Random testing

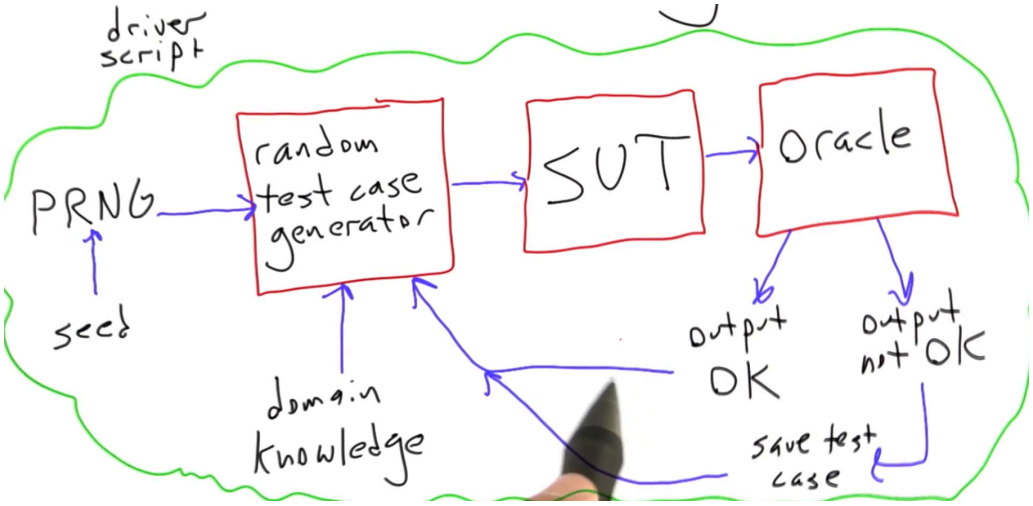
https://youtube.com/playlist?list=PLAwxTw4SYaPkWVHeC_8aSIbSxE_NXI76g&si=lhs0mKJKm5pW3D1R

1. Random testing

- Test cases are created using input from a random number generator.

  PRNG here stands for **pseudorandom number generator**.

  A seed completely determines the sequence of random numbers it's going to generate.
- Random testing diagram:



- SUT executes and produces some output. The output is inspected by a test oracle. The oracle makes a determination whether the output is either good or bad.
- If the output is good, i.e., if it passes whatever checks we have, we just go back and do it again.
- On the other hand, if the output is not okay, we save the test case somewhere for later inspection and we go back and do more random testing.
- **The key to making this all work is wrap the entire random testing tool chain into some sort of a driver script which runs it automatically.**
- And while we're doing other things, the random testing loop executes hundreds, thousands, or millions of times.
- The next time we want to see what's going on, we look at what kind of test cases have been saved. If anything interesting turned out, we have some followup work to do, like creating reportable test case and debugging. If nothing interesting happened, then that's good. We didn't introduce any new bugs and we can rebuild the latest version of a SUT and start the testing loop again.

- If the random test generator is well done, and if we have sufficient amount of CPU resources for the testing loop, and if it's not finding any problems, **random testing can significantly increase our confidence that the SUT is working as intended.** And it turns out that, in general there are only a couple of things that are hard about making this work.
- First of all, it can be tricky to come up with a good random test case generator, and second, it can be tricky to come up with good oracle.
  We've already said that these are **the hard things about testing in general, making test cases, and determining if outputs are correct.**

