Chapter 7

Ch. 7 Problem Solving and Algorithms
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 the list is quite general

We can use it to solve computer related problems!
1. Understand the problem
   – What are the hypotheses? Data? Unknowns?
2. Devise a plan
   – Can we find a related problem? A sub-problem?
   – Can we strengthen or relax the hypotheses to obtain an easier problem?
3. Carry out the plan
4. Look back
   – Check result
   – Find shortcuts and alternate solutions
   – Generalize to related problems
Strategies

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

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 we solved in Python?
Strategies

Divide and Conquer!

Break up a large problem into smaller subproblems and solve each separately

– Applies the concept of abstraction

– The divide-and-conquer approach can be applied over and over again until each subtask is manageable
Polya’s 4 steps can be applied to Computer Problem-Solving

Analysis and Specification Phase
   Analyze
   Specification
Algorithm Development Phase
   Develop algorithm
   Test algorithm
Implementation Phase
   Code algorithm
   Test algorithm
Maintenance Phase
   Use
   Maintain
QUIZ: Match the steps in Polyá’s method to the ones in the computer method for problem solving

Devise a plan: Analysis and Specification
Look back: Implementation
Understand: Algorithm Development
Carry out the plan: Maintenance
Phase Interactions

- Analysis and Specification (test)
- Algorithm Development (General Solution) (test)
- Implementation (Concrete Solution) (test)
- Maintenance
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
7.2 Algorithms with simple variables

Variable = a means of storing intermediate results from one task to the next.

At the hardware level, a simple variable is just one or several adjacent Bytes in the computer memory.

Q: How many Bytes does a simple variable have in PEP/8?
Algorithms with Selection Statements (a.k.a. decision or if)

Flow of control of if statement

true

Boolean expression

false

Zero or more statements in sequence

Rest of program or module

Zero or more statements in sequence
Algorithm with Selection

Problem: Write the appropriate dress for a given temperature.

Algorithm *Determine Dress* v.1

Write "Enter temperature"
Read temperature
Determine Dress

*Which statements are concrete?*
*Which statements are abstract?*

Computer language is Python from now on!
Algorithm Determine Dress v.2

Write “Enter temperature”
Read temperature
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”

Is this concrete enough for implementation in Python?
Algorithms with Loops (a.k.a. repetition)

Flow of control of while statement
QUIZ: In Ch.6 we distinguished between loops that can execute 0 times and loops that execute at least 1 time. Which type is this?
Event-controlled Loops

They are the most general type of loops

```
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
```
Counter-controlled Loops

They are a particular case of event-controlled loops: the event is that a counter variable has reached a predetermined limit

Set sum to 0
Set limit to 42
Set count to 1
While (count <= limit)
    Read number
    Set sum to sum + number
    Increment count
Write "Sum is " + sum
Important application of looping: Successive approximation algorithms

Algorithm *Calculate Square Root* v.1

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

Which steps are abstract and which concrete?
Algorithm \textbf{Calculate Square Root v.2}

Set \( \epsilon \) to 1

\textbf{WHILE} \ (\epsilon > 0.001) \n
Calculate new guess

Set \( \epsilon \) to \( \text{abs} (\text{square} - \text{guess} \times \text{guess}) \)

\textbf{A more appropriate name for guess would be approximation}

\textit{In Python use}

\texttt{math.fabs(...)}

Which steps are abstract and which concrete?
Algorithm Calculate Square Root - Refinements in v.2

What’s the mathematical formula for the new approximation?

Set newGuess to

\[ (guess + (\text{square/guess})) / 2.0 \]

How do we get the loop started?

Set guess to square/4
Set epsilon to 1
Algorithm *Calculate Square Root* v.3

Read in square
Set guess to square/4
Set epsilon to 1
WHILE (epsilon > 0.001)
    Set guess to (guess + (square/guess)) / 2.0
    Set epsilon to abs(square - guess * guess)
Write out square and guess

Which steps are abstract and which concrete?
Extra-credit:

Implement algorithm **Calculate square root** in Python, using the **while** command.

Check the result using **math.sqrt()**.

Due next time (Mon) at the beginning of class.

