CMU 15-112: Fundamentals of Programming and Computer Science
Class Notes: Algorithmic Thinking

1. **Optional Reading**
    1. Wikipedia page on Polya's "How to Solve It" (http://en.wikipedia.org/wiki/How_to_solve_it)
        - Or you can read the entire book: Polya, How to Solve It (http://amzn.com/4871878309)
    2. Wikipedia section on Common Barriers to Problem Solving (https://en.wikipedia.org/wiki/Problem_solving#Common_barriers_to_problem_solving)

2. **Problem Solving with Programming**
    1. Understand the Problem (read the prompt!)
        1. Define the problem precisely.
        2. Generate test cases based on the prompt, or carefully read through test cases if they're provided.
        3. Make sure you can restate the problem in your own words, or explain it to someone else.

    2. Devise a Plan (solve the problem without programming!)
        1. Identify what level of problem-solving will be required for the problem. Is the prompt asking you to...
            - **Translate** a provided algorithm into code? You can skip to the Step 3.
            - **Apply** a known algorithm pattern to the problem? You can skip to Step 2.4.
            - **Generate** an algorithm to solve a problem? Keep reading!
        2. Use problem-solving strategies to build an algorithmic approach. There are several strategies you can apply while trying to solve a problem. Here are three common programming strategies:
            - **Induction**: Investigate several examples (test cases) to find a pattern that can be generalized into an algorithm.
            - **Top-Down Design**: Break down a complex problem into several simpler problems, then solve each of the simpler problems individually. See more here (topDownDesign)
            - **Human Computer**: Pretend that you need to carry out the task by hand. Determine the steps you would take in order to complete the task.
        3. Compare possible alternative algorithms. Algorithms can be ranked based on many features, including:
            - **Clarity**: How clear is the approach to you?
            - **Simplicity**: How straightforward will the approach be to implement?
            - **Efficiency**: How much work will the algorithm need to do?

        - **Generality**: How well will this algorithm adapt to new inputs?
        - **Future concerns (after more CS courses)**: usability, accessibility, security, privacy, testability, reliability, scalability, compatibility, extensibility, durability, maintainability, portability, provability, ...
    4. Choose an algorithm and write your solution so it is amenable to being translated into code. You shouldn't write in code or even pseudocode yet, but you should use explicit, small, and clear steps that do not require human ingenuity, intuition, or memory.
        - If you need to remember a thing, give it a name- those names will later become variables.

3. Carry out the Plan (translate your solution into code)
    1. Write the test cases that you generated in Step 1.
    2. Translate your manual solutions from above, step by step, into code. If you already know how to translate a step into code, do so! Otherwise:
        - Review the relevant course materials to find a programming tool that approximately meets your need.
        - Experiment with that programming tool in the interpreter until you understand how it works.
        - Apply the new concept to the step in your algorithm.
    3. Run your function on a test input to make sure it works. If your code has a syntax or runtime error, use debugging to identify the problem and fix it.

4. Review your Work (test your code)
    1. Run your test cases on the code to make sure they all pass. If a test case fails, use debugging to identify the algorithmic problem and fix it.
    2. Once all test cases pass, read over your code and make sure it makes sense based on common sense.
    3. (After the deadline) Discuss and compare your solution with others to learn about alternative approaches to solving the problem.

3. **Programming Patterns**
    - There are many algorithmic patterns that appear in various problems, patterns which can be adapted to suit new circumstances at need. We'll see many of these patterns in class; for now, here are a few common patterns from the first weeks of class.

    - **nthSomething**

```
def nthSomething(n):
    found = 0
    guess = 0
    while found <= n:
        guess += 1
        if isSomething(guess):
            found += 1
    return guess
```

Copy Visualize ▶ Run

    - **propertyTrueForAll**

```
def propertyTrueForAll(s):
    for c in s:
        if property(c) == False:
            return False
    return True
```

Copy Visualize Run

- **countProperty**

```
def countProperty(s):
    count = 0
    for c in s:
        if property(c) == True:
            count += 1
    return count
```

Copy Visualize Run

- **mostCommonItem**

```
def mostCommonItem(s):
    bestCount = 0
    bestItem = None
    for c in s:
        tmpCount = count(s, c)
        if tmpCount > bestCount:
                bestCount = tmpCount
                bestItem = c
        elif tmpCount == bestCount and c > bestItem:
                bestItem = c
    return bestItem
```

Copy Visualize Run

4. **Top-Down Design**
   - We often ask you to solve problems that are too complicated to be solved by one function alone. In these cases, you will need to break the problem down into sub-problems that can be mapped directly to helper functions.
   - To identify a sub-problem, start writing the algorithm out. When you get to a step that seems too complicated, imagine you have a helper function that can solve that step. If you can write the function by including a call to this imaginary function, you've taken a step towards solving the problem! The process can then be repeated on the helper functions until the solution is complete.
   - When testing multi-function problems, test the simplest functions first. It's important to ensure that your helper functions are working before you move on to test the main function.

   - **Example: nthThreeDigitPrime**

```
# First, we can write the part that we know how to do already: a basic n
th function.
def nthThreeDigitPrime(n):
    found = 0
    guess = 0
    while found <= n:
        guess += 1
        if isThreeDigitPrime(guess):
            found += 1
    return guess

# Next, break down the problem further
def isThreeDigitPrime(n):
    return hasThreeDigits(n) and isPrime(n)

# Once the helper functions seem simple enough, write them directly.
def hasThreeDigits(n):
    return 100 <= n <= 999

def isPrime(n):
    if n < 2:
        return False
    for factor in range(2, n):
        if n % factor == 0:
            return False
    return True

# Start testing with isPrime and hasThreeDigits, then isThreeDigitPrime.
# Finally test nthThreeDigitPrime, and you're done!
```

Copy Visualize ▶ Run

5. **Practice**
    - In the end, the best way to learn problem solving is to **practice with feedback**. If you're struggling with problem solving, ask a TA or a friend to sit with you while you while you work through designing algorithms for practice problems, or ask them to demonstrate how they approach problem solving themselves. Practice should eventually lead to better understanding of how to approach problem solving in general.