2. Testing compilers

- In the following is presented a tool created by a research group at School of Computing, University of Utah, called **Csmith** (https://github.com/csmith-project/csmith)
  The tool is a **random test case generator**, used to **test** (break) **C compilers**.
- The results for an in-progress Csmith run that's been going for a couple of days are:

```
Last login: Mon May 14 17:32:34 2012 from 23-24-209-141-static.hfc.comcastbusiness.net
[regehr@dyson ~]$ cd z/test
[regehr@dyson test]$ see_results
work0/output.txt:COMPILER FAILED current-gcc
work1/output.txt:CSMITH FAILED
work2/output.txt:CSMITH FAILED
work3/output.txt:COMPILER FAILED current-gcc
work3/output.txt:COMPILER FAILED clang
work3/output.txt:COMPILER FAILED current-gcc
work4/output.txt:COMPILER FAILED current-gcc
work4/output.txt:COMPILER FAILED current-gcc
work5/output.txt:CSMITH FAILED
work6/output.txt:COMPILER FAILED current-gcc
work7/output.txt:COMPILER FAILED clang
work7/output.txt:CSMITH FAILED
work7/output.txt:COMPILER FAILED clang
work7/output.txt:COMPILER FAILED current-gcc
tests: 150051
[regehr@dyson test]$ grep Assertion work*/output.txt
work7/output.txt:  clang: LazyValueInfo.cpp:605: bool <anonymous namespace>::LazyValueInfoCache::solveBlockVal
nLocal(<anonymous>::LVILatticeVal &, llvm::Value *, llvm::BasicBlock *): Assertion `isa<Argument>(Val) && "Unk
 live-in to the entry block"' failed.
work7/output.txt:  clang: LazyValueInfo.cpp:605: bool <anonymous namespace>::LazyValueInfoCache::solveBlockVal
nLocal(<anonymous>::LVILatticeVal &, llvm::Value *, llvm::BasicBlock *): Assertion `isa<Argument>(Val) && "Unk
 live-in to the entry block"' failed.
work7/output.txt:  clang: LazyValueInfo.cpp:605: bool <anonymous namespace>::LazyValueInfoCache::solveBlockVal
nLocal(<anonymous>::LVILatticeVal &, llvm::Value *, llvm::BasicBlock *): Assertion `isa<Argument>(Val) && "Unk
 live-in to the entry block"' failed.
work7/output.txt:  clang: ScheduleDAG.cpp:452: void llvm::ScheduleDAGTopologicalSort::InitDAGTopologicalSortin
 Assertion `Node2Index[SU->NodeNum] > Node2Index[I->getSUnit()->NodeNum] && "Wrong topological sorting"' faile
[regehr@dyson test]$
```

- They were making a program using Csmith and then using the latest version of GCC and the latest version of Clang that is LLVM C front-end to compile each test case with a variety of optimization levels.
- During this testing run GCC has failed a half dozen times or so, Clang has failed a few times, and also we see a few Csmith failures. It could be that the Csmith failures are

actual bugs but most of the time these are timeouts and so generally all of these tools are ran in their timeouts when we use a random testing loop because random test tend to be really good at finding performance pathologies where the tool runs for a really long time.

3. Random testing example

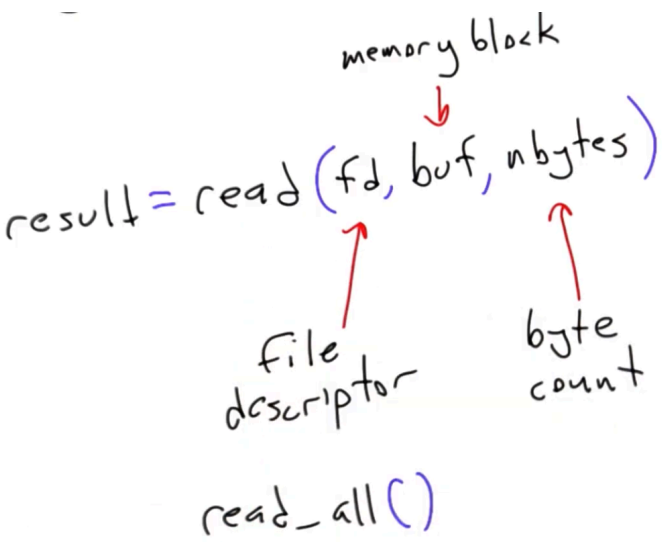
- We don't know what about that program (1600 lines or 37 kilobytes of code) was that caused Clang to crash. The stack trace is giving us a detailed version of the arguments and some other stuff. More about the analysed violated assertions and demo see the Udacity lesson 5.

4. Random testing loop

- We go back to random testing diagram and describe the process for Csmith tool.
- It was given seeds by a driver's script, the driver's script run the tools in a loop and the oracle in this case was just simply looking for compiler crashes, so it wasn't even running the compiler output.
- We weren't even running that code or even looking at it, all we're doing is waiting for the compiler to tell us that it failed, and the reason is because the **compiler developers** have conveniently **included lots of assertion violations**.
- So the driver's script is then checking the output, taking the test cases or in this case the seed in the log file and then go back and do it again.
- In about 2 days of testing time, this loop would've executed about 150,000 times on a fast day core machine, and of course, we were testing simpler systems and compilers.

