

Problem Solving Principles and Abstraction

Alexander Lam

Problem Solving

- How to solve it? (George Pólya)
 - Understand the problem
 - What are you asked to find?
 - Draw 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
 - Review and reflect, also for future.

Problem Solving

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

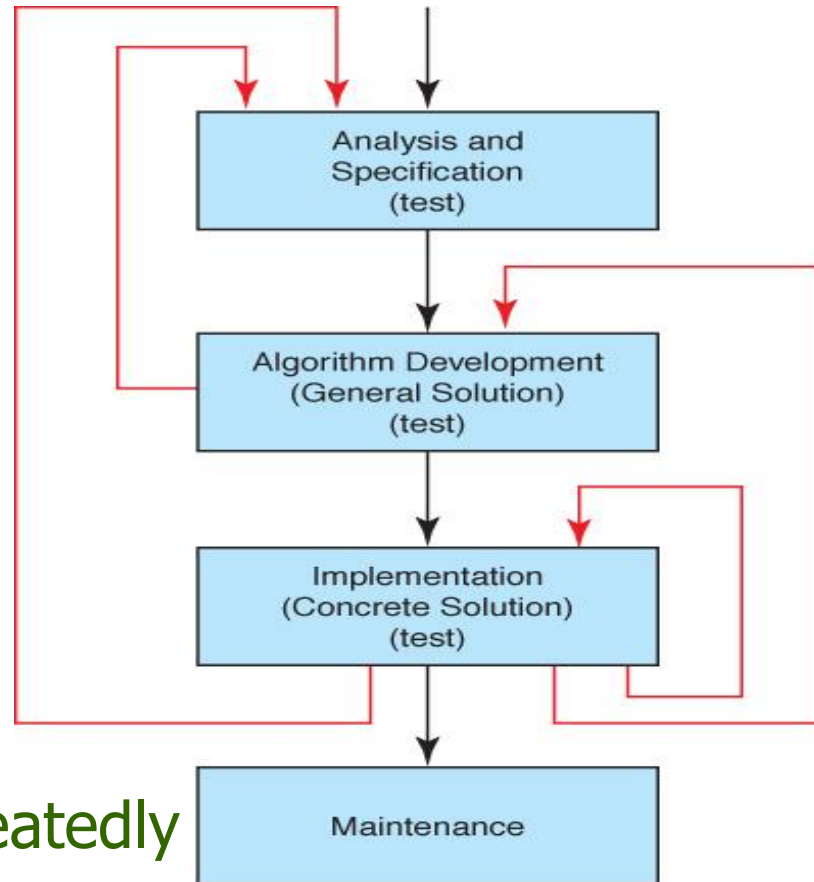
Problem Solving

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

Problem Solving

- Another view

Feedback loop
at each step



If to be used repeatedly

Problem Solving

- In **computing**, we **automate** our **abstractions**.
 - Once automated, computer can help solving the problem via many **repeated steps**.
 - A computer never gets tired, even if you ask it to count from 1 to 1 billion!
- **Computational thinking** involves:
 - Choosing the **right abstractions**.
 - Having the **right algorithm** for the task.
- “The computer is incredibly **fast**, **accurate**, and **stupid**. Humans are unbelievably **slow**, **inaccurate**, and **brilliant**. The marriage of the two is a force beyond calculation” - Leo Cherne.

Problem Solving

- Problem types
 - **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 the assignments.
 - **Hard** problems
 - You need **strategies** for coming up with a potential solution, and sometimes for even getting started.
 - Challenging exercises.

