# LECTURE 15

# CORRECTNESS (CONTINUED)

# We are here

- Part 1: How computers works
  - Boolean logic, integers
  - Instructions
  - Memory
- Part 2: Software development
  - Compiling, make
  - ABIs & APIs
  - git
- Part 3: Correctness
  - Specifications
  - Documentation, testing
  - Static & dynamic analysis, debugging ⟵ TODAY
- Part 4: Performance
  - CPU pipelines, caches
  - Data structures
  - Parallel computation

# DEBUGGING TECHIQUES

# Instrumentation

- The basic approach to debugging is:

  Check that what we **think is true** is **actually true**.

- Narrow down the precise point at which execution deviates from our assumptions
- We can use
  - assertions: `assert` / `assert()`
  - debugging messages: `print()` / `printf()`
  - machine-readable output

# Crash instrumentation example

```
void perform_actions(struct state *s)
{
    action_a(s);
    action_b(s);
    action_c(s);
    action_d(s);
    action_e(s);
}
```

```
void perform_actions(struct state *s)
{
    printf("Action A...\n");
    action_a(s);
    printf("Action B...\n");
    action_b(s);
    printf("Action C...\n");
    action_c(s);
    printf("Action D...\n");
    action_d(s);
    printf("Action E...\n");
    action_e(s);
    printf("Actions done.\n");
}
```

```
Action A...
Action B...
Action C...
Segmentation fault
```

→ crash in `action_c()` assuming no time-traveling UB.

# Machine-readable output example

```
def matrix_inverse(mtx):
    ...
    return result
```

```
def matrix_inverse(mtx):
    ...

    error_matrix = mtx * result - matrix_identity()
    matrix_write(mtx, "mtx.m")
    matrix_write(result, "result.m")
    assert matrix_norm(error_matrix) < 1e-5

    return result
```

# How to handle large test cases?

- assume our `matrix_inverse()` code has a bug
- we find a wrong result for a specific 2000x2000 matrix
- how do we proceed?

- we would like to instrument `matrix_inverse()` by printing the matrix at each step,
- but a 2000x2000 matrix is too large to visualize

# Testcase reduction

- Input: $A \in \mathbb{R}^{n \times n}$
- Step 1: construct $B \in \mathbb{R}^{m \times m}$ by selecting an arbitrary square submatrix of $A$
- Step 2: test `matrix_inverse()` on $B$
- Step 3: if `matrix_inverse(`$B$`)` fails again, then $A := B$
- Step 4: go back to Step 1

Example approach:

- at first we can try removing a random half of the rows and columns of $A$
- if it fails repeatedly, we try to remove fewer rows and columns of $A$
- if it fails again, we remove a single row and column of $A$

This process can be automated!

# Code bisection

```c
void perform_actions(struct state *s)
{
    action_000(s);
    action_001(s);
    action_002(s);
    . . .
    action_998(s);
    action_999(s);
}
```

```c
void perform_actions(struct state *s)
{
    printf("First action...\n");
    action_000(s);
    action_001(s);
    action_002(s);
    . . .
    printf("Action 500...\n");
    action_500(s);
    . . .
    action_998(s);
    action_999(s);
    printf("Actions done.\n");
}
```

```
First action...
Action 500...
Segmentation fault
```

→ crash between 500 and 999 (assuming no time-traveling UB).

```c
void perform_actions(struct state *s)
{
    printf("First action...\n");
    action_000(s);
    . . .
    printf("Action 500...\n");
    action_500(s);
    . . .
    printf("Action 750...\n");
    action_750(s);
    . . .
    action_999(s);
    printf("Actions done.\n");
}
```

```
First action...
Action 500...
Action 750...
Segmentation fault
```

$\longrightarrow$ crash between 750 and 999.

```c
void perform_actions(struct state *s)
{
    printf("First action...\n");
    action_000(s);
    . . .
    printf("Action 500...\n");
    action_500(s);
    . . .
    printf("Action 750...\n");
    action_750(s);
    . . .
    printf("Action 875...\n");
    action_875(s);
    . . .
    action_999(s);
    printf("Actions done.\n");
}
```

```
First action...
Action 500...
Action 750...
Segmentation fault
```

$\longrightarrow$ crash between 750 and 875.

```c
void perform_actions(struct state *s)
{
    printf("First action...\n");
    action_000(s);
    . . .
    printf("Action 500...\n");
    action_500(s);
    . . .
    printf("Action 750...\n");
    action_750(s);
    . . .
    printf("Action 812...\n");
    action_812(s);
    . . .
    printf("Action 875...\n");
    action_875(s);
    . . .
    action_999(s);
    printf("Actions done.\n");
}
```

```
First action...
Action 500...
Action 750...
Action 812...
Segmentation fault
```

$\longrightarrow$ crash between 812 and 875.

# Version bisection

```
git log --oneline
9e9e6fc (HEAD -> main, origin/main) Added perf version check.
ff3c21b Changed branch mispredict ratio displayed.
fd49f78 Silently ignore branch events.
85afe03 Support new perf-script brstack format with added spaces.
77f8759 Made perf script output parsing more lenient.
637f374 Version bump.
47b578b Fixed erroneous use of atime, should have been mtime.
1dadd0f Moved objdump cache to /tmp.
60a534a Added caching of objdump output.
6f3c377 Some debugging code.
b2daa9b Updated version.
```

# Version bisection

```
git log --oneline
9e9e6fc (HEAD -> main, origin/main) Added perf version check. ← test this
ff3c21b Changed branch mispredict ratio displayed.
fd49f78 Silently ignore branch events.
85afe03 Support new perf-script brstack format with added spaces.
77f8759 Made perf script output parsing more lenient.
637f374 Version bump. ← test this
47b578b Fixed erroneous use of atime, should have been mtime.
1dadd0f Moved objdump cache to /tmp.
60a534a Added caching of objdump output.
6f3c377 Some debugging code.
b2daa9b Updated version. ← test this
```

# DEBUGGERS

- A debugger is a tool that allows us to run our code step-by-step (e.g. line by line)
- Between each step, we can examine
  - program output
  - program state (i.e. variables)
- Debuggers for interpreted languages are language-specific
- Debuggers for compiled languages work at the assembly level