COMP1002
Computational Thinking and Problem Solving

Lecture 7
Problem Solving I

Lecture 7

› Introduction to Problem Solving

› How to be a good problem solver?

› Problem Solving Steps
  – Abstraction
  – Algorithm design
  – Implementation

› Abstraction
  – A closer look
  – Examples

Introduction

- How to solve it? (George Pólya)
  - Understand the problem
    - What are you asked to find?
    - Think of a diagram that might help you understand the problem
    - Is there enough information to enable you to find a solution?
  - Devise a plan
    - Guess and check
    - Make an orderly list and eliminate possibilities
    - Use direct reasoning, e.g., solve equation
    - Look for a pattern or familiar problem / use a model
    - Solve a simpler problem first
    - Work backward
  - Carry out the plan
    - Follow what you have planned
    - Look back

Problem Solving

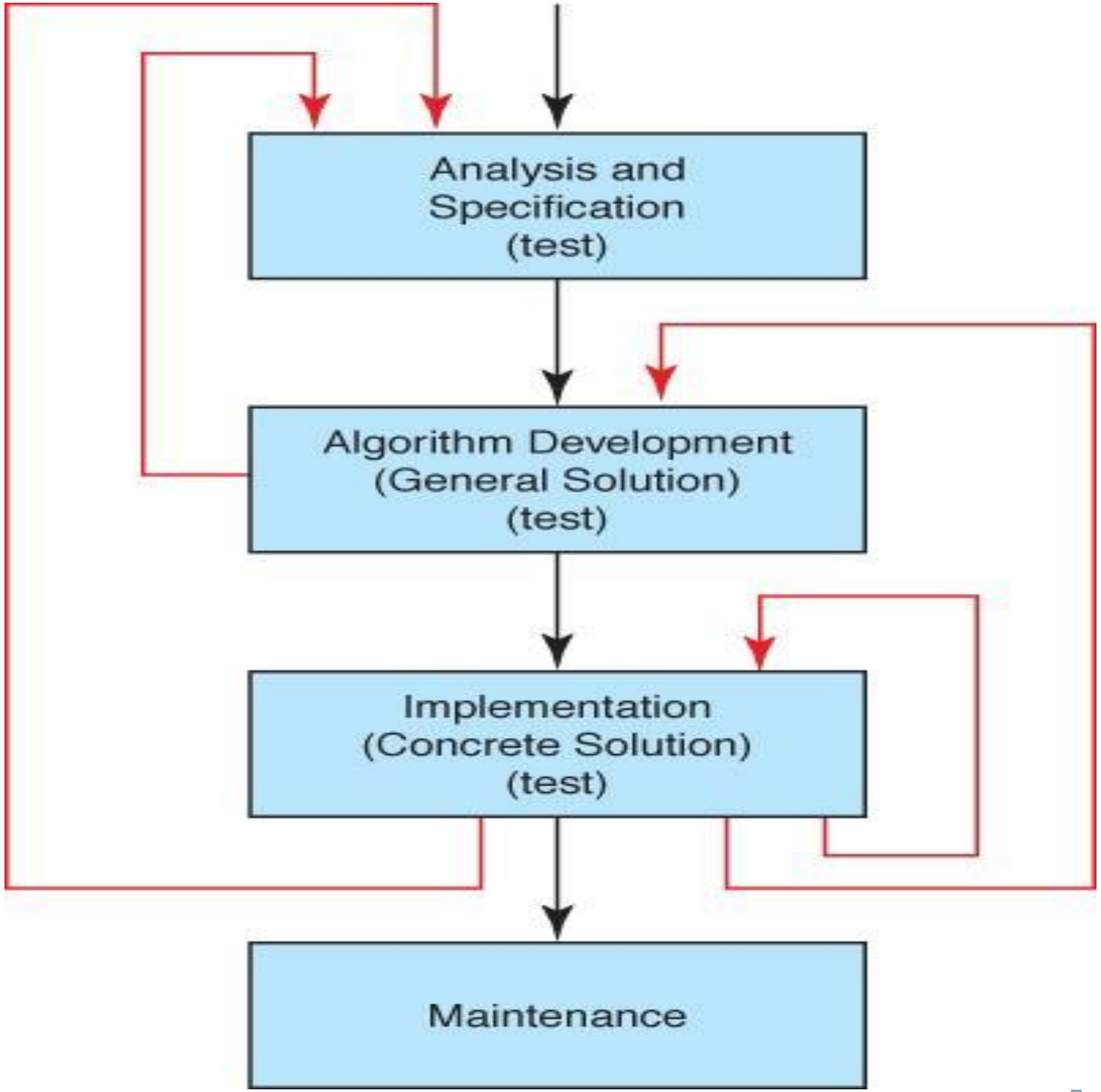
- Recall that problem solving in computing often involves three steps:
  - Abstraction
    - Problem formulation
  - Automation
    - Solution expression
  - Analysis
    - Solution execution and evaluation

