# Computational Goals
## Correctness & Efficiency

# 6A

Correctness



## Software Errors

- Software errors consist of:
  - syntax errors (incorrect use of computer language)
  - runtime errors (invalid execution condition)
  - logical errors (incorrect computation/algorithm)
  - race conditions (more on this later)
- Modern software applications contain a huge amount of computer instructions (lines of code):
  - Mozilla (web browser): over 2 million
  - Red Hat Linux 7.1: over 30 million
  - Windows XP: over 40 million

Source: www.dwheeler.com (2002)

# First Computer Bug?

- Grace Hopper's team was working on the Harvard Mark II.
- In September 1945, a moth became trapped between the points of a relay in the system.
- The moth was removed and taped into the log with the following analysis: "First actual case of bug being found."

www.jamesshuggins.com

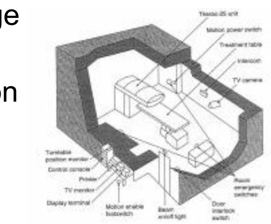


# Famous Software Errors

- Mariner I (1962)
  - The omission of a hyphen in coded computer instructions transmitted incorrect guidance signals to the spacecraft.
  - The program went automatically into a series of unnecessary course correction signals which threw spacecraft off course.
  - Spacecraft was destroyed 6 seconds before it would separate from its booster.
  - Cost: approx. $80 million

# Famous Software Errors

- Therac-25 (1985-1987)
  - Medical linear accelerator used to destroy tumors.
  - Software-only control to deliver specific doses to patient (approximately 200 rad).
  - Race condition software error caused machine to emit a dosage of about 15,000-20,000 rad.
  - Cost: At least 6 cases of radiation overdose, leading to 3 deaths. (Lawsuits settled out of court.)






# Famous Software Errors

- AT&T Software Bug (1990)
  - A switching node fails, sending a "out of service" message to neighboring nodes to reroute voice traffic.
  - The software running the switch was a misplaced "break" statement in C code.
  - The surrounding nodes also crashed and repeated the process to more and more nodes in the phone network.
  - Cost: 9 hours without long-distance service for an estimated 60 million people, and at least $60 million in lost revenue.

# Famous Software Errors

- Patriot Missile Failure (1991)
  - During the Gulf War, an American Patriot Missile battery in Saudi Arabia failed to track and intercept an incoming Iraqi Scud missile.
  - Cause of failure: Inaccurate recording of time, causing the patriot missile to miscalculate the intercept point and veer past the incoming scud.
  - Cost: Death of 28 soldiers and injury of 100 other people

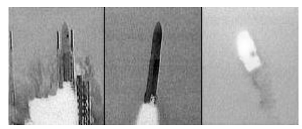


# Famous Software Errors

- Denver Airport (1994)
  - New automated baggage handling system was to be deployed with the opening of Denver's new airport.
  - Software errors caused a delay of 11 months.
  - Conveyors belts were jammed, carts were loaded with bags even though they were full, timing was not synchronized between carts and conveyors, causing bags to lodge underneath carts, ...
  - Costs: $1 million per day to the city of Denver (original project estimate: $234 million)

# Famous Software Errors

- Ariane 5 (1996)
  - Launched by the European Space Agency
  - Went out of control shortly after takeoff and exploded 40 seconds into flight
  - Cause of failure: data conversion software error in the inertial reference system
  - Project cost: over $7 billion





# Famous Software Errors

- Y2K Bug (2000)
  - Software written in the 1960s and 1970s represented years with 2 digits (e.g. 1975 would be stored as 75) to conserve expensive memory cells.
  - This software was still in use in the late 1990s, causing a panic in the software industry and throughout government, predicting doomsday scenarios.
  - Cost: Governments and businesses spent an estimated $500 Billion to repair Y2K bugs before the year 2000.

## The Need for Correctness

- Software bugs cost the U.S. economy approximately $59.5 billion annually (in a NIST 2002 report).
- Half of these costs are borne by software users.
- More than half of the discovered software errors are reported post-sale by the consumer/user.
- $22.2 billion can be saved by improving software testing and correctness methods.

