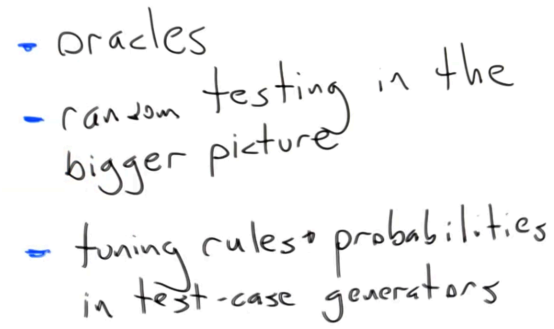
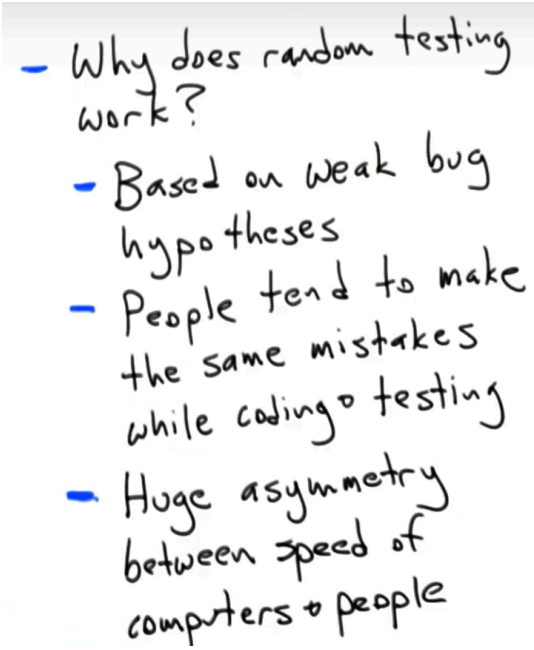
Random testing in practice

https://youtube.com/playlist?list=PLAwxTw4SYaPkWVHeC_8aSIbSxE_NXI76g&si=lhs0mKJKm5pW3D1R

1. Introduction



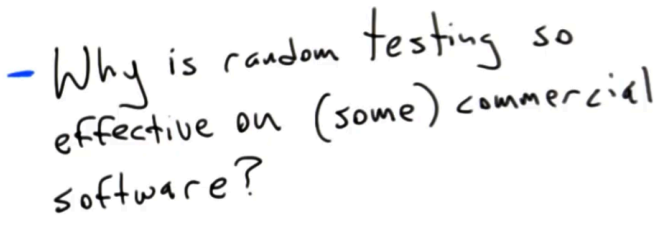
2. Random testing in the bigger picture



•We don't have to guess about some particular thing that might fail, rather about a whole class of things that might possibly fail. This turns out to be powerful because, given the very complex behavior of modern software, people don't seem to be very good about forming good hypotheses about where bugs lie.

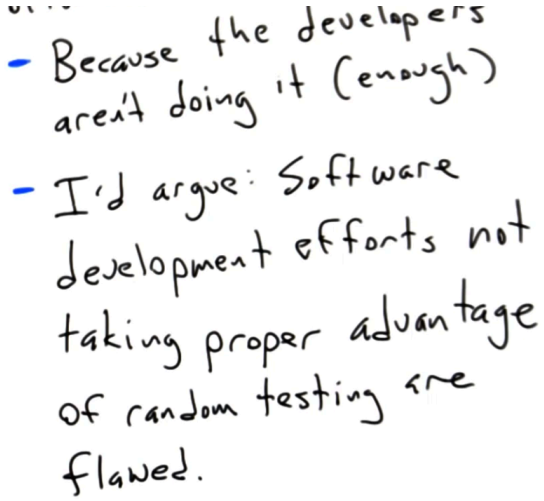
•If we forgot to implement some feature or if we miss-implemented a feature, we won't test the things done wrong. Random testing to some extent get us out of this problem because it can construct test cases to test things that we don't actually understand or know very well.

