Monday: MLK Day
* Unclear about building access and lab access
* Best case:
  * Lab is open with TAs: 4—6 pm (Binh), 6—8 (Hoang)
  * Worst case: locked building, lab
    * TAs at Piazza
* Week 1 exercise submission deadline: 12pm, Tuesday

* All cs121 exercise and homework programs must in the default package and use the class names as given.

  * If you use eclipse, all java source code (.java files) need to be in the src folder, not the project folder.
Problem Solving Example

- **Flash Mind Reader**
  - Different tables at different runs
  - Apparent random choice for users
  - Final outcome
    - Deterministic for each table
    - Same symbol regardless of user choice
  - More information, including the proof, can be found in the complete slides (now liked from cs121 site)

- Problem solving processes: specializing, generalizing, conjecturing, convincing

- (Reminder: take Math 114 this semester, a prerequisite for several int. CS courses)
Some of the slides on testing are adopted from slides from

- *Big Java*, Horstmann
- *Introduction to Computing*, Sedgewick and Wayne
Errors

• Compile-time error: A violation of the programming language rules that is detected by the compiler
  • Example:
    
    System.out.println("Hello, World!");

• Syntax error

• Run-time error: Causes the program to take an action that the programmer did not intend
  • Examples:
    
    System.out.println("Hello, Word!");
    System.out.println(1/0);

• Logic error
Error Management Strategy

• Learn about common errors and how to avoid them
• Use defensive programming strategies to minimize the likelihood and impact of errors
• Apply testing and debugging strategies to flush out those errors that remain
Program Development Process

- Understand the problem
- Develop and describe an algorithm
- Test the algorithm with different inputs
- Translate the algorithm into Java
- Compile and test your program

Figure 12
The Program Development Process
The Waterfall Model

- Sequential process of analysis, design, implementation, testing, and deployment
- When rigidly applied, waterfall model did not work
The Spiral Model

- Problem: Can lead to many iterations, and process can take too long to complete
Debugging

is 99% of program development in any programming language, even for experts.

**Bug:** A mistake in a program.

**Debugging:** The process of eliminating bugs.

You will make many mistakes as you write programs. It’s normal.

“As soon as we started programming, we found out to our surprise that it wasn’t as easy to get programs right as we had thought. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.”

— Maurice Wilkes

**Impossible ideal:** "Please compile, execute, and debug my program."

Why is this impossible? Stay tuned.

**Bottom line:** Programming is primarily a process of finding and fixing mistakes.
Debugging is challenging because conditionals and loops dramatically increase the number of possible outcomes.

<table>
<thead>
<tr>
<th>program structure</th>
<th>no loops</th>
<th>$N$ conditionals</th>
<th>1 loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of possible execution sequences</td>
<td>1</td>
<td>$2^N$</td>
<td>no limit</td>
</tr>
</tbody>
</table>

Most programs contain numerous conditionals and loops, with nesting.

Good news. Conditionals and loops provide structure that helps us understand our programs.

Old and low-level languages have a `goto` statement that provides arbitrary structure. Eliminating `gotos` was controversial until Edsgar Dijkstra published the famous note "Goto considered harmful" in 1968.

“The quality of programmers is a decreasing function of the number of goto statements in the programs they produce.”

— Edsgar Dijkstra
A class to test the BankAccount class.

Tests the methods of the BankAccount class.
@param args not used

In essence, testing is done by comparing actual results generated by your program to expected results.

Program Run:
1500
Expected: 1500
At all stages of problem solving
- Understand the problem
  - e.g. For a small-sized problem, do I know what the expected results are?
  - Do I understand the information and format in the input data? ...
- Devise (and analyze) a plan
  - E.g. starting with a small problem, how I can compute the solution? By hand, as an algorithm... Does my solution produce expected results?
  - How to generalize ideas for small/special problems to the general problem and corner cases?
- Carry out the plan
  - Incremental development: thorough testing of developed methods before moving on to new ones
- Look back
  - reflect on informative (including failed) test cases, algorithms, coding, practices...
  - better understanding, more insights ⇒ better performances in the future
Every programmer knows they should write tests for their code.