Problem Solving

- To be more specific, problem solving in computing can also be considered as (slightly different terms):
  - Abstraction
    - Also called data modeling
  - Algorithm design
    - Often based on pseudo-code
  - Implementation
    - Really develop the program, test it and run it

Problem Solving

- Another view
- Feedback loop at each step



Problem Solving

- Computing is the automation of our abstractions
  - Once automated, computer can help solving the problem via many repeated steps
  - Computer never gets tired, even if you ask it to add 1 every time by starting from 0 until it reaches 1 billion!
- Computational thinking involves
  - Choosing the right abstractions
  - Choosing the right "computer" for the task
- The computer is incredibly fast, accurate, and stupid
  - Human is unbelievably slow, inaccurate, and brilliant
  - The marriage of the two is a force beyond calculation (Leo Cherne)

Problem Solving

- Problem types (in informal categorization)
  - Easy problems
    - See the answer quickly
    - Most exercises you are doing in lab
  - Medium problems
    - See the answer once you engage
    - Most exercises you are doing in assignments
  - Hard problems
    - You need strategies for coming up with a potential solution, and sometimes for even getting started

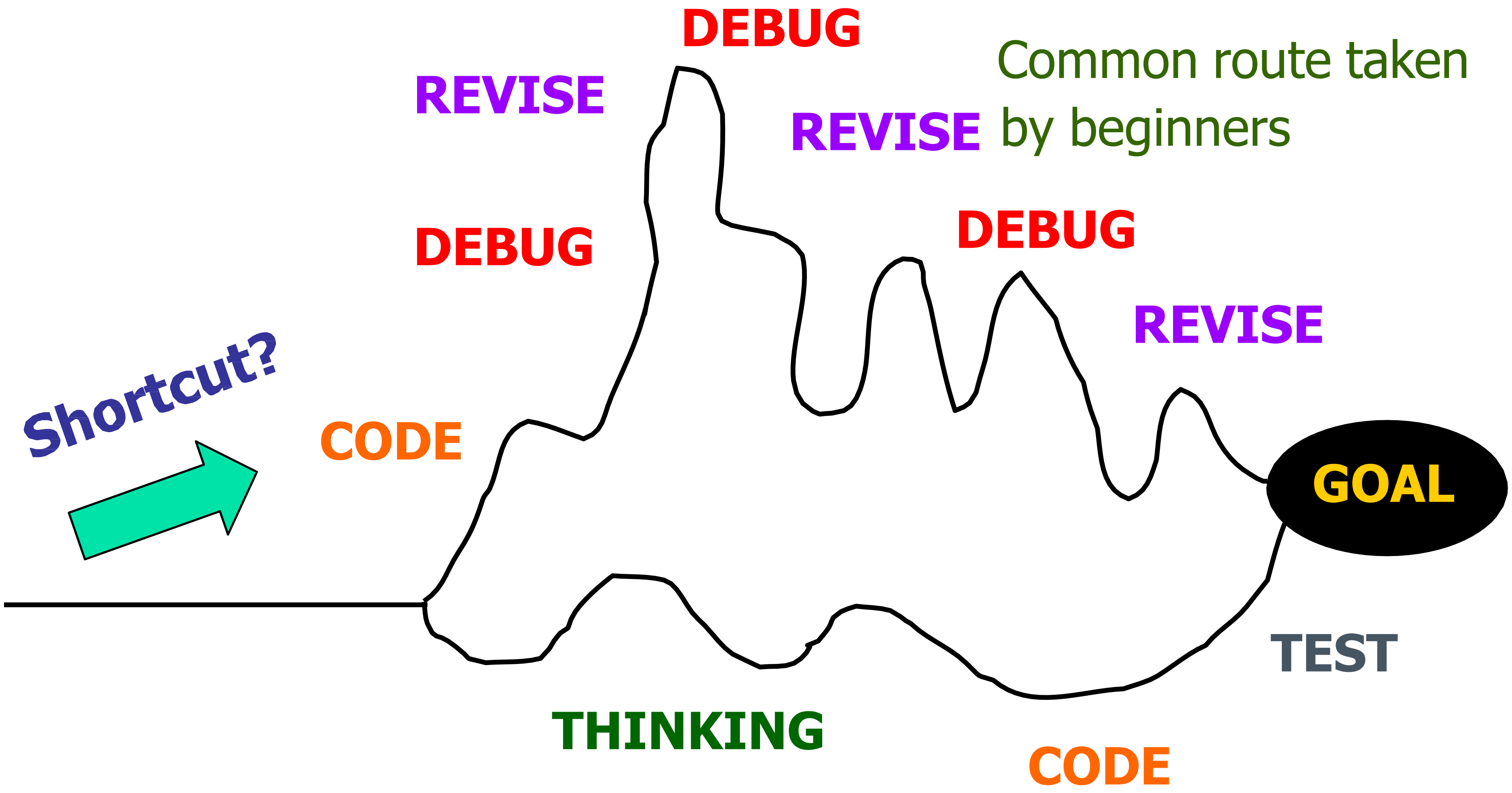
Problem Solving

- A good problem solver should possess
  - Positive attitude