- The random tester is mostly generating stupid test cases, but if it can generate a clever test case, say one in a million times, then that still might be a more effective use of our testing resources than writing test cases by hand.



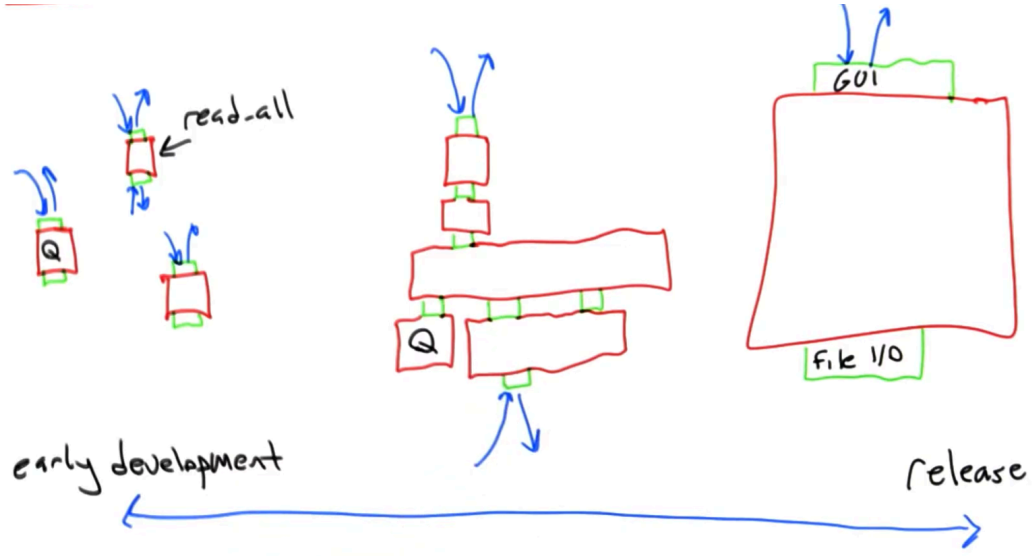
•There's pretty ample evidence, for example the fuzzing papers that we discussed or the Charlie Miller's talk that you can watch online, Babysitting an army of monkeys.

3. How random testing should work



•Modern software systems are so large and so complicated that test cases produced by non-random processes are simply unlikely to find the kind of bugs that are lurking in these software systems.

- A rough software development timeline with a releasing software and early development stages:

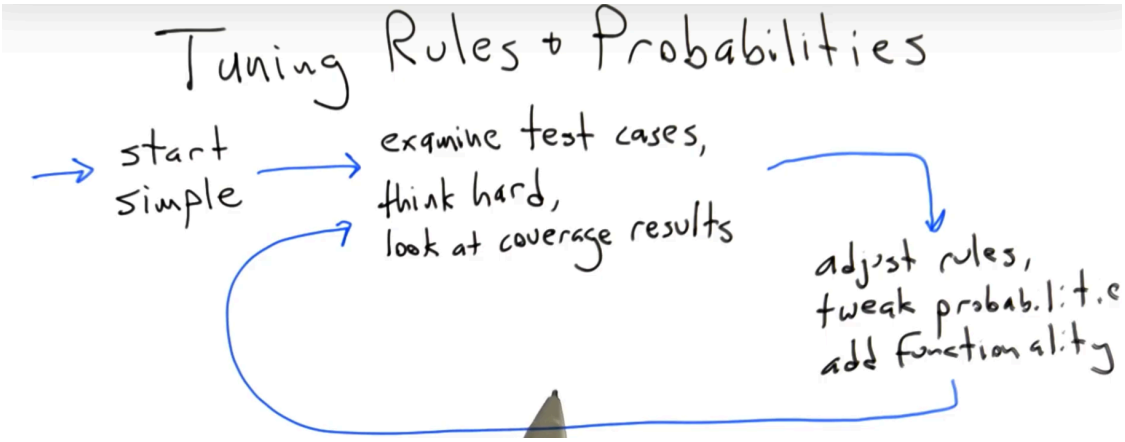


- We want to ensure as we're developing the modules that we're creating robust pieces of software whose interfaces we understand and that will be solid foundation for future work.
- It's going to be the case that some of our random testers become useless (e.g. the bounded queue) but others (e.g. those that come in at the top level and those that perform system-level fault injection) may still be useful.
- Our focus should be on external interfaces provided, things like file I/O and the graphical user interface, and so we're fuzzing exactly these sorts of things.

