w02-Lec2

Problem Solving

for 204111

by Kittipitch Kuptavanich

Problem Solving

- ทักษะที่สำคัญที่สุดของ **computer scientist** คือ ***problem solving***.
  - **the ability to formulate problems,**
    - ความสามารถในการกำหนดปัญหา
  - **think creatively about solutions,**
    - การคิดวิธีการแก้ปัญหาอย่างสร้างสรรค์
  - **and express a solution clearly and accurately.**
    - การแสดงวีธีการแก้ปัญหาอย่างชัดเจน และถูกต้อง

Problem-Solving with Programming

1. First, you have to understand the problem.
   - เข้าใจปัญหา
2. After understanding, then make a plan.
   - วางแผน
3. Carry out the plan.
   - ทำตามแผน
4. Look back on your work. How could it be better?
   - ตรวจสอบผลที่ได้ ตลอดจนกระบวนการในการแก้ปัญหา ว่ามีสิ่งที่ควรปรับปรุงหรือไม่

Step 1: Understand the Problem

- กำหนดปัญหาให้ชัดเจน
- ไม่ควรข้ามขั้นตอนนี้
  - มักถูกมองข้ามเนื่องจาก อาจเห็นว่าชัดเจนอยู่แล้ว
- คำถามที่ควรถาม
  - โจทย์ให้หาอะไร หรือให้แสดง/พิสูจน์อะไร
  - ลองอธิบายปัญหาด้วยถ้อยคำของตัวเองตามความเข้าใจได้หรือไม่
  - สามารถวาดรูป หรือไดอะแกรม เพื่อทำให้เข้าใจมากขึ้นได้หรือไม่

Step 1: Understand the Problem [2]

- **<u>คำถามที่ควรถาม</u> (ต่อ):**
  - **ณ ขณะนี้ มีข้อมูลเพียงพอที่จะหาทางแก้ปัญหาหรือไม่**
  - **เข้าใจคำทุกคำที่อยู่ในโจทย์หรือไม่**
  - **จำเป็นต้องถามคำถาม หรือหาข้อมูลอะไรเพิ่มหรือไม่ ก่อนที่จะหาคำตอบได้**

Step 2: Devise a Plan

แก้ปัญหาให้ได้ก่อนโดยไม่ต้องใช้การเขียนโปรแกรม

1. ใช้กลยุทธ์การแก้ปัญหาที่กล่าวมา
   - **Abstraction, Analogy, Reduction, Trial-and-error, Proof, etc...**
2. พิจารณาทางแก้ปัญหาหลาย ๆ ทาง
   - คุณสมบัติที่ต้องการ: ความชัดเจน ความไม่ซับซ้อน ประสิทธิภาพ ความสามารถในการนำไปประยุกต์ใช้กับกรณีอื่น ๆ
   - **Future concerns (after more CS courses): usability, accessibility, security, privacy, testability, reliability, scalability, compatibility, extensibility, durability, maintainability, portability, provability, ...**

Step 2: Devise a Plan

3. เขียนวิธีแก้ปัญหาในลักษณะที่สามารถนำไปแปลงเป็น Code ได้สะดวก

- ใช้ขั้นตอนที่สั้นและชัดเจน ไม่จำเป็นต้องอาศัยการตัดสินใจหรือการตีความเพิ่มเติม
- ทำให้การแก้ปัญหาดังกล่าว ใช้ได้กับ Input ขนาดใหญ่ ($n \to \infty$)
- ไม่ต้องอาศัยความจำของผู้อ่านในการแก้ปัญหา
  - เขียนออกมาให้ชัดเจนว่าจำเป็นต้องจำอะไรบ้าง
  - แล้วตั้งชื่อสิ่งเหล่านั้นให้สื่อความหมาย (เพื่อนำสิ่งที่ต้องจำเหล่านั้นนำมาแปลงเป็น Variable เมื่อต้อง Code)

Step 3: Carry out the Plan

Translate your solution into code

1. **เขียน Test Case (ควรทำเป็นสิ่งแรก ไม่ใช่สิ่งสุดท้าย)**
2. **แปลงวิธีการแก้ปัญหา จาก Step 2 ทีละขั้นตอน ให้เป็น Code**
   - **ใช้วิธีแบ่งปัญหาจากปัญหาใหญ่เป็นปัญหาเล็ก Top-down Design (Divide and Conquer)**
3. **ทดสอบวิธีแก้ปัญหา (Robotically and Manually)**
   - **หากรณีขอบเขต และ กลุ่ม (Class) ของ Input**
   - **มองหา Case ยกเว้นต่าง ๆ เช่น หารด้วย 0 หรือ Input มีขนาดใหญ่มาก ๆ ($n \to \infty$) หรือเล็กมาก ๆ (เช่น $0 < n < 1$)**

