, a free space opens up
  - Instead of writing the value found on a second list, exchange it with the value currently in the position where the crossed-off item should go
Selection Sort

Figure 7.11  Example of a selection sort (sorted elements are shaded)
Selection Sort

Set firstUnsorted to 0

WHILE (not sorted yet)
  Find smallest unsorted item
  Swap firstUnsorted item with the smallest
  Set firstUnsorted to firstUnsorted + 1

Not sorted yet
firstUnsorted < length – 1
Selection Sort

**Find smallest unsorted item**
Set indexOfSmallest to firstUnsorted
Set index to firstUnsorted + 1
WHILE (index <= length – 1)
    IF (data[index] < data[indexOfSmallest])
        Set indexOfSmallest to index
        Set index to index + 1
    Set index to indexOfSmallest
Selection Sort

**Swap firstUnsorted with smallest**

Set tempItem to data[firstUnsorted]
Set data[firstUnsorted] to data[indexOfSmallest]
Set data[indexOfSmallest] to tempItem
Bubble Sort

- Bubble Sort uses the same strategy
  - Find the next item
  - Put it into its proper place
- But uses a different scheme for finding the next item
- Starting with the last list element, compare successive pairs of elements, swapping whenever the bottom element of the pair is smaller than the one above it
Bubble Sort

a. First iteration (Sorted elements are shaded.)

b. Remaining iterations (Sorted elements are shaded.)
**Bubble Sort**

Set `firstUnsorted` to 0
Set `index` to `firstUnsorted` + 1
Set `swap` to TRUE

WHILE (index < length AND swap)
    Set `swap` to FALSE
    “Bubble up” the smallest item in unsorted part
Set `firstUnsorted` to `firstUnsorted` + 1

Why is “swap” needed?
Bubble Sort

Bubble up
Set index to length – 1
WHILE (index > firstUnsorted + 1)
  IF (data[index] < data[index – 1])
    Swap data[index] and data[index – 1]
    Set swap to TRUE
  Set index to index - 1
Insertion Sort

- If you have only one item in the array, it is already sorted.
- If you have two items, you can compare and swap them if necessary, sorting the first two with respect to themselves.
- Take the third item and put it into its place relative to the first two.
- Now the first three items are sorted with respect to one another.
Insertion Sort

- The item being added to the sorted portion can be bubbled up as in the bubble sort.
Insertion Sort

*InsertionSort*

Set current to 1

WHILE (current < length)

  Set index to current
  Set placeFound to FALSE
  WHILE (index > 0 AND NOT placeFound)
    IF (data[index] < data[index – 1])
      Swap data[index] and data[index – 1]
      Set index to index – 1
    ELSE
      Set placeFound to TRUE
  SET current to current + 1
Insertion Sort

(a) Sue
   Cora
   Beth
   Ann
   June

(b) Ann
   Cora
   Beth
   Sue
   June

(c) Ann
   Beth
   Cora
   Sue
   June

(d) Ann
   Beth
   Cora
   Sue
   June

(e) Ann
   Beth
   Cora
   June
   Sue
Quicksort

Ordering a list using the Quicksort algorithm

It is easier to sort a smaller number of items: Sort A…F, G…L, M…R, and S…Z and A…Z is sorted
Quicksort algorithm

- With each attempt to sort the stack of data elements, the stack is divided at a splitting value, splitVal, and the same approach is used to sort each of the smaller stacks (a smaller case).
- Process continues until the small stacks do not need to be divided further (the base case).
- The variables first and last in Quicksort algorithm reflect the part of the array data that is currently being processed.
Quicksort

Quicksort(first, last)

IF (first < last) // There is more than one item

Select splitVal

Split (splitVal) // Array between first and

// splitPoint−1 <= splitVal

// data[splitPoint] = splitVal

// Array between splitPoint + 1

// and last > splitVal

Quicksort (first, splitPoint - 1)

Quicksort (splitPoint + 1, last)

Recursion or not?
Quicksort

splitVal = 9

<table>
<thead>
<tr>
<th>9</th>
<th>20</th>
<th>6</th>
<th>10</th>
<th>14</th>
<th>8</th>
<th>60</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>[first]</td>
<td>[last]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

smaller values: 9, 8, 6
larger values: 10, 14, 20, 60, 11

<table>
<thead>
<tr>
<th>9</th>
<th>8</th>
<th>6</th>
<th>10</th>
<th>14</th>
<th>20</th>
<th>60</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>[first]</td>
<td>[last]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

smaller values: 6, 8, 9
larger values: 10, 14, 20, 60, 11

<table>
<thead>
<tr>
<th>6</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>14</th>
<th>20</th>
<th>60</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>[first]</td>
<td>[split-Point]</td>
<td>[last]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Quicksort**

Given an array `data` and a value `splitVal`, the quicksort algorithm works as follows:

1. **Split**(`splitVal`)
2. Set `left` to `first + 1`
3. Set `right` to `last`
4. **WHILE** (`left` <= `right`)
   - Increment `left` until `data[left]` > `splitVal` OR `left` > `right`
   - Decrement `right` until `data[right]` < `splitVal` OR `left` > `right`
   - **IF** (`left` < `right`)
     - Swap `data[left]` and `data[right]`
   - Set `splitPoint` to `right`
5. **Swap** `data[first]` and `data[splitPoint]`
6. **Return** `splitPoint`

- **Example:**
  - `splitVal = 9`
  - Array: `9 20 6 10 14 8 60 11`
  - Initial state: `[first] 9 20 6 10 14 8 60 11 [last]`
  - After splitting (and recursively): `5 10 6 14 8 9 60 11`
Quicksort

a. Initialization

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>20</td>
<td>6</td>
</tr>
</tbody>
</table>

[first] [left] [right]

b. Increment left until list[left] > splitVal or left > right

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>20</td>
<td>6</td>
</tr>
</tbody>
</table>

[first] [left] [right]

c. Decrement right until list[right] > splitVal or left > right

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>20</td>
<td>6</td>
</tr>
</tbody>
</table>

[first] [left] [right]

d. Swap list[left] and list[right]; move left and right toward each other

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

[first] [left] [right]

e. Increment left until list[left] > splitVal or left > right

Decrement right until list[right] <= splitVal or left > right.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

[first] [right] [left]

f. left > right so no swap occurs within the loop

Swap list[first] and list[right]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

[first] [right] (splitPoint)
Important Threads

- **Information Hiding**
  - The practice of hiding the details of a module with the goal of controlling access to it

- **Abstraction**
  - A model of a complex system that includes only the details essential to the viewer

- **Information Hiding and Abstraction** are two sides of the same coin
Important Threads

- **Data abstraction**
  - Separation of the logical view of data from their implementation

- **Procedural abstraction**
  - Separation of the logical view of actions from their implementation

- **Control abstraction**
  - Separation of the logical view of a control structure from its implementation
Important Threads

- **Identifiers**
  - Names given to data and actions, by which we access the data
    - We access the data
      \[
      \text{Read firstName, Set count to count + 1}
      \]
    - We execute the actions
      \[
      \text{Split(splitVal)}
      \]
  - Giving names to data and actions is a form of abstraction
Abstraction is the most powerful tool people have for managing complexity!
Picture sources of today’s slides

- Jones & Barlett Learning’s slides
- Prof. Jaehoon Jeong’s slides