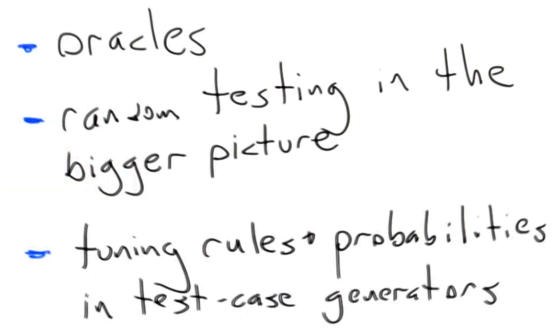
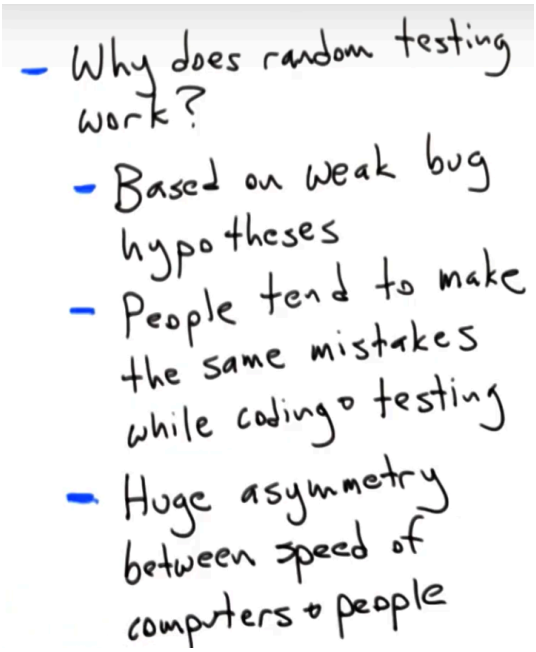
Random testing in practice

https://www.udacity.com/course/software-testing--cs258

1. Introduction



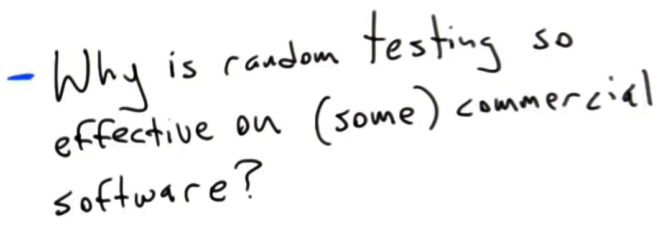
2. Random testing in the bigger picture



•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.

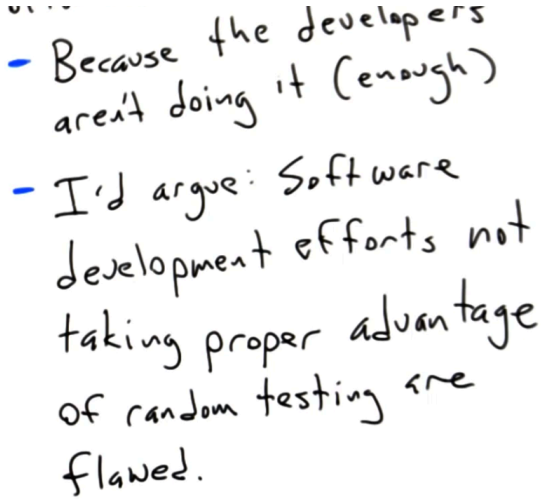
•If we forgot to implement some feature or if we miss-implemented on feature, we won't test the things done wrong. Random testing to some extent get us out of this problem because it can construct test cases to test things that we don't actually understand or know very well.

- 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.



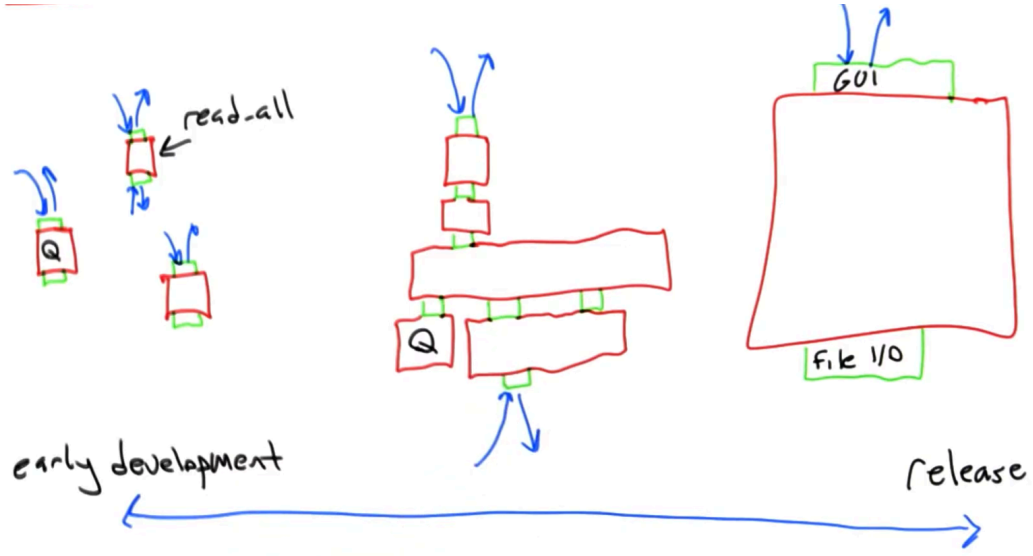
•There's pretty ample evidence, for example the fuzzing papers that we discussed or the Charlie Miller's talk that you can watch online, Babysitting an army of monkeys.