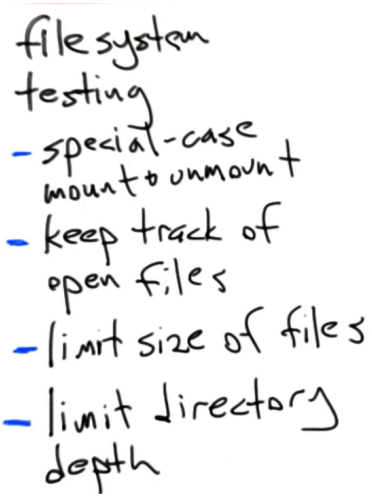
4. Random system testers

- Our software evolves to be more robust as we move toward releasing if we're evolving our random tester to be stronger and stronger.
- At some point we may have a feature where we generate a new kind of random input that we haven't generated before. Also it's going to generate some bugs report, and we'll fix them, and our software evolves to be more robust. If we keep doing this for weeks or years, we'll end up with a random tester and a system that have gone through this co-evolution process where they both become much stronger, i.e. we've evolved an extremely sophisticated random tester, and a robust system with respect to the kind of faults that can be triggered by that random tester.

5. Tuning rules and probabilities



6. Filesystem testing



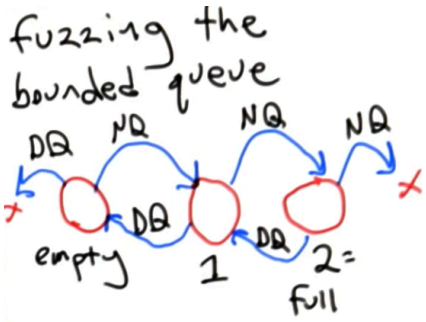
•If we start with a simple file system tester, we'll make a list of all the API calls that we'd like to test then call them randomly with random arguments.

•After observing that this doesn't work very well we consider special test cases in order to remove limitations of our random tester and, over time, this process ends up with a random tester that will be extremely strong.

7. Fuzzing the bounded queue

- We already wrote a random tester for the bounded queue data structure. Did that further do a good job at all? We found all the type of the bugs seeded so the answer is yes.

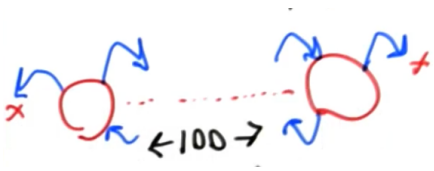
- Let’s look at the queue as a FSM (FSM stands for finite state machine). It contains 3 nodes.



- NQ stands for enqueue operation, DQ stands for dequeue operation.
- We start off with an empty queue and 50% of the time, at this point, we're going to make a dequeue call, which is going to fail, 50% of the time, we're going to enqueue something resulting in a queue containing one element and so on.
- The dynamic process that we'll get when we run a random tester is a random walk through this FSM. **Does this random walk have a reasonable probability of executing all the cases?** The most interesting cases are dequeuing from an empty queue, enqueuing to a full queue, and then walking around the rest of the states.