Source: www.nist.gov/public_affairs/releases/n02-10.htm



## Verification

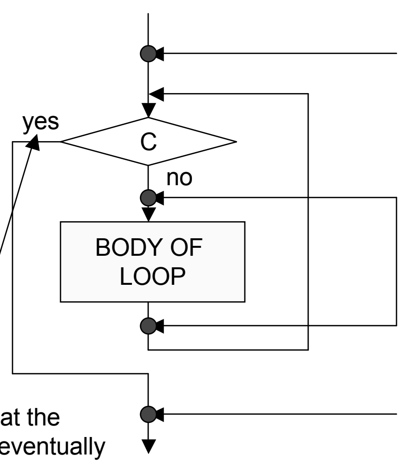
- Techniques have been developed for program verification for limited situations.
- To prove partial correctness, we can attach assertions to specific points in the software.
  - An <u>assertion</u> is a statement that should be true if the software reaches this point.
  - An assertion in the body of a loop is called an <u>invariant</u>.

# Showing Correctness: Loop Invariants

- Showing a loop is correct:
  0. Start with an expression for the loop invariant.
  1. Show that the invariant is true when the loop begins execution.
  2. Show the invariant is true at the start and end of each iteration of the loop.
  3. Show that the invariant and the loop condition together imply the answer is correct.
  4. Show that the exit condition of the loop must eventually become true (so loop terminates).



# Loop Invariant



# Loop Invariant Example

**Computing sum of 1+2+...+n, n > 0**

1. **Let sum = 1.**
2. **Let i = 2.**
3. **While i $\leq$ n do the following:**
   **a. Add i to sum.**
   **b. Add 1 to i.**
4. **Output sum.**

Loop invariant: $sum = 1 + ... + (i-1) = \sum_{z=1}^{i-1} z$



# Loop Invariant Example

**Computing sum of 1+2+...+n, n > 0**

1. **Let sum = 1.**
2. **Let i = 2.**
3. **While i $\leq$ n do the following:**
   **a. Add i to sum.**
   **b. Add 1 to i.**
4. **Output sum.**

1

IS INVARIANT TRUE RIGHT BEFORE THE LOOP BEGINS?

We assert sum = 1 and i = 2 after step 2 in the algorithm.

$$sum = \sum_{z=1}^{i-1} z \Rightarrow 1 = \sum_{z=1}^{2-1} z \Rightarrow 1 = \sum_{z=1}^{1} z \Rightarrow 1 = 1$$

# Loop Invariant Example

**Computing sum of 1+2+...+n, n > 0**

1. **Let sum = 1.**
2. **Let i = 2.**
3. **While i ≤ n do the following:**
   **a. Add i to sum.**
   **b. Add 1 to i.**
4. **Output sum.**

**2** FOR ANY ITERATION, IF INVARIANT IS TRUE BEFORE BODY BEGINS, IS IT TRUE AFTER BODY ENDS?

$$sum = \sum_{z=1}^{i-1} z$$

Add i to sum

$$sum = \sum_{z=1}^{i-1} z + i = \sum_{z=1}^{i} z$$

Add 1 to i

$$sum = \sum_{z=1}^{i-1} z$$



# Loop Invariant Example

**Computing sum of 1+2+...+n, n > 0**

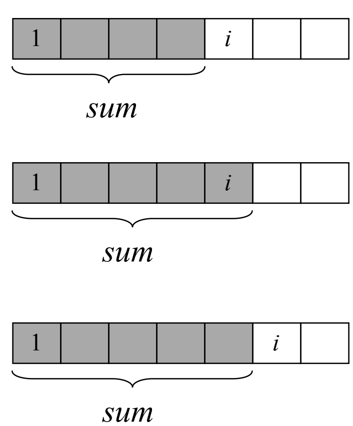
$$sum = \sum_{z=1}^{i-1} z$$

Add i to sum

$$sum = \sum_{z=1}^{i-1} z + i = \sum_{z=1}^{i} z$$

Add 1 to i

$$sum = \sum_{z=1}^{i-1} z$$



# Loop Invariant Example

**Computing sum of 1+2+...+n, n > 0**

1. **Let sum = 1.**
2. **Let i = 2.**
3. **While i ≤ n do the following:**
   a. **Add i to sum.**
   b. **Add 1 to i.**
4. **Output sum.**