5. Testing a Unix utility

- We're going to test the tiny UNIX utility function using a very tiny random tester.
- Remember a couple of units ago we talked about the **UNIX read system call**, so let's look how read system call works.
- This example is implemented in C language, because it's not ok in Python.
- Usually what the read system call does is it reads the number of bytes that you expected into the memory block that you provided



and all was good.

- The **return value** of read is going to be the **number of bytes read**, so that's what usually happens, but there a couple of other things that can happen.
- A 2nd possibility is the read system call then returns **0**, indicating that you've reached the end of the file.
- Another thing that can happen is it can return **-1** if it failed.
- The 4th possibility it can return the number of bytes **less than the number you asked for** and this isn't a failure, this just doesn't represent any sort of out of memory condition or end of file or anything like that, it just means we need to try again.

- And so the little UNIX utility function that we are going to test here is a different version of read, called **read_all** that acts just like read except that **it's not going to have this property of returning partial reads ever**. So, in the case of a partial read is going to just issue another read call that picks up where the first call left off and it's going to repeatedly do that until the read is complete or until some sort of an error or end of file condition occurs. If anything bad happens, it will just return a **-1** value.

6. Testing read all

- Left variable is the number of bytes left to read. Initially is the total number of bytes to read. The while loop is going to operate until either the read system call returns something less than 1, i.e. it returns 0 indicating an end of file condition or -1 indicating an error.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#include <assert.h>
#include <sys/time.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <stdio.h>

# fi := fault injection
ssize_t read_fi (int fildes, void *buf, size_t nbyte)
{
      nbyte = (rand() % nbyte) + 1;
      return read (fildes, buf, nbyte);
}
```

```
ssize_t read_all (int fildes, void *buf, size_t nbyte)
{
      assert (fildes >= 0);
      assert (buf);
      assert (nbyte >= 0);
      size_t left = nbyte;
      while (1) {
            int res = read (fildes, buf, left);
            // printf ("%d\n", res);
            if (res < 1)
                  return res;
            buf += res;
            left -= res;
            assert (left >= 0);
            if (left == 0)
                  return nbyte;
      }
}

int main (void)
{
      srand(time(NULL));

      int fd = open ("splay.py", O_RDONLY);
      assert (fd >= 0);

      struct stat buf;
      int res = fstat (fd, &buf);
      assert (res == 0);

      off_t len = buf.st_size;
      char *definitive = (char *) malloc (len);
      assert (definitive);

      res = read (fd, definitive, len);
      assert (res == len);

      int i;
      char *test = (char *) malloc (len);
      for (i=0; i<100; i++) {
            res = lseek (fd, 0, SEEK_SET);
            assert (res == 0);
            int j;
            for (j=0; j<len; j++) {
                  test[j] = rand();
            }
            res = read_all (fd, test, len);
            assert (res == len);
            assert (strncmp(test, definitive, len) == 0);
      }

      return 0;
}

/*  gcc read.c -o read
./read  */
```

7. Fixing the test

- For demo see the corresponding video from Udacity lesson 5.
- 100 test cases passed over read_all in that tiny amount of time it took us to run, so one thing we could conclude is that the logic is solid but it turns out that this conclusion isn't really warranted.
- After printing out a value indicating the number of bytes that was actually read by the raw read system call we run our testing loop and see that every single time it read the full size of the file (i.e., 3121). So, we need to make read sometimes return less file bytes than we asked to test our logic. One way is **hack Mac OS** or **insert faults** by calling a **read function** with fault injection, namely read_fi.
- The function read_fi has exactly the same interface as read just a different filename, but what it's going to do is instead of reading the nbytes it's going to set the number of bytes to read to be a random number between 1 and the number of bytes inclusive.
- In read_all function we call read_fi instead of read:

```
int res = read_fi (fildes, buf, left);
```

8. Testing fault injection

- The first time read_all is called, we get numbers in the range from 1-3121, so is our confidence in the software now higher? Probably there are other tests we should do.
  - For example, read_fi function instead of being random just reads 1 byte every time. That might end up being a reasonable stress test.
  - Another thing we might do is simulate random end of file conditions and random errors that read is allowed to return.
- So during our testing, the read system call never actually returned an error value. It always read the file successful, but if we want to use fault injection to make it do that, then we have to modify our program a little. Just not here.
- We run the test sequence 1,000,000 times instead of 100 times.
- We've established that the logic here is fairly solid.

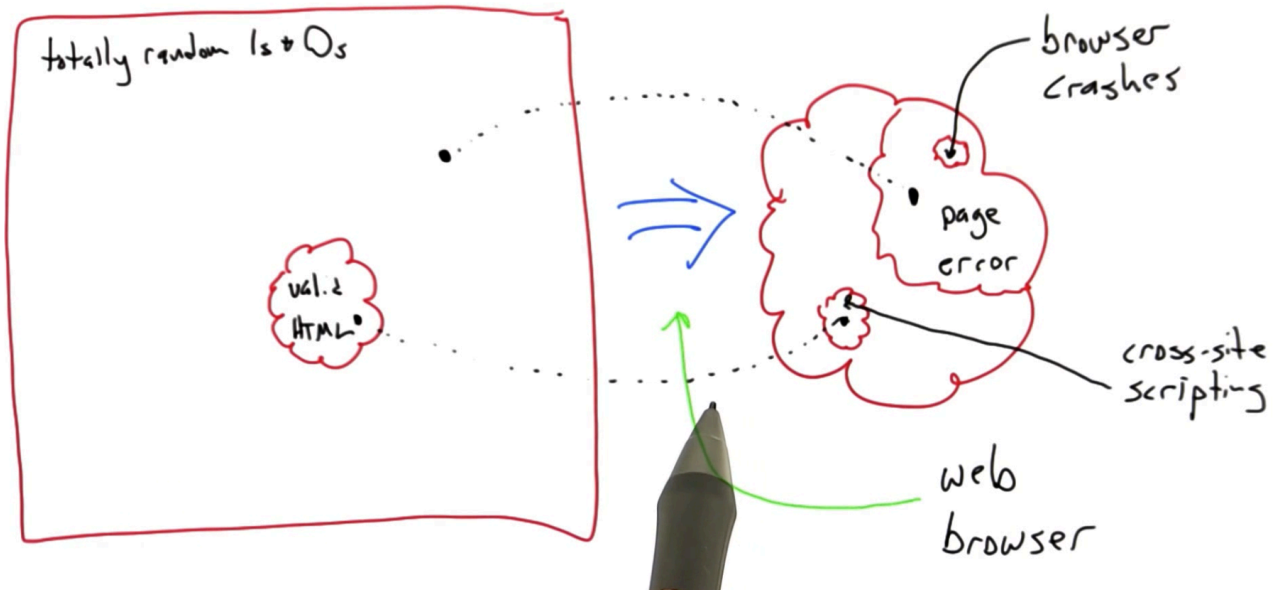
9. How it fits in the loop

- We go back to our master diagram.
  - What we have is a driver's script which in this case was just the C program.
  - We have a random test case generator which consists of 2 lines of code, one of them used a random number to compute the number of bytes to read and the other one, actually called the read function.

- The SUT was the read_all function.
- The oracle was implemented by memory comparison function, i.e. actually read the right bytes from a file, and as we saw, the oracle never detect an error since we got the function right and everything work out.

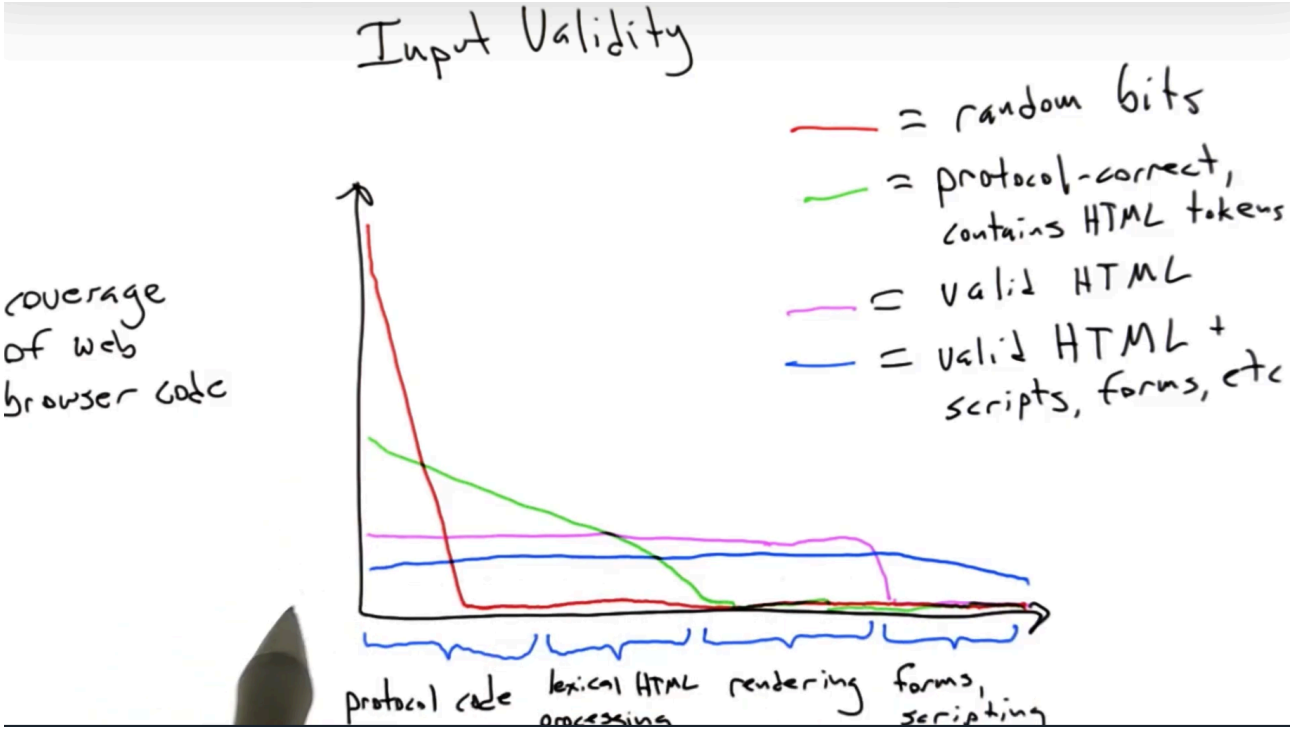
10. Input validity

- We learned from the previous example that building a random test case generator doesn't have to be very difficult.
- But realistically, it's usually a little bit more involved than the one we just saw. The key **problem** is **generating inputs that are valid,** i.e. inputs that are part of the input domain for the SUT.
- Let's say that we're testing a web browser. How much of the space of totally random 1s and 0s constitutes a valid input for a web browser?