Step 4: Examine and Review

- พิจารณาวิธีแก้ปัญหาที่ได้ด้วย **Common Sense**
- อภิปราย/พิจารณา/วิพากษ์ ข้อดีข้อเสียของ **Solution** ที่ได้กับคนอื่น ๆ
  - ไม่แนะนำให้ทำกับ **assignment** ที่ต้องทำเป็นงานเดี่ยว
- เก็บ **solution** ไว้ในที่ ๆ หาได้สะดวก (เพื่อการนำกลับมาใช้อีก)

Basic Program Instructions (Recap)

A few basic instructions appear in just about every language:

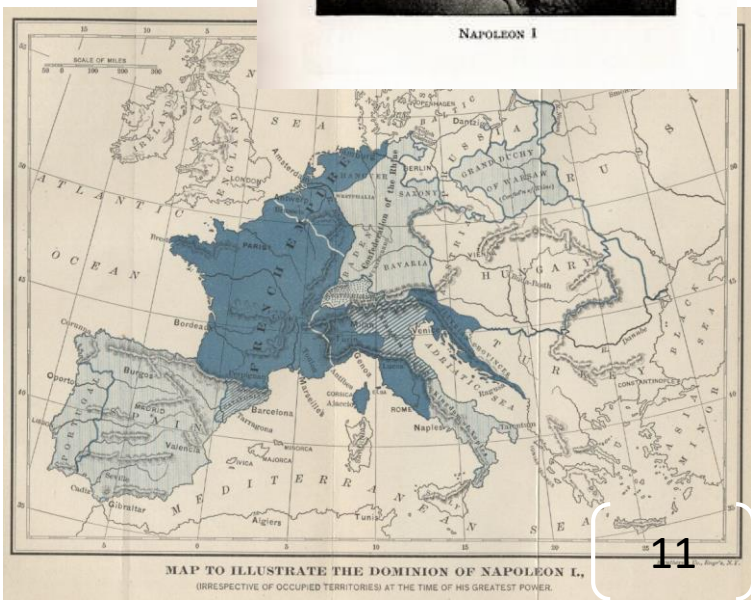
- **Input**
- **Output**
- **Math**
- **Conditional Execution**
- **Repetition**

You can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

Divide and Conquer

- **หรือ Divide and Rule**
  - **ในทางประวัติศาสตร์และการปกครอง คือการสร้างอำนาจหรือรักษาอำนาจไว้โดยวิธี**
    - **แบ่งเป้าหมายเป็นหน่วยเล็ก ๆ ที่มีกำลังน้อยกว่า (Divide)**
    - **แล้วเข้ายึดอำนาจทีละส่วน (Conquer)**

NAPOLEON I



MAP TO ILLUSTRATE THE DOMINION OF NAPOLEON I.,
(IRRESPECTIVE OF OCCUPIED TERRITORIES) AT THE TIME OF HIS GREATEST POWER.

Divide and Conquer [2]

- การแก้ปัญหาโดยใช้หลัก Divide and Conquer มี 3 ขั้นตอน
  - Divide แบ่งปัญหาที่ต้องการแก้ เป็นปัญหาย่อย (Subproblem) - ควรแบ่งแล้วปัญหาเล็กลงหรือซับซ้อนน้อยลง
  - Conquer แก้ปัญหาย่อย
  - Combine นำคำตอบของปัญหาย่อยมารวมกันเพื่อให้ได้คำตอบของปัญหาหลัก

Top-Down Design

- เขียนฟังก์ชันจากใหญ่ไปเล็ก
  Write functions top-down
  - ให้สมมติว่า **Helper Function** ต่าง ๆ เขียนเสร็จแล้ว
    Assume helper functions already exist!
- ทำการ **Test Function** จากเล็กไปใหญ่
  Test functions bottom-up
  - ไม่นำฟังก์ชันใด ๆ มาเรียกใช้ จนกว่าจะผ่านการ **test** อย่างถี่ถ้วน
    Do not use a function before it has been thoroughly tested
- ในทางปฏิบัติสามารถเขียน **Stub Function** มาใช้ชั่วคราวขณะทำ **Top-down Design**
  Practicality: May help to write stubs (simulated functions as temporary placeholders in top-down design)

Reference

- http://en.wikipedia.org/wiki/Problem_solving
- http://en.wikipedia.org/wiki/How_to_solve_it
- http://www.kosbie.net/cmu/spring-12/15-112/handouts/notes-problem-solving.html