- Few do.
- The universal response to "Why not?" is "I'm in too much of a hurry."
- This quickly becomes a vicious cycle -
  - the more pressure you feel, the fewer tests you write.
  - The fewer tests you write, the less productive you are and the less stable your code becomes.
  - The less productive and accurate you are, the more pressure you feel.

You are doing yourself, your partner, company... a **disservice** by not rigorously testing or analyzing your algorithm or code.
Proactive, thorough, documented testing at all stages, early and often

Tasks
- Develop Test cases
  - Given ones
  - Your own
- Figure Out Expected Results
- Compare Actual Results and Expected Results
Testing

- **Unit Test** (More on this later)
  - Verifies that a functional unit (such as a method, a class...) works correctly by itself

- **Test suite**: a collection of tests for repeated testing

- **Bug Cycling Phenomena**
  - Fixed bugs reappear in later versions, as an unwanted side effects of the attempts for fixing some other bugs

- **Regression testing**: repeating previous tests to ensure that known failures of prior bugs do not appear in new versions
Where to test?
  * Separate test classes (More on this later)
  * Save results to a file, compare that file to a file with expected results
  * ...

Note: don’t use main methods in Princeton programs as reference for your testing
  * Their main methods are more like demo, not testing
  * Their highly polished programs were tested
    * Thoroughly
    * somewhere else.
In essence, compare expected and returned results for each test case.

How to get expected results?
- Sample test cases from problem setters, assignment pages ...
- Manual computation for small test cases
- Large test cases
  - Carefully designed cases: compute expected results in a different way
  - ...

How to compare?
- Problem dependent
- Eyeball comparison of numerical values
  - Only feasible for small test cases, small numbers of tests

Automatic, systematic comparison
- Result file comparison: fc (on windows), diff (on linux, and apple?)...

interesting algorithmic issues
- Different representations for a same solution
- multiple solutions
Testing

What to test? This is the test coverage issue.

- In principle, complete coverage
  - Test all required functionality for all possible inputs (black box testing)
  - Run each part of your program at least once by one test case (white box testing)

- In practice, it takes
  - time
  - careful thinking
  - insights
  - skills, and
  - experiences
  to design good test cases

- In cs121
  - test your code proactively, thoroughly
  - Gain experience and develop expertise in testing and professional code development
  - Week 1 exercise: Three Sort
Debugging a program: a running example

**Problem:** Factor a large integer $N$.

**Application:** Cryptography.

**Surprising fact:** Security of internet commerce depends on difficulty of factoring large integers.

**Method**

- Consider each integer $i$ less than $N$
- While $i$ divides $N$ evenly
  - Print $i$ (it is a factor of $N$).
  - Replace $N$ with $N/i$.

**Rationale:**
1. Any factor of $N/i$ is a factor of $N$.
2. $i$ may be a factor of $N/i$.

```java
public class Factors {
    public static void main(String[] args) {
        long N = Long.parseLong(args[0]);
        for (i = 0; i < N; i++) {
            while (N % i == 0) {
                System.out.print(i + " ");
                N = N / i;
            }
        }
    }
}
```

This program has bugs!

Another version of buggy factorization code and debugging

- Additional Princeton Slides on this example in their file for Conditionals and Loops
Factors Testing Plan

* Easy numbers that we know the factors for
  * Small numbers, familiar numbers (e.g. \(1024=2^{10}\))
  * Examples

* Categories of numbers (think about test coverage: as complete as possible)
  * Prime: 2, 3, 5, 7, ..., 97
    * Expected output: the number itself
  * Composite: 4, 6, 8, 9, ..
    * All unique factors: 15 (3, 5), ...
    * Some repeating factors: 4 (2 \(2\)), 12(2 \(2\) 3)...

* Corner Cases:
  * (error cases: more on this later)
  * Smallest number for prime factorization: 2
  * Largest number: Long.MAX_VALUE (find the real value, its prime factors... )