8. First attempt

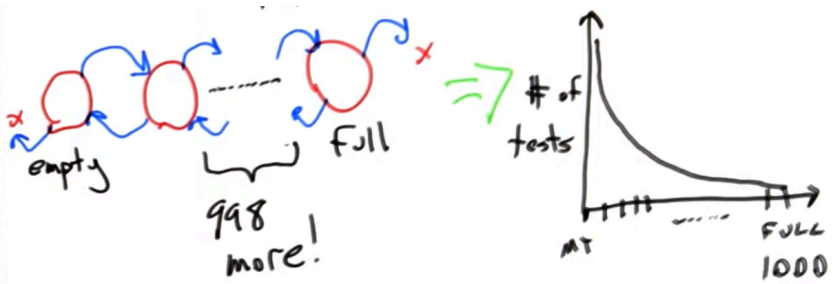
- **We go back to the random tester presented in the Random testing lecture at section 26.** We modify the code to make a 2 elements queue and then run the random tester. We get a crude coverage metric. Similar information we get using a branch coverage tool.
- Let's say that we had a queue containing 100 elements. **Are we going to get good behavioral coverage of our queue in this case?**



•Run the tester.

9. Tuning probabilities in practice

- Let’s visualize the execution of the queue that stores a 1000 elements. The FSM contains 1001 nodes.



- When we run the random tester, this time we've done around 50,000 adds to a non-full queue, and we haven't done any adds to a full queue. We've done almost 50,000 removes from a non-empty queue and 10 removes from an empty queue (numbers may vary for each execution).

- So, we'll get to a situation where the probability of visiting the states farther away drops off exponentially. **What do we have to do differently to a random tester to make sure to test this situation?** There is no certain answer.
- We might have to adjust the probabilities to compensate.
- One possible solution would be to bias the probabilities towards enqueueing, e.g. unbalance them using a 60-40 distribution. This time there are cases not tested. Or:

```
def random_test():
    N = 4
    add = 0
    remove = 0
    addFull = 0
    removeEmpty = 0
    q = Queue(N)
    q.checkRep()
    l = []
    for x in range(20):
        bias = 0.2
        if x%2 == 1:
            bias = -0.2
        for i in range(100000):
            if random.random() < (0.5 + bias):
                z = random.randint(0, 1000000)
                res = q.enqueue(z)
                q.checkRep()
                if res:
                    l.append(z)
                    add += 1
                else:
                    assert len(l) == N
                    assert q.full()
                    q.checkRep()
                    addFull += 1
            else:
                dequeued = q.dequeue()
                q.checkRep()
                if dequeued is None:
                    assert len(l) == 0
                    assert q.empty()
                    q.checkRep()
                    removeEmpty += 1
                else:
                    expected_value = l.pop(0)
                    assert dequeued == expected_value
                    remove += 1
        while True:
            res = q.dequeue()
            q.checkRep()
            if res is None:
                break
            z = l.pop(0)
            assert z == res
        assert len(l) == 0
```

```
    print("adds: " + str(add))
    print("adds to a full queue: " + str(addFull))
    print("removes: " + str(remove))
    print("removes from an empty queue: " + str(removeEmpty))

random_test()
```