- Almost all arbitrary combinations of 1s and 0s fail to create valid web pages so there's going to be some small subset of the set of random inputs that constitutes valid inputs to the web browser and if we take one of these other inputs and hand it to the web browser there's going to mapped to a part of the output space that corresponds to a malformed HTML.
- We want to take some tests from this broader set of completely random inputs but most from a set of valid webpages. These end up exposing something like cross-site scripting bugs.



11. Random browser input

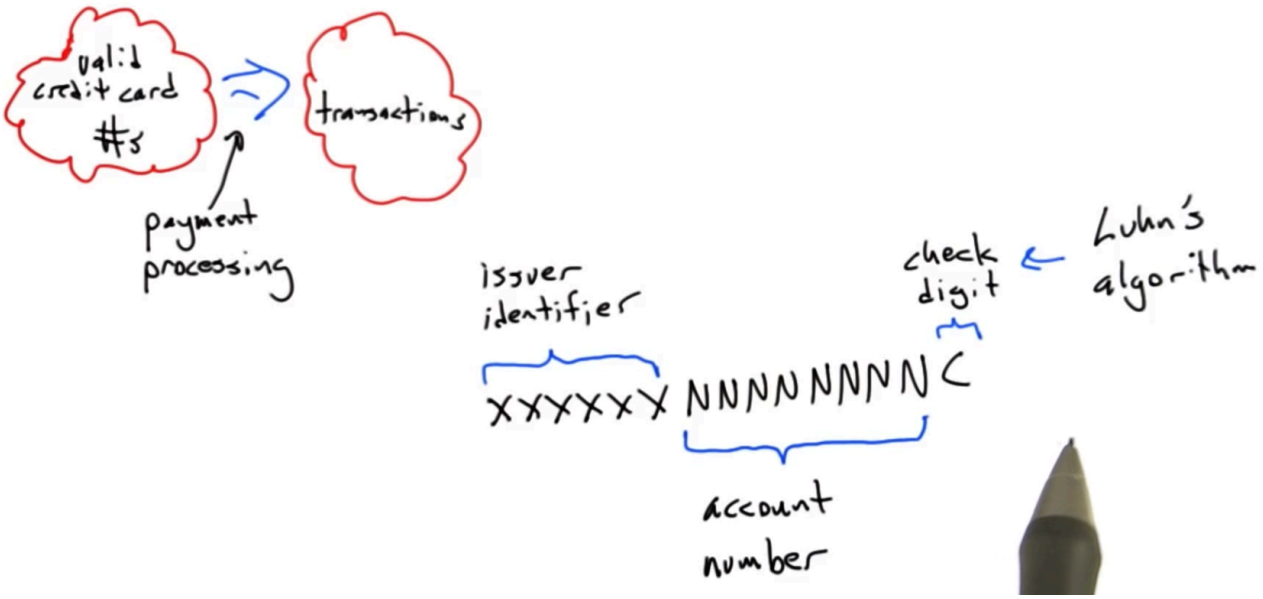
- Let's take a little bit different view of the same problem and draw a graph on the level of code coverage with random test cases induced on the SUT.



- So, in most cases when we do random testing what we're looking for is something like the blue “flat” line, which indicates that we're covering all parts of the SUT roughly equally. Often requires quite a lot of work, quite a lot of sensitivity to the structure of the input demand, but on the other hand, we get paid back for that work with random tests that can exercise the entire SUT and that's going to be a valuable thing in many cases.

12. Generating credit card numbers

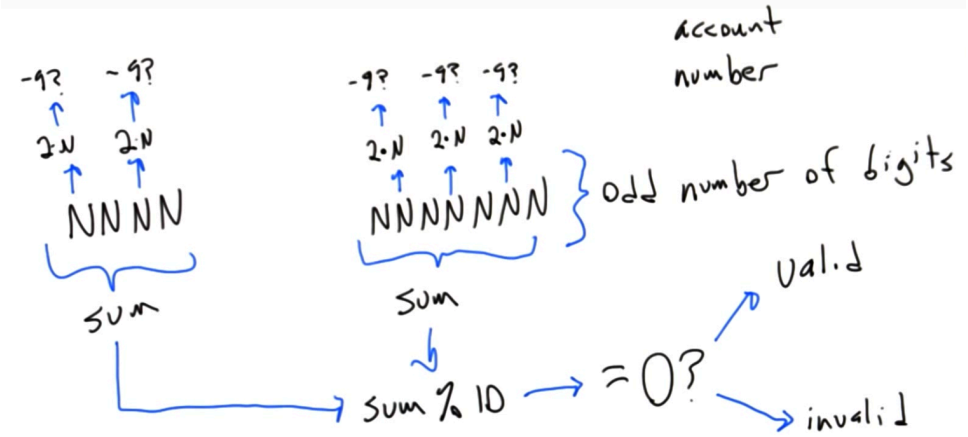
- Let’s assume that we work on some SUT whose input domain is valid credit card numbers. **SUT** is probably doing something like **processing payments** and its output is going to be something like **completed transactions**.
- We can test a variety of credit cards by running through our processing system to see if it works. After finishing we wonder if we removed all the bugs out of the code and the answer is probably no.
- So what we to want to do is **randomly generate valid credit card numbers** and use that to test the payment processing system.
- Let’s take a look at **the structure of a credit card number**.



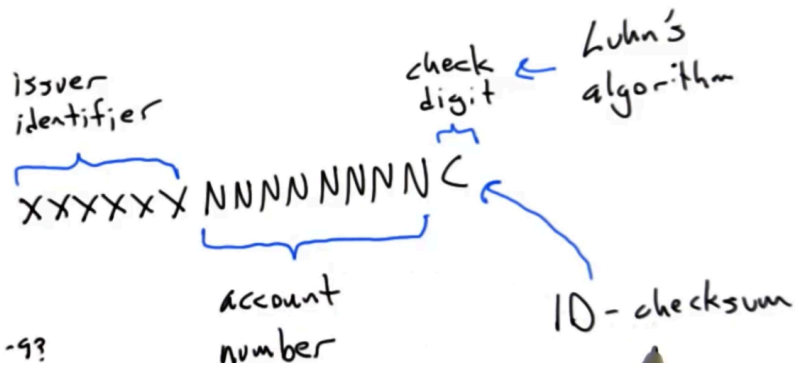
- The check digit is computed algorithmically from the preceding numbers, and **the function of the check digit is to serve as a validity check for a credit card number**.
