

Tools for correctness (1)

We are here

- Part 1: How computers works

- Boolean logic, integers
- Instructions
- Memory

- Part 2: Software development

- Compiling (clang, make, ...)
- Code management (regex, git)

- Part 3: Correctness

- ABIs, Specifications
- Static & dynamic analysis, debugging

↑ TODAY

- Part 4: Performance

- CPU pipelines, caches
- Parallel computation

Assertions

Assertions

- Used to document (and check) assumptions made in the code.
- An assertion failure
 - should correspond to a **bug** in your code,
 - in Python, raises `AssertionError` exception
 - in C, triggers an immediate crash (`abort()`) of your program.

Example: the bisection method

Given an alphabetically-ordered list of n strings

```
acre  
airlock  
embassy  
helicopter  
iron  
log  
olive  
puddle  
skeleton  
trouble  
virus  
whey  
zoology
```

determine if the string

```
pen
```

is part of the list.

Checking every string one by one until a string is alphabetically after pen: $O(n)$

Bisection: $O(\log(n))$ expected complexity

```
acre  
airlock  
embassy  
helicopter  
iron  
log  
olive      <-- midpoint, before pen  
puddle  
skeleton  
trouble  
virus  
whey  
zoology
```

```
acre  
airlock  
embassy  
helicopter  
iron  
log  
olive  
puddle  
skeleton  
trouble    <-- midpoint, after pen  
virus  
whey  
zoology
```

acre
airlock
embassy
helicopter
iron
log
olive
puddle
skeleton
trouble
virus
whey
zoology

<-- midpoint, after pen

acre
airlock
embassy
helicopter
iron
log
olive
puddle
skeleton
trouble
virus
whey
zoology

_____ (pen not found)

Bisection implementation

```
def binary_search(sorted_list, target):
```

```
    lo = 0
    hi = len(sorted_list) - 1
```

```
    while lo <= hi:
```

```
        mid = (lo + hi) // 2
```

```
        if sorted_list[mid] == target:
            return True
```

```
        elif sorted_list[mid] < target:
            lo = mid + 1
```

```
        else:
            hi = mid - 1
```

```
    return False
```

Bisection implementation with assertions

```
def binary_search(sorted_list, target):  
    assert sorted_list == sorted(sorted_list)  
  
    lo = 0  
    hi = len(sorted_list) - 1  
  
    while lo <= hi:  
  
        mid = (lo + hi) // 2  
  
        if sorted_list[mid] == target:  
            return True  
        elif sorted_list[mid] < target:  
            lo = mid + 1  
        else:  
            hi = mid - 1  
  
    return False
```

Bisection implementation with assertions

```
def binary_search(sorted_list, target):  
    assert sorted_list == sorted(sorted_list)  
  
    lo = 0  
    hi = len(sorted_list) - 1  
  
    while lo <= hi:  
        assert sorted_list[lo] <= target  
        assert sorted_list[hi] >= target  
  
        mid = (lo + hi) // 2  
  
        if sorted_list[mid] == target:  
            return True  
        elif sorted_list[mid] < target:  
            lo = mid + 1  
        else:  
            hi = mid - 1  
  
    return False
```

Bisection implementation with correct assertions

```
def binary_search(sorted_list, target):
    assert sorted_list == sorted(sorted_list)

    if len(sorted_list) < 1 or target < sorted_list[0] or target > sorted_list[-1]:
        return False

    lo = 0
    hi = len(sorted_list) - 1

    while lo <= hi:
        assert sorted_list[lo] <= target
        assert sorted_list[hi] >= target

        mid = (lo + hi) // 2

        if sorted_list[mid] == target:
            return True
        elif sorted_list[mid] < target:
            lo = mid + 1
        else:
            hi = mid - 1

    return False
```

Disabling assertions

In Python:

```
python -O script.py
```

In C:

```
clang -D NDEBUG -Wall -O3 -o main main.c
```

(equivalent to

```
#define NDEBUG
```

at the beginning of every file)

Error vs assertion failure

- an error happens when, for external reasons, your program cannot run
 - examples: out of memory, file cannot be read, network unreachable
- an assertion fails if a fundamental assumption in your code is violated
 - indicates a **bug** in your code

Testing

```
/*  
    This functions returns:  
    0          if both of its arguments are zero  
    nonzero   if one or both of its arguments are nonzero  
*/  
int either_nonzero(int a, int b)  
{  
    if (a != 0)  
        a = 1;  
  
    if (b != 0)  
        b = 1;  
  
    return (a | b) == 1;  
}
```

```
void run_tests_0()  
{  
    assert(either_nonzero(5, 5) != 0);  
    assert(either_nonzero(0, 5) != 0);  
    printf("OK\n");  
}
```