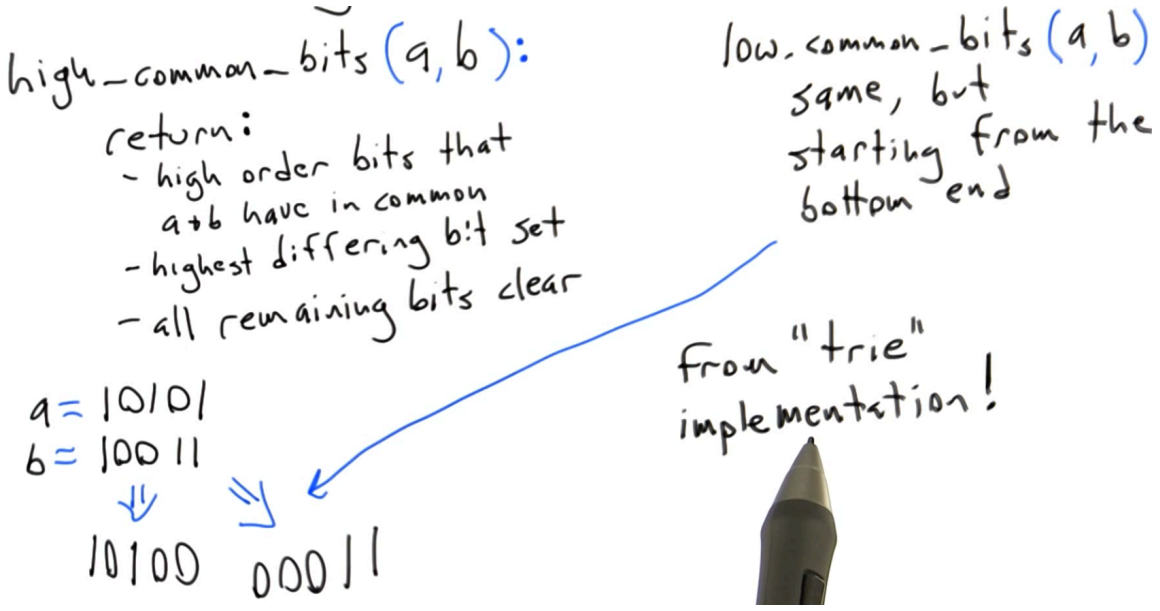
- We took a random testing loop and we **enclose it in a larger random testing loop** and in that larger loop, we made qualitative change to one of the probabilities, i.e. we bias execution towards one of our API calls in favor of the other and on even-numbered calls, we biased it the other way.

10. Stressing the whole system

- In the file system example if we call mount and unmount with equal probability as the rest of the calls there are too many times. But if we hardcode a call to those at the beginning and the end of our entire random testing run, there are too few.
- What we can do is mount the file system, executing a bunch of API calls, unmount it, and then **enclose that in an outer testing loop to make sure that we stress all the parts of the system**.
- The state machine for the file system case is considerably more complicated than the state machine for the queue, but on the other hand we find that we need to adjust our probabilities.
- Let's see another example.

11. Testing bitwise functions

- We want to write 2 Python functions with 2 inputs, a and b which are integers that have the same size, let's say 32 or 64-bit integers:



- The functions come from an optimized trie implementation. A **trie** is a kind of a balanced ordered tree, not very different than the splay tree, and relies on bitwise operations to find the descendants of nodes in order to get really high performance and really low-memory footprints.