- The check digit is computed using Luhn's algorithm.

13. Luhn's algorithm

- For a sequence of numbers there are 2 cases. If there's an odd number of digits, we're going to do one thing. And if there's an even number, there'll be a slight variation.



- In order to implement Luhn's algorithm there's one little detail left off. In the above we presented a way to check whether the number is valid. We need to create a valid credit card number. So, given the issue identifier **we will create the account number randomly.** Check digit is 10 - checksum computed using the



algorithm.

14. Luhn's algorithm cont

- We will write a random tester for the credit card transaction processing system.
- **The core of this random tester is a function called generate** that take 2 parameters.
  - The 1st one is the prefix, which corresponds to the issuer identifier, a sequence of digits which is given and has to appear at the start of the credit card number.
  - The 2nd parameter is the length, an integer parameter that determines the total length in digits of the credit card number.
  - We construct a credit card number, which has the given issuer identifier or prefix, the required total length, completely random integers in the middle, and a valid checksum digit as computed using Luhn's algorithm. A procedure will turn the Luhn checksum into a digit, which makes an overall credit card number's checksum come out to be 0 and therefore be valid.
  - Remark: **Python strings are indexed starting at 0.**

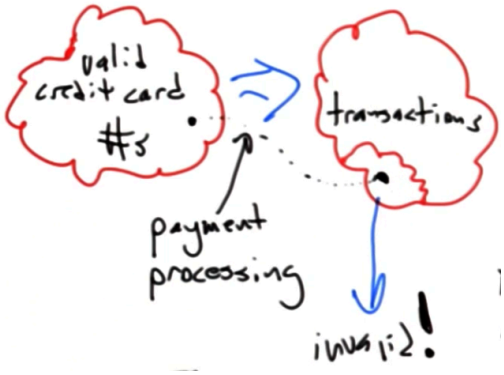
```
# concise definition of the Luhn checksum:
#
# "For a card with an even number of digits, double every odd numbered
# digit and subtract 9 if the product is greater than 9. Add up all
# the even digits as well as the doubled-odd digits, and the result
# must be a multiple of 10 or it's not a valid card. If the card has
# an odd number of digits, perform the same addition doubling the even
# numbered digits instead."
#
# Implement the Luhn Checksum algorithm as described above.