3 — AFTER LOOP EXITS, INVARIANT IS TRUE AND EXIT CONDITION IS TRUE. DOES THIS IMPLY THE DESIRED COMPUTATION?

$$sum = \sum_{z=1}^{i-1} z \quad \text{AND} \quad \boxed{i = n+1} \quad \Rightarrow \quad sum = \sum_{z=1}^{(n+1)-1} z = \sum_{z=1}^{n} z$$

**EXIT CONDITION**



# Loop Invariant Example

**Computing sum of 1+2+...+n, n > 0**

1. **Let sum = 1.**
2. **Let i = 2.**
3. **While i ≤ n do the following:**
   a. **Add i to sum.**
   b. **Add 1 to i.**
4. **Output sum.**

4 — DOES THIS LOOP EVENTUALLY TERMINATE TO YIELD THE DESIRED COMPUTATION?

ARGUMENT:
i starts at 2. After each iteration, i increases by 1.
Since n is positive, i will eventually have to be greater than n, causing the loop to terminate. (Specifically, i = n+1 when the loop terminates.)

# Loop Invariant Example #2

**Computing $2^n$, n > 0**

1. Input n, an integer where n > 0.
2. Set i = 1.
3. Set f = 2.
4. While i ≠ n do the following:
    a. Add 1 to i.
    b. Multiply f by 2.
5. Output f.

Invariant: $f = 2^i$



# Loop Invariant Example #2

**Computing $2^n$, n > 0**

1 IS THE INVARIANT TRUE RIGHT BEFORE THE LOOP BEGINS?

# Loop Invariant Example #2

**Computing $2^n$, n > 0**

2 FOR ANY ITERATION, IF THE INVARIANT IS TRUE BEFORE BODY BEGINS, IS IT TRUE AFTER BODY ENDS?



# Loop Invariant Example #2

**Computing $2^n$, n > 0**

3 AFTER THE LOOP EXITS, THE INVARIANT IS TRUE AND THE EXIT CONDITION IS TRUE. DOES THIS IMPLY THE DESIRED COMPUTATION?

# Loop Invariant Example #2

**Computing $2^n$, n > 0**

4 DOES THIS LOOP EVENTUALLY TERMINATE?



# Verification using Induction

- Induction is used when there are an infinite number of conditions to check.
- Start with an assertion that you wish to prove is true about the computation.
  - Show inductive assertion is obviously true for a simple base case.
  - Then assuming assertion is true for one case, prove that the assertion is true for the next case.

# Induction Example

**Towers of Hanoi**

**Assertion: The number of moves required to move N discs is $2^N$-1.**

- Simple base case:

  When N=1, the number of moves is clearly 1 so the assertion is valid since for N=1, the formula for the number of moves gives $2^1 - 1 = 1$.



# Induction Example

**Towers of Hanoi (cont'd)**

- Inductive case:

  Using the assertion, <u>assume</u> the number of moves for N-1 discs is $2^{N-1} - 1$.

  Then using our algorithm for N discs,

  - We move N-1 discs to the extra peg
  - We move the largest disc
  - We move N-1 discs from the extra peg

  Total number of moves for N discs

  $= (2^{N-1} - 1) + 1 + (2^{N-1} - 1) = 2(2^{N-1}) - 1 = 2^N - 1.$

  This is our assumption, so if the formula works for N-1, it also works for N.

# Rationale behind Induction

- The base step says the assertion is true for N=1.
- The inductive step says that if the assertion is true for some N-1, it will also be true for N.
  - Since it's true for N=1, the inductive step says it's true for N=2.
  - Since it's true for N=2, the inductive step says it's true for N=3.
  - etc.
- Thus, we can show the assertion is true for all N in just two steps.



# Induction Example #2

- Using induction, prove that the sum of the integers from 1 to n is n(n+1)/2 for all n > 0.
- Simple Base Case:

# Induction Example #2

- Inductive Case:



# Quotations on Correctness

- "Testing proves a programmer's failure. Debugging is the programmer's vindication." *Boris Beizer*
- "Beware of bugs in the above code; I have only proved it correct, not tried it." *Donald Knuth*
- "Program testing can be used to show the presence of bugs, but never to show their absence." *Edsger Dijkstra*
- "It's not a bug. It's an undocumented feature." *Anonymous*