    - Believe that the given computational problems can be solved through persistence
  - Concern for accuracy
    - Actively work to check and make sure you understand
  - Break the problem into parts
    - Top-down approach / Bottom-up approach / divide-and-conquer
  - Avoid guessing
    - Do not jump to conclusions too quickly
  - Active in problem solving
    - Do and practice more

Problem Solving

- Common problem-solving strategies
  - What do I know about the problem?
  - What is the information that I have to process to find the solution?
  - What does the solution look like?
  - What sort of special cases exist?
  - Similar problems come up again and again in different forms: do not reinvent the wheel
  - Divide-and-conquer
    - break up a large problem into smaller units and solve each smaller problem respectively

Shortcut?


DEBUG
REVISE
Common route taken by beginners
REVISE
DEBUG
DEBUG
REVISE
Shortcut?
CODE
GOAL
TEST
THINKING
CODE

Example 1 – Throwing a die

- Count the number of times for each face of a die

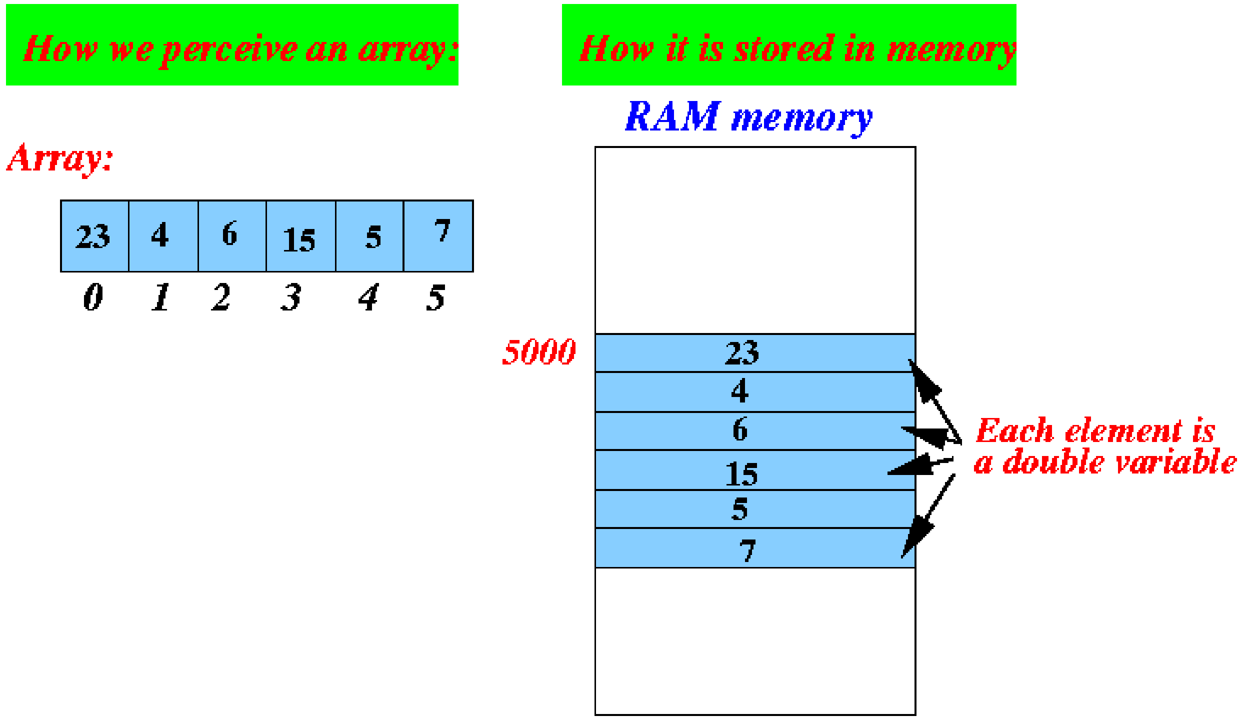
```
reset count1 to count6 value to zero
for i in [1…1000] do
    face = throw a die
    if face = 1 then count1 = count1 + 1
    if face = 2 then count2 = count2 + 1
    if face = 3 then count3 = count3 + 1
    if face = 4 then count4 = count4 + 1
    if face = 5 then count5 = count5 + 1
    if face = 6 then count6 = count6 + 1
```