Problem Solving

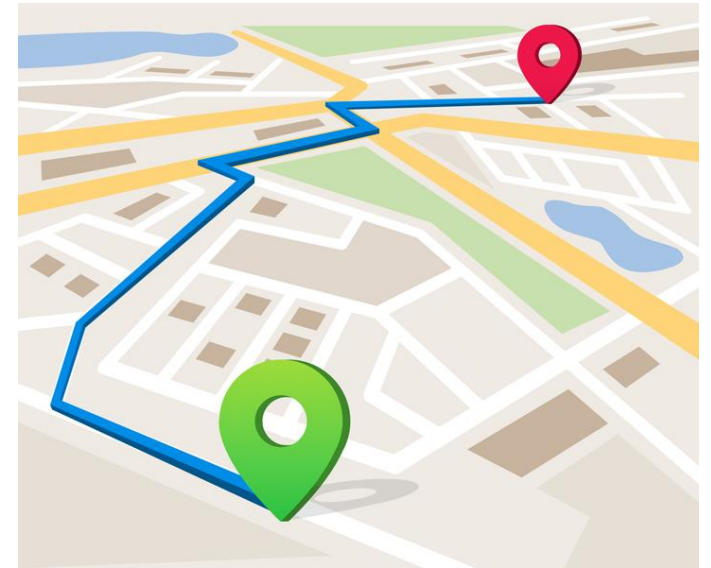
What we get taught in school

What comes in the exam



Recall: Examples of Problems

- Multiplying two numbers
- Finding the optimal route
- Predicting the Stock Market
- Verifying that you are a real person