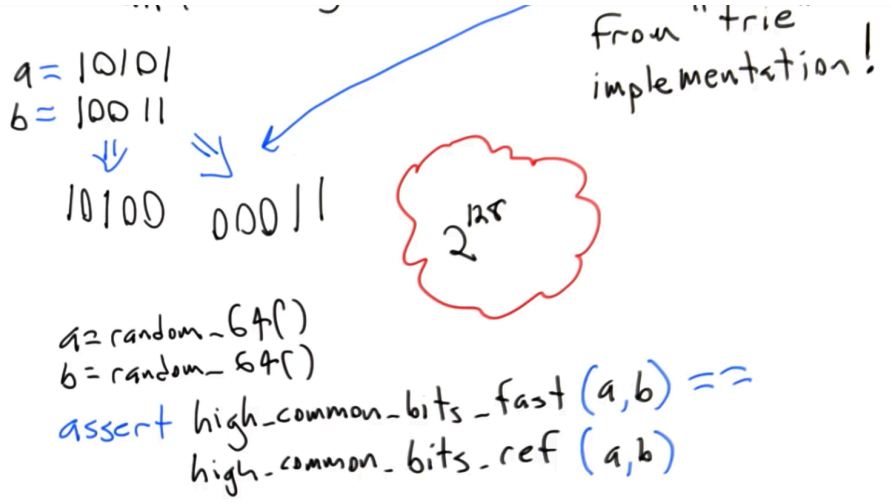
12. Bitwise implementation

- Let's assume we have 64-bit inputs. A really slow implementation:

```
def high_common_bits(a, b):
      mask = 0x8000000000000000
      output = 0
      for i in reversed(range(64)):
            if (a & mask) == (b & mask):
                  output |= a & mask
            else:
                  output |= mask
                  return output
            mask >>= 1
      return output

def low_common_bits(a, b):
      mask = 1
      output = 0
      for i in range(64):
            if (a & mask) == (b & mask):
                  output |= a & mask
            else:
                  output |= mask
                  return output
            mask <<= 1
      return output

def test(a, b):
      print("a= " + str(a) + " b= " + str(b))
      print(high_common_bits(a, b))
      print(low_common_bits(a, b))

test(5584184435867171854, 754275839179325636)
```

- Python's |= operator when applied to two Boolean values a and b performs the logical OR operation.
- Optimized implementations might execute in just a handful of clock cycles as opposed to maybe a hundred clock cycles for the optimized C versions and probably thousands of clock cycles for the Python versions.
- Performances are our concern, but first we need to check if the implementations are correct or not.

- We will write a random tester because the input domain of these functions is going to contain 2^128 elements so it's far too big to test. We are forced to do systematic testing or random testing.
- In a random 64-bit integer we assert that our super super optimized high_common_bits_fast function returned the same results as a reference implementation (Python or C version):



- Is this a good random tester? Let's run the code coverage tool.

13. Coverage for random testing

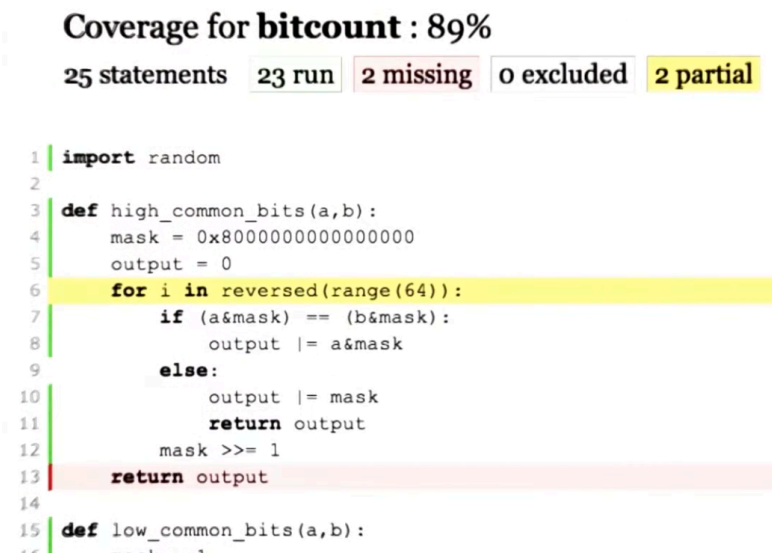
- Since all we have is the reference implementation, we will not test 2 implementations against each other, rather we will make up 2 random 64-bit integers and just run the high_common_bits and low_common_bits function on them.

```
for i in range(100000):
      a = random.getrandbits(64)
      b = random.getrandbits(64)
      print(high_common_bits(a, b) + low_common_bits(a, b))
```

- Run the code coverage tool:

```
coverage erase ; coverage run --branch main.py ; coverage html -i
```

- The report:



- Why completely random testing with valid inputs 100 000 times did not manage to cover this? What are the odds of 2 randomly generated 64-bit integers being the same? They're extremely low compared to a number of test cases running.
- When we do optimized implementations of these functions, we're going to use specialized instructions that modern architecture support with providing bit counts.
- We consider GCC as compiler. See the documentation at https://gcc.gnu.org/onlinedocs/gcc-4.4.7/gcc/Other-Builtins.html

— Built-in Function: int **__builtin_clz** (*unsigned int x*)

Returns the number of leading 0-bits in *x*, starting at the most significant bit position. If *x* is 0, the result is undefined.