The full pseudocode is on pp.208-9 of our text.
Read Sections 7.1 and 7.2

To do in notebook for next time:
End-of-chapter questions
1-10 and 16 – 24
• Solutions for Exam 2
7.3 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.

Are these the *lists* from Python? Why not?

Python strings are arrays of characters!
Composite Data Types

Lists (will be covered in next chapter)
A named heterogeneous collection of items in which individual items are accessed by position (index).

We have them in Python, e.g.

```python
>>> myList = [“dog”, 42, 51.375, [1,2]]
>>> myList[1]
42
```
Composite Data Types - Records

Employee

- name
- age
- hourly/Wage

Algorithm to store values into the fields of record:

```
Employee employee  // Declare an Employee variable
Set employee.name to “Frank Jones”
Set employee.age to 32
Set employee.hourlyWage to 27.50
```
Composite Data Types - Arrays

numbers

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>62</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>71</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>78</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>73</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>75</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>67</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `numbers[0]` contains 62
- `numbers[4]` contains 73
Some items in an array may be unused at a given time

<table>
<thead>
<tr>
<th>numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

(first)  

(last)
Useful Algorithms on Arrays

- Initializing all items
- Printing all items
- Searching for an item
- Sorting the array
Initializing arrays

Fill an array numbers with \textit{length} values that are being input from the keyboard

\begin{verbatim}
integer data[20]
Write “How many values?”
Read \textit{length}
Set index to 0
WHILE (index < \textit{length})
    Read data[index]
    Set index to index + 1
\end{verbatim}
QUIZ

Modify this pseudocode to print the values after initializing them.

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

length 6

data

60 [0]
75 [1]
95 [2]
80 [3]
65 [4]
90 [5]

[MAX_LENGTH-1]
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.

Reality check: In a real-life problem it’s very common to encounter repeated keys!
A Sorted Array

<table>
<thead>
<tr>
<th>length</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>60 [0]</td>
</tr>
<tr>
<td></td>
<td>65 [1]</td>
</tr>
<tr>
<td></td>
<td>75 [2]</td>
</tr>
<tr>
<td></td>
<td>80 [3]</td>
</tr>
<tr>
<td></td>
<td>90 [4]</td>
</tr>
<tr>
<td></td>
<td>95 [5]</td>
</tr>
</tbody>
</table>

[MAX_LENGTH-1]
8.4 Search algorithms
Sequential Search in 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, we have a loop with two ending conditions.
The array’s name is **numbers**
The value we’re searching for is stored in **searchItem**

```plaintext
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
```
QUIZ: When the loop exits, what do we need to do?

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
Sequential Search in **Sorted** Array

Idea:
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.
Sequential Search in 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”

Which steps are abstract and which concrete?
Sequential Search in 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”

This was explained before – see array initialization
Sequential Search in 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
QUIZ:
End-of-chapter question 66
Binary Search in Sorted Array

Search begins at the middle and finds the item or eliminates half of the unexamined items; the process is then repeated on the half where the item might be

<table>
<thead>
<tr>
<th>24</th>
<th>30</th>
<th>31</th>
<th>42</th>
<th>44</th>
<th>90</th>
<th>92</th>
<th>94</th>
<th>99</th>
</tr>
</thead>
</table>
Binary Search Algorithm

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 example

- **Searching for cat**
  - **First** | **Last** | **Middle** | **Comparison**
  - 0 | 10 | 5 | cat < dog
  - 0 | 4 | 2 | cat < chicken
  - 0 | 1 | 0 | cat < ant
  - 1 | 1 | 1 | cat = cat **Return: true**

- **Searching for fish**
  - **First** | **Last** | **Middle** | **Comparison**
  - 0 | 10 | 5 | fish > dog
  - 6 | 10 | 8 | fish < horse
  - 6 | 7 | 6 | fish = fish **Return: true**

- **Searching for zebra**
  - **First** | **Last** | **Middle** | **Comparison**
  - 0 | 10 | 5 | zebra > dog
  - 6 | 10 | 8 | zebra > horse
  - 9 | 10 | 9 | zebra > rat
  - 10 | 10 | 10 | zebra > snake
  - 11 | 10 | | first > last **Return: false**