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:

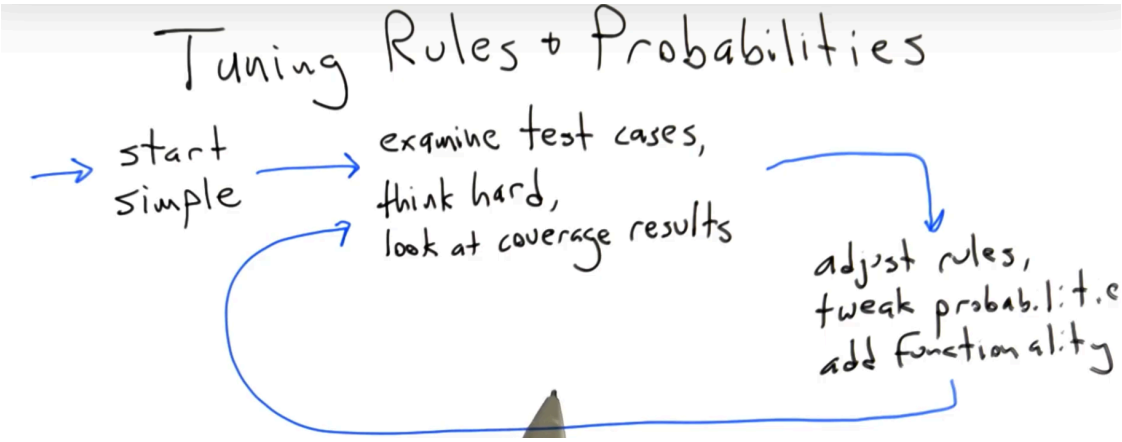


- We want to ensure as we're developing the modules that we're creating robust pieces of software whose interfaces we understand and that will be solid foundation for future work.
- It's going to be the case that some of our random testers become useless (e.g. the bounded queue) but others (e.g. those 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.

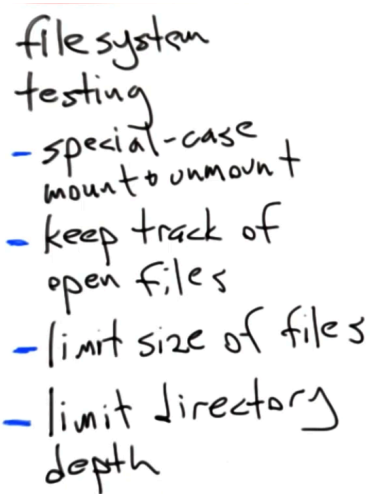
4. Random system testers

- 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 bug 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



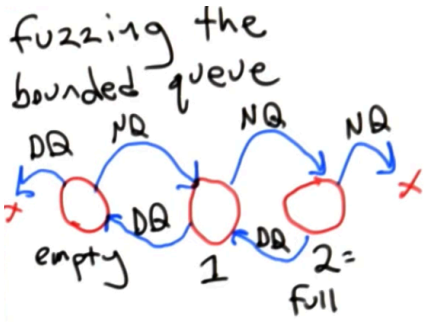
•If we start with a simple file system tester, we'll make a list of all the API calls that we'd like to test then call them randomly with random arguments.

•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.

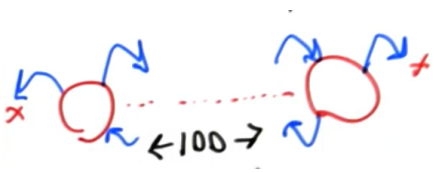
- Let's look at the queue as a FSM (FSM stands for finite state machine). It contains 3 nodes.



- 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 to 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

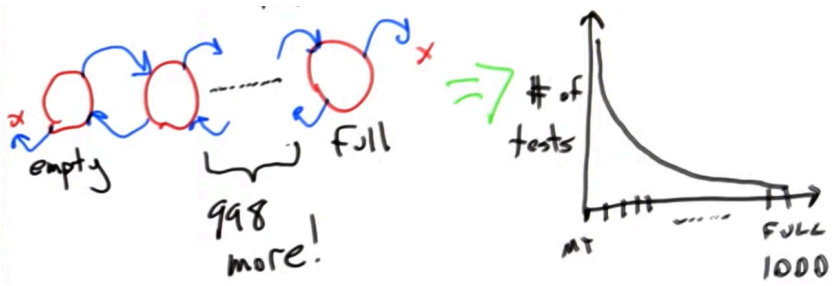
- **We go back to the random tester presented in the Random testing lecture at section 26.** We modify the code to make a 2 element queue and then run the random tester. We get a crude coverage metric. Similar information we get using a branch coverage tool.
- Let's say that we had a queue containing 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 common 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

-