Chapter 7

Problem Solving and Algorithms
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

How would you define problem solving?

Problem solving = The process of finding a solution to a perplexing, distressing, vexing, or unsettled question

Why is problem solving important in CS?
Why is problem solving different in CS?
Problem Solving

*How to Solve It: A New Aspect of Mathematical Method* by George Polya

"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!
Problem Solving

*How do we solve problems?*

Understand the problem

Devise a plan

Carry out the plan

Look back:
  – Check result
  – Find shortcuts
  – Generalize
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

Ask questions! 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?*
Strategies

Divide and Conquer!

Break up a large problem into smaller sub-problems 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
Computer Problem-Solving

Analysis and Specification Phase
- Analyze
- Specify

Algorithm Development Phase
- Develop algorithm
- Test algorithm

Implementation Phase
- Code algorithm
- Test algorithm

Maintenance Phase
- Use
- Maintain

Can you name a recurring theme?
Phase Interactions

Should we add another arrow?
(What happens if the problem is revised?)
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
Developing an Algorithm

Two methodologies used to develop computer solutions to a problem

– Top-down design focuses on the tasks to be done
– Object-oriented design focuses on the data involved in the solution (We will discuss this design in Ch. 9)
Summary of Methodology

Analyse 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
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. The end-result is a sequence of tasks to be performed.
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.

“Simple” is in the eye of the beholder – it means that the algorithm we’re using is treating the variable as a whole.
Control Structures

Control structure
An instruction that determines the order in which other instructions in a program are executed

Can you name the ones we defined in Python and 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.

Algorithm \textit{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

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”
To do in notebook for next time:
End-of-chapter questions 1 through 10
Looping Statements

Flow of control of while statement
Looping Statements

A counter-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

In Python:

```python
for i in range(1, limit+1):
    number = input(…)
    …
```
To do in class:

Fire up the Python command window and write a program that adds up all numbers from 1 to 100
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

In Python:

```python
while allPositive :
    number = input(…)
...
```
To do in class:

Implement the previous algorithm in Python, using the **while** command
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 **Calculate Square Root** v.2

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

In Python do this:
```
import math
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?

\[
x_{n+1} = \frac{1}{2} \left( x_n + \frac{S}{x_n} \right)
\]

Set *newGuess* to

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

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 the previous algorithm in Python, using the **while** command.

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

To do in notebook for next time:

End-of-chapter questions 16 – 24
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*.

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

Arrays

A named *homogeneous* collection of items in which an individual item is accessed by its position (index) within the collection.

[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:

```java
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**

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

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

<table>
<thead>
<tr>
<th></th>
<th>numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>??</td>
</tr>
<tr>
<td>1</td>
<td>71</td>
</tr>
<tr>
<td>2</td>
<td>78</td>
</tr>
<tr>
<td>3</td>
<td>66</td>
</tr>
<tr>
<td>4</td>
<td>??</td>
</tr>
<tr>
<td>5</td>
<td>??</td>
</tr>
<tr>
<td>6</td>
<td>??</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}
An Unsorted Array

data

length

6

0
60
75
95
80
65
90

[0]
[1]
[2]
[3]
[4]
[5]

data[0]...data[length-1] is of interest

[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

length
6

list
60
65
75
80
90
95

[0]
[1]
[2]
[3]
[4]
[5]

[MAX_LENGTH-1]
8.4 Search algorithms
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.
Sequential Search Algorithm

The array’s name is **numbers**
The value we’re searching for is stored in **searchItem**

*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*
How do we compare/evaluate algorithms? (not in text)

The **cost** of an algorithm can be defined in many ways. Here we mention only two:

- The time needed to run it
- The memory needed to run it
How do we compare/evaluate algorithms? (not in text)

Zooming in on the time-cost, in many applications we can isolate a “basic” operation (e.g. addition, multiplication, comparison, accessing an array element) that is responsible for most of the running-time of the algorithm.

Then the cost can be defined as the number of those operations performed by the algorithm.
How do we compare/evaluate algorithms? (not in text)

Example: Algorithm to add two arrays of integers. Both arrays have N integers.

Take the basic operation to be the addition of two integers.

Cost = N
How do we compare/evaluate algorithms? (not in text)

Another twist: Many algorithms do not take the same number of steps every time they execute!

Solution: Worst-case, average-case, best-case
Application: Sequential Search in Unsorted Array

What is the cost?