Test coverage

- line coverage:
is every line of code covered by some test case?
- branch coverage:
for every conditional branch, is there a test covering each of the two possibilities
(taking the branch or not taking it)?
- path coverage:
is there a test covering all possible execution paths?

```
gcc -Wall -O3 --coverage -c -o either_nonzero.o either_nonzero.c
gcc -Wall -O3 --coverage -o run_tests main.c either_nonzero.o
```

```
./run_tests
```

OK

```
gcov either_nonzero.c
```

```
File 'either_nonzero.c'
Lines executed:100.00% of 4
Creating 'either_nonzero.c.gcov'

Lines executed:100.00% of 4
```

```
gcov -b either_nonzero.c
```

```
File 'either_nonzero.c'
Lines executed:100.00% of 4
Branches executed:100.00% of 4
Taken at least once:75.00% of 4
No calls
Creating 'either_nonzero.c.gcov'

Lines executed:100.00% of 4
```

```
cat either_nonzero.c.gcov
```

```
function either_nonzero called 2 returned 100% blocks executed 100%
    2:    4:int either_nonzero(int a, int b)
    -:    5:{
    2:    6:    if (a != 0)
branch 0 taken 50% (fallthrough)
branch 1 taken 50%
    -:    7:        a = 1;
    -:    8:
    2:    9:    if (b != 0)
branch 0 taken 100% (fallthrough)
branch 1 taken 0%
    -:   10:        b = 1;
    -:   11:
    2:   12:    return (a | b) == 1;
    -:   13:}
```

```

/*
  This functions returns:
    0      if both of its arguments are zero
  nonzero if one or both of its arguments are nonzero
*/
int either_nonzero(int a, int b)
{
    if (a != 0)
        a = 1;

    if (b != 0)
        b = 1;

    return (a | b) == 1;
}

```

```

void run_tests_0()
{
    assert(either_nonzero(5, 5) != 0);
    assert(either_nonzero(0, 5) != 0);
    printf("OK\n");
}

```

Line coverage vs. branch coverage

```
/*  
  This functions returns:  
  0      if both of its arguments are zero  
  nonzero if one or both of its arguments are nonzero  
*/  
int either_nonzero(int a, int b)  
{  
    if (a != 0)  
        a = 1;  
  
    if (b != 0)  
        b = 1;  
  
    return (a | b) == 1;  
}
```

```
void run_tests_x()  
{  
    assert(either_nonzero(5, 5) != 0);  
    printf("OK\n");  
}
```

Line coverage: 100%

Branch coverage: 50%

Branch coverage vs. path coverage

```
/*  
    This functions returns:  
    0          if both of its arguments are zero  
    nonzero    if one or both of its arguments are nonzero  
*/  
int either_nonzero(int a, int b)  
{  
    if (a != 0)  
        a = 1;  
  
    if (b != 0)  
        b = 1;  
  
    return (a | b) == 1;  
}
```

```
void run_tests_y()  
{  
    assert(either_nonzero(0, 0) == 0);  
    assert(either_nonzero(0, 5) != 0);  
    assert(either_nonzero(5, 0) != 0);  
    printf("OK\n");  
}
```

Line coverage: 100%

Branch coverage: 100%

Path coverage: 75%

How does it work?

```
gcc -Wall -O3 --coverage -c -o either_nonzero.o either_nonzero.c
```

```
/*  
  This functions returns:  
    0      if both of its arguments are zero  
  nonzero if one or both of its arguments are nonzero  
*/  
int either_nonzero(int a, int b)  
{  
    line_covered(6);  
    if (a != 0) {           // line 6  
        branch_covered(6, 1);  
        line_covered(7);  
        a = 1;             // line 7  
    } else {  
        branch_covered(6, 0);  
    }  
  
    line_covered(9);  
    if (b != 0) {           // line 9  
        branch_covered(9, 1);  
        line_covered(10);  
        b = 1;             // line 10  
    } else {  
        branch_covered(9, 0);  
    }  
  
    line_covered(12);  
    return (a | b) == 1;    // line 12  
}
```