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

Figure 7.10 Trace of the binary search

<table>
<thead>
<tr>
<th>Index</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ant</td>
</tr>
<tr>
<td>1</td>
<td>cat</td>
</tr>
<tr>
<td>2</td>
<td>chicken</td>
</tr>
<tr>
<td>3</td>
<td>cow</td>
</tr>
<tr>
<td>4</td>
<td>deer</td>
</tr>
<tr>
<td>5</td>
<td>dog</td>
</tr>
<tr>
<td>6</td>
<td>fish</td>
</tr>
<tr>
<td>7</td>
<td>goat</td>
</tr>
<tr>
<td>8</td>
<td>horse</td>
</tr>
<tr>
<td>9</td>
<td>camel</td>
</tr>
<tr>
<td>10</td>
<td>snake</td>
</tr>
</tbody>
</table>

Search for **deer**

<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>

Return: true
Binary Search Conclusion

Binary search algorithms have average and worst-case cost of $O(\log_2 N)$, but the array must be sorted.
Search Conclusion

Sequential search:  \( O(N) \), sorted or unsorted

Binary search:  \( O(\log_2 N) \), only sorted.
8.5 Sorting algorithms
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)
QUIZ Selection Sort

*Show the swapped elements with arrows.*
*Show the sorted elements with shading.*

<table>
<thead>
<tr>
<th>24</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
current < 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
To do in notebook for next time:
End-of-chapter questions 30, 31, 32, 67
Quiz

1. What are the 4 fundamental types of algorithms used to manipulate arrays?
2. What control structure is normally used to access the elements of an array?
3. Which is faster, sequential search or binary search?
   – How much faster? (use “Big-Oh” notation)
4. What is the downside of binary search?
SKIP Bubble Sort and Insertion Sort
7.6 Recursive Algorithms

Can we do sorting with less than $O(N^2)$ comparisons?

Yes, but it involves a new concept (recursion) ... ... and a new control structure! (subprogram)
Subprogram Statements

We can give a section of code a name and use that name as a statement in another part of the program.

When the name is encountered, the processing in the other part of the program halts while the named code is executed.

When execution is finished, the first part of the program resumes execution.

That section of code is called a subprogram.
Subprogram Statements

(a) Subprogram A does its task and calling unit continues with next statement
We already used subprograms in Python!

... but we called them

- Functions
  - int() float() ord() chr() str() open() etc.

- Methods
  - list.append() string.upper() file.close() etc.

Do you remember what each of them does?
Subprogram Statements

What if the subprogram needs data from the calling unit? This data is called **input**.

**Parameters**
Identifiers listed in parentheses beside the subprogram declaration; sometimes called **formal parameters**

**Arguments**
Identifiers listed in parentheses on the subprogram call; sometimes called **actual parameters**
Parameters and arguments in Python

def printer(a, b):
    print a, 'and', b

x, y = 1, 2
printer(x, y)
What if the subprogram needs to give data back to the calling unit? This data is called **output**.

**Void subprograms**
They do not return a value, just perform certain actions

```python
>>> print("Enter a positive integer")
>>> printer(3, 4)
```

**Value-returning subprograms**

```python
>>> a = input("Enter a positive integer")
The keyword RETURN is used in many programming languages
def sum(a, b):
    return a + b

x, y = 1, 2
print sum(x, y)
def sum(a, b):
    return a + b

Write a Python function that multiplies three numbers
Value-returning and void Subprograms

(a) Subprogram A does its task and calling unit continues with next statement

(b) Subprogram B does its task and returns a value that is added to 5 and stored in x

EoL
Subprogram Statements

Subprograms are very important tools for abstraction.

Other popular names for subprograms:

- sub
- subroutine
- function
- procedure
- module
- method
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

The **factorial** of a positive integer is defined as the integer times the product of all the integers between itself and 0:

\[ N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N \]

But an alternate recursive definition is possible:

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

**Base case**

\[ \text{Fact}(0) = 1 \quad (0! \text{ is } 1) \]

**General Case**

\[ \text{Fact}(N) = N \cdot \text{Fact}(N-1) \quad (\text{for } N \geq 1) \]
QUIZ

N! = N * (N − 1)!

Base case
Facto(0) = 1  (0! is 1)

General Case
Fact(N) = N * Fact(N-1)  (for N ≥ 1)

Calculate:
0! =
1! =
2! =
5! =
Recursive Factorial algorithm

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

\texttt{Factorial}(n)
IF \,(n\text{ equals }0)\,
\quad \text{RETURN } 1
ELSE
\quad \text{RETURN } n \times \texttt{Factorial}(n-1)
Recursive Binary Search

**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)
Quicksort algorithm

Imagine having to sort a stack of exams alphabetically by the students’ names.

To avoid the “wild swings” we’ve seen in Selection Sort, we’re going to split the stack into two smaller stacks, such that no exam will have to cross from one stack to another in the future.

How to split? In the case of names, we know that, on average, half of the names will start with A...L and half with M...Z.
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

In the general case, when we don’t know much about the nature of data elements, the array is divided at a splitting value, $splitVal$, which we choose at random from the array itself.

The process continues recursively until the small stacks do not need to be divided further (the base case).
The variables **first** and **last** indicate the part of the array (sub-stack) that is currently being processed.

