Algorithms: Heart of Computer Science

- **Algorithm**: A step-by-step procedure for how to do something (usually a calculation).
- **Implementation**: How to write an algorithm in a specific programming language
- Good programmers know how to separate the two
  - Work out algorithm on paper or in head
  - Once done, implement it in the language
  - Limits errors to syntax errors (easy to find), not conceptual errors (much, much harder to find)
- Key to designing algorithms: **stepwise refinement**

---

Stepwise Refinement: Basic Principles

- **Write Specifications First**
  - Write a function specification before writing its body
- **Take Small Steps**
  - Do a little at a time; make use of placeholders
- **Run as Often as You Can**
  - This can catch syntax errors
- **Separate Concerns**
  - Focus on one step at a time
- **Intersperse Programming and Testing**
  - When you finish a step, test it immediately

---

Mañana Principle

- If not in current step, delay to “tomorrow”
  - Use comments to write steps in English
  - Add “stubs” to allow you to run program often
  - Slowly replace stubs/comments with real code
- Only create new local variables if you have to
- Sometimes results in creation of more functions
  - Replace the step with a function call
  - But leave the function definition empty for now
  - This is called **top-down design**

---

Using Placeholders in Design

- Delay do anything not immediately relevant
  - Use comments to write steps in English
  - Add “stubs” to allow you to run program often
  - Slowly replace stubs/comments with real code
- Only create new local variables if you have to
- Sometimes results in creation of more functions
  - Replace the step with a function call
  - But leave the function definition empty for now
  - This is called **top-down design**

---

Function Stubs

<table>
<thead>
<tr>
<th>Procedure Stubs</th>
<th>Fruitful Stubs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single statement: pass</td>
<td></td>
</tr>
<tr>
<td>Body cannot be empty</td>
<td></td>
</tr>
<tr>
<td>This command does nothing</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong></td>
<td></td>
</tr>
<tr>
<td>def fo():</td>
<td></td>
</tr>
<tr>
<td>pass</td>
<td></td>
</tr>
<tr>
<td>Single return statement</td>
<td></td>
</tr>
<tr>
<td>Type should match spec.</td>
<td></td>
</tr>
<tr>
<td>Return a “default value”</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong></td>
<td></td>
</tr>
<tr>
<td>def first_four_letters(s):</td>
<td></td>
</tr>
<tr>
<td>return '' # empty string</td>
<td></td>
</tr>
</tbody>
</table>

**Purpose of Stubs**

Create a program that may not be correct, but does not crash.

---

Example: Reordering a String

- **last_name_first('Walker White')** is 'White, Walker'

```python
def last_name_first(s):
    """Returns: copy of s in form <last-name>, <first-name>"
    Precondition: s is in the form <first-name> <last-name> with one blank between the two names"
    # Find the first name
    # Find the last name
    # Put them together with a comma
    return '"' # Currently a stub
```
Example: Reordering a String

- last_name_first('Walker White') is 'White, Walker'

```python
def last_name_first(s):
    '''Returns: copy of s in form <last-name>, <first-name>
    Precondition: s is in the form <first-name> <last-name>
    with one or more blanks between the two names'''
    end_first = s.find(' ')
    first_name = s[:end_first]
    # Find the first name
    # Put them together with a comma
    return first_name # Still a stub
```

Exercise: Anglicizing an Integer

- anglicize(1) is "one"
- anglicize(15) is "fifteen"
- anglicize(123) is "one hundred twenty three"
- anglicize(10570) is "ten thousand five hundred"

```python
def anglicize(n):
    '''Returns: the anglicization of int n.
    Precondition: 0 < n < 1,000,000'''
    pass # ??
```

```python
def anglicize(n):
    '''Returns: the anglicization of int n.
    Precondition: 0 < n < 1,000,000'''
    if n < 1000:
        # no thousands place
        return anglicize1000(n)
    elif n % 1000 == 0:
        # no hundreds, only thousands
        return anglicize1000(n/1000) + ' thousand'
    else:
        # mix the two
        return (anglicize1000(n/1000) + ' thousand ' +
                anglicize1000(n))
```

Exercise: Anglicizing an Integer