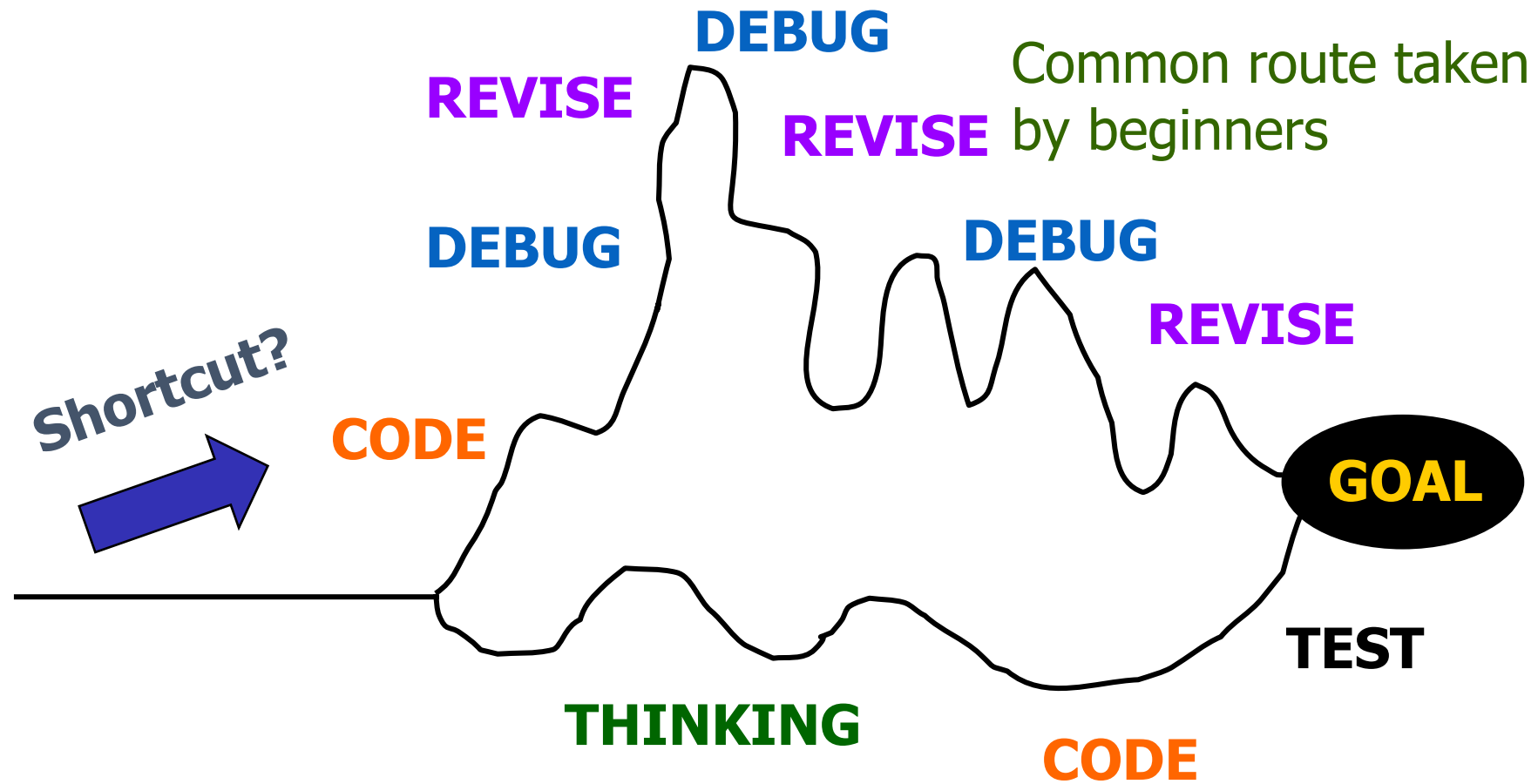
Problem Solving

- Attributes of a **good problem solver**
 - **Positive Mental Attitude (PMA)**
 - 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 / divide-and-conquer.
 - **Avoids 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?



Case Study

- You are playing online Monopoly with your friends, and Karen has just gone bankrupt.
- Angrily, she declares that the game is rigged and unfair.
- You try to remind her that Monopoly is a metaphor for life, and is intended to be rigged and unfair.
- She says she thinks that the dice is unfair, and keeps rolling ones.
- Although you try to explain that this is just confirmation bias, she wants you to conduct rigorous simulations to test if the dice is rigged and unfair.



Example Pseudocode

```
set count1, count2, ..., count6 = 0
for i in [1..1000] do
    face = throw a dice
    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
```

- Lots of repeated lines
- 6 different variables count1 to count6
- Is there a more concise and efficient method?

Example Pseudocode with an Array

```
declare count[0..5]
initialize all count elements to 0
for i in [1..1000] do
    face = throw a die
    count[face-1] = count[face-1] + 1
```

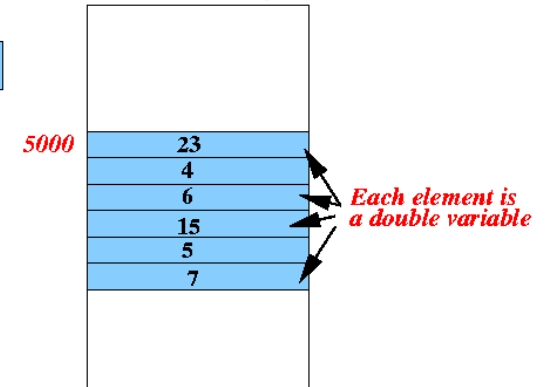
How we perceive an array:

Array:

23	4	6	15	5	7
0	1	2	3	4	5

How it is stored in memory

RAM memory



- This works well since face is between 1 to 6.
 - If you use C/C++, **count[6]** would mean **count[0..5]**.
 - If you use Python, **count[0:6]** is a count with 6 elements (from 0 to 5), but it is a list, not an array.

Counting Digits in Text

- What if someone gives us the following text and asks us how many times each (numerical) digit appears?

```
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
```

```
ihb5r49832756by34098534yntr
whrewutg98tb3458u34
threighsriughsgerybt342q98t0
bgafsd8asbdfw87rbt23r7
```

- But each `c` is a character!
- We need to:
 - Check if `c` is a digit
 - Convert `c` to an integer (to act as the array index for `count[c]`)

Counting Digits in Text

- 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`

Counting Digits in Text

- Checking if `c` is a digit in `C`:
 - if (`c >= '0' && c <= '9'`)
 - if (`isdigit(c)`)
- Adding to `count[c]` in `C`:
 - `count[c-'0'] = count[c-'0'] + 1` // `ord()` is not needed - `char` means byte/small int in `C`
- `C` allows conversion from a string to an integer, but not from a character:
 - `value = atoi(str)` // only for a `string str`