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
To do in notebook for next time:
End-of-chapter question 66
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 \textbf{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

What is the cost? Count only the comparisons!
Sequential search algorithms have average and worst-case cost of $O(N)$, whether the array is sorted or not.

In Algorithm Analysis, this is called “Big-Oh” notation. Constant terms and factors are neglected.
Binary Search in Sorted Array

**Sequential search**
Still possible, but it would be silly ...

**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
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

Integer Division!
Binary Search

Figure 7.10 Trace of the binary search
## Binary Search

<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>

*Table 7.1 Average Number of Comparisons*

Is a binary search always better?
Binary search algorithms have average and worst-case cost of $O(\log_2 N)$, but the array must be sorted.

Since changing the base of the logarithm means only a constant factor, we use the shorter notation $O(\log N)$.
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)
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?
Quiz

5. Perform **binary search** on the array given in the handout.
Selection Sort (not in text)

What is the cost?

The total number of comparisons is
\[ N + (N-1) + (N-2) + \ldots + 2 \]

To make a neat sum, let’s add a final 1 to it:

\[ N + (N-1) + (N-2) + \ldots + 2 + 1 = \frac{N \cdot (N + 1)}{2} = O(N^2) \]
The important thing to remember is that all 3 algorithms in Section 7.5 have cost of $O(N^2)$ comparisons.
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

Figure 7.14 Subprogram flow of control
We already used subprograms in Python!

... but we called them

• Functions
  – int()  float()  ord()  math.sqrt()  math.sin()  etc.

• Methods
  – list.append()  string.upper()  file.close()  etc.
We even defined our own subprogram in Python!

... the cool *ternary tree* at the end of Section 1.9 of the lab manual (p.43):

def ternaryTree(size, factor, t):
    if size >= 1:
        for i in range(3):
            t.forward(size)
    ...
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 ternaryTree(size, factor, t):
    if size >= 1:
        ...

ternaryTree(75, 0.5, Turtle())
Subprogram Statements

What if the subprogram needs to give data back to the calling unit? This data is called **output**.

**Value-returning subprograms**
The keyword RETURN is used in many programming languages

- \( a = \) input("Enter a positive integer")

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

- print("Enter a positive integer")
Subprogram Statements

Subprograms are very important tools for **abstraction**.

Other popular names for subprograms:

- `sub`
- `subroutine`
- `function`
- `procedure`
- `module`
- `method`
Value-returning and void Subprograms

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

Subprogram A()

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

x = 5 + SubprogramB()

Return result
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 \text{Factorial}(N-1)
Recursive Factorial

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

\texttt{Factorial}(n)
\texttt{IF} (n \texttt{equals} 0)
\hspace{1em} \texttt{RETURN} 1
\texttt{ELSE}
\hspace{1em} \texttt{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

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

Ordering a list using the Quicksort algorithm
Quicksort algorithm

With each attempt to sort the stack of data elements, the stack is divided at a splitting value, \textit{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 \textit{first} and \textit{last} in Quicksort algorithm reflect the part of the array \textit{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)
QuickSort

Initial array

After splitting

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

**Split(splitVal)**

Set left to first + 1
Set right to last
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
Swap data[first] and data[splitPoint]
Return splitPoint

Which steps are abstract and which concrete?
### Detailed operation of the split () function

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<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>
<td>8</td>
<td>60</td>
<td>11</td>
</tr>
<tr>
<td>[first]</td>
<td>[left]</td>
<td>[right]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**a. Initialization**

**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>
<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>
<td>8</td>
<td>60</td>
<td>11</td>
</tr>
<tr>
<td>[first]</td>
<td>[left]</td>
<td>[right]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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>
<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>
<td>8</td>
<td>60</td>
<td>11</td>
</tr>
<tr>
<td>[first]</td>
<td>[left]</td>
<td>[right]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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>
<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>
<td>20</td>
<td>60</td>
<td>11</td>
</tr>
<tr>
<td>[first]</td>
<td>[left]</td>
<td>[right]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
Important Threads

Identifiers

Names given to data and actions, by which

– we access the data and
  
  \textit{Read firstName, Set count to count + 1}

– execute the actions
  
  \textit{Split(splitVal)}

Giving names to data and actions is a form of abstraction
Abstraction is the most powerful tool people have for managing complexity!
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, Dec. 3
(last homework for this class)

End of chapter

11 through 15
33 through 36
63
65