- More efficient ways exist, and the ways also depend on the programming language

Example 1

› Solution using an *array or a list*

```
declare count[1…6]
initialize all count elements to 0
for i in [1…1000] do
   face = throw a die
   count[face] = count[face] + 1
```



Example 1 – Throwing a die

› This works well since face is between 1 to 6
  - In most programming languages, in which the index is zero-based, you would need to write

    count[face-1] = count[face-1] + 1
  - Or you can define count[0…6] instead of count[0…5] and ignore/waste the space for count[0]

Example 2 – Counting digits in a text

- Count the number of times for each digit in a text

```
declare count[0…9]
initialize all count elements to 0
for c in text do
   if c is a digit then
     count[c] = count[c] + 1
```

- But *c* is a character!
- We need to
  - check whether *c* is a digit
  - convert *c* into an integer in order to add to count[]

Example 2

Many ways to solve the same problem in Python!

- Checking if c is a digit in Python:

```
if c in ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']: # list
if c in ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9'): # tuple
if c in {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}: # set
if c >= '0' and c <= '9':
if '0' <= c <= '9':
if c.isdigit():
```

- Adding to count[c] in Python:

```
count[int(c)] = count[int(c)] + 1
count[eval(c)] = count[eval(c)] + 1
count[ord(c) - ord('0')] = count[ord(c) - ord('0')] + 1
```

Example 3 – Counting letters in a text

› Count the number of times for each letter in a text

```
declare count[0…25]
initialize all count elements to 0
for c in text do
  if c is a letter then
    convert c to lower case (or upper case)
    count[c] = count[c] + 1
```

› We need to
  – Check whether c is a letter
  – Convert c into lower case (or upper case)
  – Map c into a number 0 to 25 so as to add to the count[c]

Example 3 – Counting letters in a text

- Checking if c is a letter in Python:
  - if c in ['A', 'B', … , 'Z', 'a', 'b', … , 'z']: # list or tuple or set
  - if c >= 'A' and c <= 'Z' or c >= 'a' and c <= 'z': # *and* is evaluated before *or*
  - if 'A' <= c <= 'Z' or 'a' <= c <= 'z':
  - if c.isalpha():
- Converting c into lower case in Python
  - c.lower() # c.upper() into upper case
- Adding to count[c] in Python:
  - count[ord(c) - ord('a')] = count[ord(c) - ord('a')] + 1
- It may be more efficient to convert c into lower case before checking if c is a letter
  - Python also allows conversion of the whole text string to lower case in one step

Example 4 – Counting "terms" in a text

› Count the number of times for each interesting term in a text (assume at most *max* terms)

```
declare count[0...max-1]
initialize all count elements to 0
# cat = 0 dog = 1 cow = 2 sheep = 3 ...
for term in text do
        count[term] = count[term] + 1
```

› We need to
  – Map term into a number 0 to max - 1 so as to add the count

Example 4 – Counting “terms” in a text

- Adding to count[term] in Python:

```
for i in range(max):
    if term == table[i]:
        count[i] = count[i] + 1
        break
```

- Is there a better way?
  - Yes, in Python!
  - Using a Python dictionary to represent the count structure, then *term* can be directly used as an index to count

    count[term] = count[term] + 1
  - Dictionary could also be used to count the 26 letters, or 52 letters (with case sensitive counting)

Problem Solving Steps

- Abstraction
  - Perform data modeling
- Algorithm design
  - Design pseudo-code
- Implementation
  - Develop the program using a suitable programming language, test it and run it