Counting Letters in Text

- What if someone gives us the following text and asks us how many times each **letter** appears?

iuhb5r49832756by34098534yntr
whrewutg98tb3458u34
threighsriughsgerybt342q98t0
bgafsd8asbdfw87rbt23r7

```
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 to act as the index for `count[c]`.

Counting Letters in 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.

Counting Letters in Text

- Checking if `c` is a letter in `C`:
 - `if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) // parentheses should be used`
 - `if (isalpha(c))`
- Converting `c` into lower case in `C`:
 - `c = tolower(c) // c = toupper(c) for upper`
- Adding to `count[c]` in `C`:
 - `count[c-'a'] = count[c-'a'] + 1`

Counting Keywords in Text

- What if we want to count the number of times each *keyword* shows up in a text.
 - Assume we have at most *max* keywords
 - This problem shows up a lot in AI (Natural Language Processing)

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

- We need to map the keyword to a number from 0 to max-1

Counting Keywords in Text

- Adding to `count[keyword]` in Python:
 - We have a table that contains our keywords

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

- Is there a better way?
 - A Python dictionary lets us directly use each keyword as an index to `count`.
 - `mydict = {"cat" : 0, "dog" : 1, "cow" : 2, "sheep" : 3}`
 - `count["dog"]`
 - `count[keyword] = count[keyword]+1`
 - A dictionary could also be used to count the 26 letters, or 52 letters (with case sensitive counting).
 - `mydict = {"A" : 0, "B" : 1, "C" : 2, "D" : 3}`
 - `count["A"]`

Counting Keywords in Text

- Adding to `count[keyword]` in C:

```
for (i = 0; i < max; i++)  
    if (strcmp(keyword, table[i]) == 0) {  
        count[i] = count[i] + 1;  
        break;  
    }
```

- Is there a better way?
 - Not really.
 - You may maintain a [sorted table](#) for binary search.
 - You may build a [hash table](#) for the terms for faster lookup.
 - Interestingly, the internal implementation of Python [dictionary](#) is based on hash table.

Counting Keywords in Text

- How to define the `count` array?
 - In `C`: array content not initialized
 - `int count[26];`
 - `int *count; // a pointer`
`count = (int*)malloc(sizeof(int)*26);`
 - `int *count;`
`count = (int*)calloc(26,sizeof(int));`
 - In `Python`: only list, no array
 - `count = [0, 0, 0, ... , 0]`
 - `count = [0]*26`
 - `count = list()`
`for i in range(26):`
`c.append(0)`
 - `count = [] # can also use count=list()`
`for i in range(26):`
`c.insert(i, 0)`

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.
- Three types of abstraction
 - Data abstraction
 - Procedural abstraction
 - Control abstraction

Abstraction

- Data abstraction

- Separation of logical view of data from their implementation.
 - You don't need to know how a list in Python or a 2D array in C is implemented within the memory structure.

- Procedural abstraction

- Separation of logical view of actions from their implementation.
 - You don't need to write the detailed steps to do something. Instead just make a single reference to the action (i.e. a function in Python or a procedure in some other languages).

- Control abstraction

- Separation of logical view of control structures from their implementation.
 - You don't need to know how a complex expression (involving parentheses) is evaluated step by step in elementary operations.

Abstraction

- **Data abstraction** is partially provided by the programming language.
 - Provision of good and handy **data types** (e.g. dictionary).
 - Much needed for programmers and this is the hardest part.
- **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.
- **Control abstraction** is often provided by the programming language.
 - Provision of useful and high level **operators**.

Abstraction

Keep only **essential** data in model
Remove **irrelevant** information

- Example problem
 - **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?
 - **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 problem
 - How to get from PolyU to TST MTR station?
 - Shortest distance?
 - Shortest time?
- Abstraction
 - 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**.
 - **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
 - Does the **starting/ending** location need to 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.
 - **Consider**
 - How precisely can the road network be represented?