Limitations of test coverage measures (1)

```
/*  
  This functions returns:  
    0      if both of its arguments are zero  
  nonzero if one or both of its arguments are nonzero  
*/  
int either_nonzero_WRONG_1(int a, int b)  
{  
    if (a != 0)  
        a = 1;  
  
    if (b != 0)  
        b = 1;  
  
    return (a + b) == 1;  
}
```

```
void run_tests_1()  
{  
    assert(either_nonzero_WRONG_1(0, 0) == 0);  
    assert(either_nonzero_WRONG_1(0, 5) != 0);  
    assert(either_nonzero_WRONG_1(5, 0) != 0);  
    //assert(either_nonzero_WRONG_1(5, 5) != 0); // <-- this one fails  
    printf("OK\n");  
}
```

Line coverage: 100%

Branch coverage: 100%

Path coverage: 75%

Limitations of test coverage measures (2)

```
/*  
  This functions returns:  
    0      if both of its arguments are zero  
  nonzero if one or both of its arguments are nonzero  
*/  
int either_nonzero_WRONG_2(int a, int b)  
{  
    return a + b;  
}
```

```
void run_tests_2()  
{  
    assert(either_nonzero_WRONG_2(0, 0) == 0);  
    assert(either_nonzero_WRONG_2(0, 5) != 0);  
    assert(either_nonzero_WRONG_2(5, 0) != 0);  
    assert(either_nonzero_WRONG_2(5, 5) != 0);  
    //assert(either_nonzero_WRONG_2(5, -5) != 0); // <-- this one fails  
    printf("OK\n");  
}
```

Line coverage: 100%

Branch coverage: 100%

Path coverage: 100%

Fuzzing

We need good tests

Assertions and tests are useful

but only if we have good test cases

and enough of them

⇒ How do we generate good tests?

On a basic level, a fuzzer proceeds as follows:

1. take a (mostly valid) example input file
2. run the tested program with that input file
3. check for crashes (e.g. segmentation fault, assertion failures)
4. modify the input file:
 - random modifications
 - truncations, duplications
 - mergers with other example input files
5. go back to 2

Advanced fuzzers

- use test coverage techniques
to determine which input files are “interesting”,
in that they cover previously-uncovered source code
- use static analysis techniques
to determine input file modifications that could trigger specific code branches

- open source project (<https://aflplus.plus/>)
- takes as an input a directory with many (mostly valid) example input files
- generates modified input files that (try to) yield crashes

```
afl-fuzz -i directory/with/example/inputs/ -o directory/for/crash/files/ -- ./executable @@
```

```

american fuzzy lop ++4.09a {default} (../examples/glpsol) [fast]
┌── process timing ────────────────────────────────────────────────────┐
│   run time : 0 days, 0 hrs, 0 min, 17 sec                          │
│   last new find : 0 days, 0 hrs, 0 min, 6 sec                      │
│ last saved crash : none seen yet                                    │
│ last saved hang : none seen yet                                    │
└── cycle progress ────────────────────────────────────────────────────┘
    now processing : 54.22 (58.1%)
    runs timed out : 0 (0.00%)
┌── stage progress ────────────────────────────────────────────────────┐
│ now trying : splice 5                                              │
│ stage execs : 56/57 (98.25%)                                       │
│ total execs : 73.4k                                                │
│ exec speed : 4212/sec                                              │
└── fuzzing strategy yields ───────────────────────────────────────────┘
    bit flips : disabled (default, enable with -D)
    byte flips : disabled (default, enable with -D)
    arithmetics : disabled (default, enable with -D)
    known ints : disabled (default, enable with -D)
    dictionary : n/a
    havoc/splice : 29/49.2k, 8/23.5k
    py/custom/rq : unused, unused, unused, unused
    trim/eff : 91.92%/96, disabled
┌── strategy: explore ───────────────────────────────────────────────────┐
│ state: started :-) ───────────────────────────────────────────────────┘
┌── overall results ────────────────────────────────────────────────────┐
│ cycles done : 2                                                    │
│ corpus count : 93                                                  │
│ saved crashes : 0                                                  │
│ saved hangs : 0                                                    │
└── map coverage ────────────────────────────────────────────────────┘
    map density : 0.25% / 0.28%
    count coverage : 3.94 bits/tuple
┌── findings in depth ───────────────────────────────────────────────────┐
│ favored items : 6 (6.45%)                                          │
│ new edges on : 8 (8.60%)                                          │
│ total crashes : 0 (0 saved)                                       │
│ total tmouts : 0 (0 saved)                                       │
└── item geometry ────────────────────────────────────────────────────┘
    levels : 4
    pending : 28
    pend fav : 0
    own finds : 37
    imported : 0
    stability : 100.00%
[cpu000: 8%]

```