Abstraction

- Abstraction
  - A model of a complex system that includes only the details essential to the viewer
- The process of abstraction can also be referred to as modeling
  - Build a proper model
    - 3D model?
- Three types of abstraction
  - Data abstraction
  - Procedural abstraction
  - Control abstraction

Abstraction

- Data abstraction
  - Separation of logical view of data from their implementation
    - You may not want to know how a list in Python or a 2D array in C/C++ is implemented in the computer systems
- Data abstraction is partially provided by the programming language
  - Provision of good and handy data types (e.g., dictionary)
  - Much need for programmer because the implementation itself is tedious
    - Structure definition and proper memory management

Abstraction

- Procedural abstraction
  - Separation of logical view of actions from their implementation
    - You may not want to write the detailed steps to do something
    - Just making a single reference to the action (i.e., a function in Python or a procedure in some other languages)
- Procedural abstraction is partially provided by the programming language
  - Provision of good libraries and modules
  - In a company, programmers also define common procedures/functions, often collected into code library to be reused
  - Proper use of functions help much in simplifying the program design

Abstraction

- Control abstraction
  - Separation of logical view of control structures from their implementation
    - You may not want to know how a complex expression (involving parentheses) is evaluated step by step in elementary operations
      - E.g., y = ((3 + x) * 5) mod 9
      - E.g., How is “if” implemented?
- Control abstraction is often provided by the programming language
  - Provision of useful and high-level operators/keywords

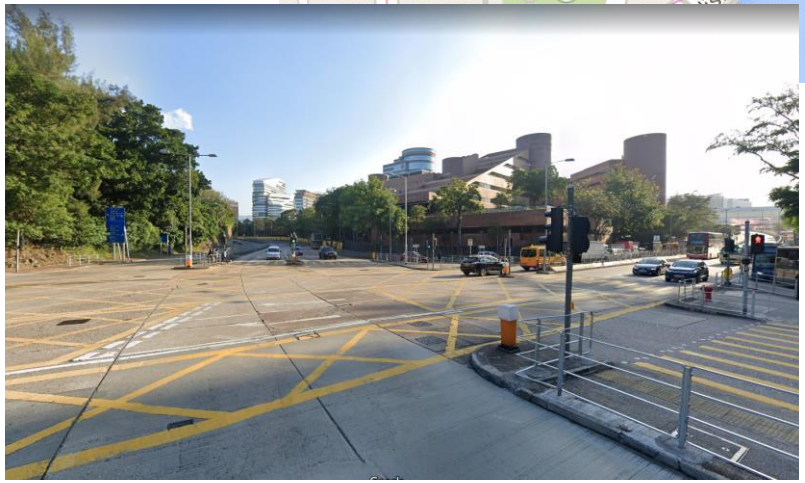
Abstraction

Keep only essential data in model
Remove irrelevant information!

- Example 1
  - Sort these animals from largest to smallest
    - elephant, ant, bee, whale, cat, dog, lion, tiger, panda, cow, dolphin, frog, bear, butterfly, snake, rabbit, eagle, penguin, ostrich, zebra
  - Does it matter how many animals are there?
  - Does the color of the animals matter?
  - What does it mean by “large”/”small”?
  - Abstraction
    - Take out only the key useful/essential information
      - Get their size and sort in size (sorting number)
      - Get their weight and sort in weight (sorting number)
      - Get both size and weight and sort in the combined score (sorting computed number)

Abstraction

- Example 2
  - How to get from PolyU to TST MTR station in map?
  - Shortest distance?
  - Shortest time?
- Abstraction
  - Regardless of where, the problem can be simplified:
    - Get the road junctions and roads
    - Mark the distance on each road segment
    - Find a shortest travelling path from A to B



Abstraction

- A representation of this information is a road network (a graph)
  - Nodes represent the road junctions (and starting/ending locations)
  - Edges represent the road segments
  - Weights associated with edges represents the travel distance or time
- Variations
  - Must starting/ending location be at a junction?
  - Is the time from A to B same as from B to A?
  - Are there roads (by car) only for one direction?

Abstraction

- Based on this (data) abstraction
  - The same algorithm can be used to solve the travelling problem in any city and any location
- A suitable algorithm can be developed
  - Input
    - A road network with nodes, edges and weights
    - Start and end points (nodes)
  - Output
    - A shortest path from start point to end point

Summary

› Introduction to Problem Solving

› How to be a good problem solver?

› Problem Solving Steps

  – Abstraction
  – Algorithm design
  – Implementation

› Abstraction

  – A closer look
  – Examples