```plaintext
Quicksort(first, last)

IF (first < last)  There is more than one item
  Select splitVal  Simplest to choose first
  Split (splitVal)  We split into the two sub-stacks
  Quicksort (first, splitPoint - 1)
  Quicksort (splitPoint + 1, last)
```
Quicksort

Initial array

After splitting

Swap split value to bring it at the split point
Quicksort – how to split the array

\textbf{Split(splitVal)}

Set left to first + 1
Set right to last
\textbf{WHILE} (left \leq right)
  \textbf{Increment} left until data[left] > splitVal \textbf{OR} left > right
  \textbf{Decrement} right until data[right] < splitVal \textbf{OR} left > right
\textbf{IF}(left < right)
  Swap data[left] and data[right]
Set splitPoint to right
\textbf{Swap} data[first] and data[splitPoint]
Return splitPoint
Detailed operation of the `split()` function

a. Initialization

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>20</td>
<td>6</td>
<td>10</td>
<td>14</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>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>20</td>
<td>6</td>
<td>10</td>
<td>14</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>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>20</td>
<td>6</td>
<td>10</td>
<td>14</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>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
</tbody>
</table>
[first] [left] [right]
```
First iteration:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>30</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[first]</td>
<td>[left]</td>
<td>[right]</td>
<td>[first]</td>
<td>[left]</td>
<td>[right]</td>
<td>[first]</td>
<td>[left]</td>
<td>[right]</td>
</tr>
</tbody>
</table>

Split value

splitPoint
7.7 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
Three types of abstraction

**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

*Unsigned integers can be implemented on 8, 16, 32, or 64 bits!*

*Subprograms do this!*

*A for loop is the same in pseudocode, but can have different syntax details in different languages!*
for loops in Python, C and FORTRAN

```python
for i in range(m, n, p):
    .......
```

```c
for (i=m; i<n; i+=p){
    .......
}
```

```fortran
do 10 i = m, n-1, p
    .......
10    continue
```
Important Threads

Identifiers

Names given to data and actions, by which
 mass
  we access the data and

  Read firstName, Set count to count + 1

  execute the actions

  Split(splitVal)

Giving names to data (variables) and actions (subprograms) is a form of abstraction
Chapter review questions

• 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
• Distinguish between a void subprogram and a value-returning subprogram
• Recognize a recursive problem and write a recursive algorithm to solve it
• Distinguish between an unsorted array and a sorted array
• Describe the Quicksort algorithm
• Apply the linear search, binary search, selection sort and Quicksort to an array of items by hand
Read and take notes:
Ethical Issues

Open-Source Software Development

What are the advantages and disadvantages of open-source software?

What does the success of Linux suggest about the future of open-source software?

Should open-source software be licensed and subject to standard copyright laws?
My wife Jill and I are holding the medal I received when I was knighted. What university did I retire from and where am I working now?
Who am I?

I am a mathematician. Why is my picture in a book about computer science?
Do you know?

What writing system did the Rosetta stone serve as a key to translating?

What are some of the adverse consequences of piggybacking for free off someone else’s paid Wi-Fi connection?

What, if any, legal privacy protections does a blogger have to resist an employer seeking to fire an anonymous blogging employee?
Homework for Ch. 7
Due Friday, April 20

End of chapter
11 through 15
33 through 36
63
65