— Built-in Function: int **__builtin_ctz** (*unsigned int x*)

Returns the number of trailing 0-bits in *x*, starting at the least significant bit position. If *x* is 0, the result is undefined.

— Built-in Function: int **__builtin_popcount** (*unsigned int x*)

Returns the number of 1-bits in *x*.

— Built-in Function: int **__builtin_parity** (*unsigned int x*)

Returns the parity of *x*, i.e. the number of 1-bits in *x* modulo 2.

— Built-in Function: int **__builtin_ffsl** (*unsigned long*)

Similar to `__builtin_ffs`, except the argument type is `unsigned long`.

— Built-in Function: int **__builtin_clzl** (*unsigned long*)

Similar to `__builtin_clz`, except the argument type is `unsigned long`.

We can use functions like clz and ctz to implement an extremely fast version of the high_common_bits and low_common_bits functions.

- Let's try to do better.

14. Improving the fuzzer

- A better random tester for these functions would set integer a completely random and then flipping a random number of bits, i.e. b is a mutated function of a.
- Number of total random tests is 10,000 instead of 100,000. We will flip bits using Python's XOR operator ^

```
for i in range(10000):
      a = random.getrandbits(64)
      b = a
      for j in range(random.randrange(63)):
            b ^= 1<<random.randrange(0, 63)
      print(high_common_bits(a, b) + low_common_bits(a, b))
```

- Run the code coverage tool.
- The report:

Coverage for **bitcount** : 100%

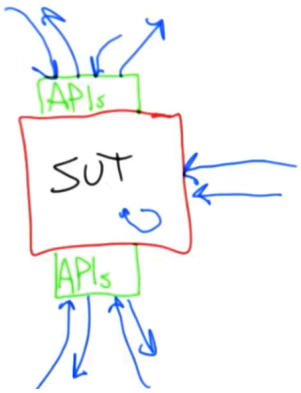
27 statements	27 run	0 missing	0 excluded	0 partial

15. Domain specific knowledge

- For the first random tester, where every bit was independently different, the probability of executing further into these functions dropped off exponentially, and so at 64-bits, the probability dropped off too fast for us to ever execute the case where a and b were the same.
- Based on some domain specific knowledge, we made a new random tester whereby flipping a random number of bits did a much more even exploration of the iteration spaces of those loops including reaching the ending state, which is what we actually wanted to test. This has been a pretty elaborate exercise for what in the end are 2 simple functions and that means is okay implementing a simple random tester and not understanding what's going on, but if we want to do a better job **we need to think about what we're testing, how the code is structured, and how we're going to execute all the way through it**, and this is a fundamental limitation of random testing.

16. Fuzzing implicit inputs

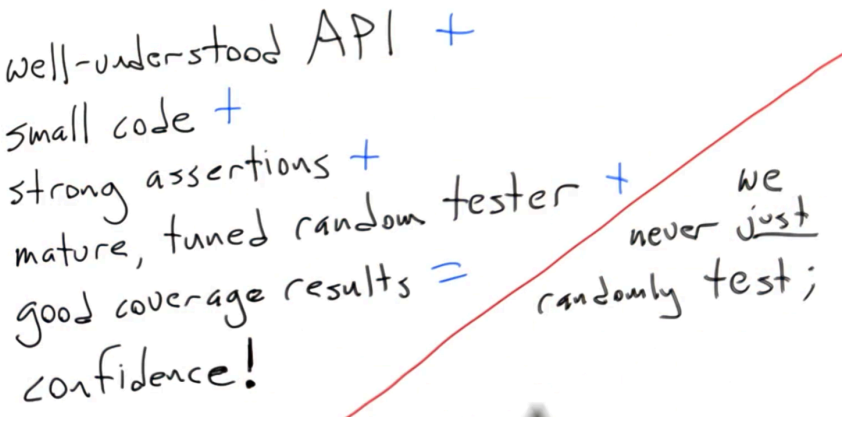
- Recall that the SUT provides APIs and most of the time that's what we're fuzzing. On the other hand, the SUT uses APIs. In the following we talk about fuzzing of the implicit inputs where we have non-API inputs that affect SUT's behavior. 3 important techniques:



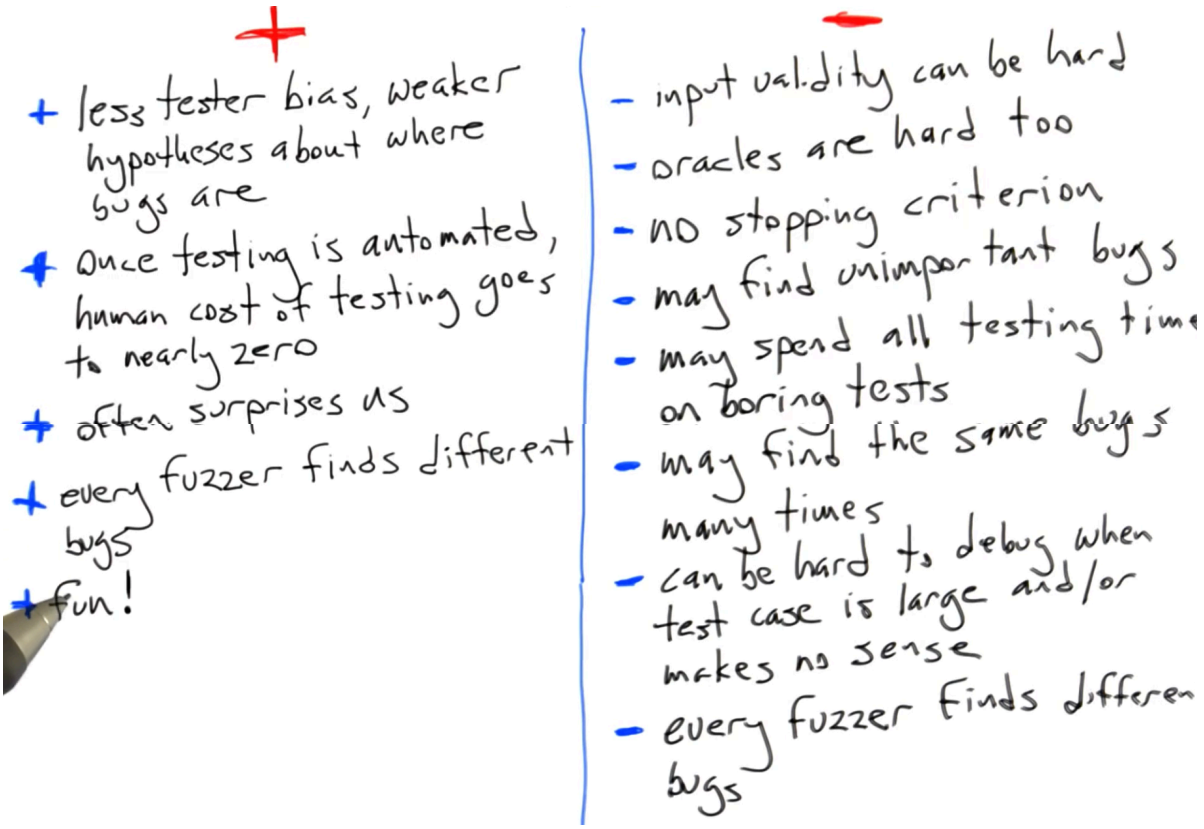
- perturbing the scheduler
  - generate load
  - network activity
  - thread stress testing tools
- inserting delays near synchronization & accesses to shared variables
- "unfriendly emulators"

17. Can random testing inspire confidence?

- **The purpose of testing is to maximize the number of bugs found per amount of effort spent testing.**



18-20. Tradeoffs in random testing



Notă[1]

[1] © Copyright 2022 Lect. dr. Sorina-Nicoleta PREDUT
Toate drepturile rezervate.