# is_luhn_valid takes a credit card number as input and verifies
# whether it is valid or not. If it is valid, it returns True,
# otherwise it returns False.

import random

def luhn_digit(n):
    n = 2 * n
    if n > 9:
        return n - 9
    else:
        return n
```

```
def luhn_checksum(n):
    l = len(n)
    sum = 0
    if l % 2 == 0:
        for i in range(l):
            if (i+1) % 2 == 0:
                sum += int(n[i])
            else:
                sum += luhn_digit(int(n[i]))
    else:
        for i in range(l):
            if (i+1) % 2 == 0:
                sum += luhn_digit(int(n[i]))
            else:
                sum += int(n[i])
    return sum % 10

def is_luhn_valid(n):
    return luhn_checksum(n) == 0

def generate(pref,l):
    nrand = l - len(pref) - 1
    assert nrand > 0
    n = pref
    for i in range(nrand):
        n += str(random.randrange(10))
    n += "0"
    check = luhn_checksum(n)
    if check != 0:
        check = 10 - check
    n = n[:-1] + str(check)
    return n

def check(pref,l,num):
    if len(num) != l:
        return False
    preflen = len(pref)
    if num[:preflen] != pref:
        return False
    return is_luhn_valid(num)

pref = "372542"
# pref = "37"
print(generate(pref, 15))  # American Express system

for i in range(10000):
    n = generate(pref, 15)
    assert check(pref, 15, n)

# N = 100000
# valid = 0
# for i in range(N):
#     n = str(random.randint(0,1000000000000000)).zfill(15)
#     if check(pref, 15, n):
#         valid += 1
# print(str(valid) + " valid out of " + str(N))
```

```
import random

def luhn_checksum_wikipedia(card_number):
    def digits_of(n):
        return [int(d) for d in str(n)]
    digits = digits_of(card_number)
    odd_digits = digits[-1::-2]
    even_digits = digits[-2::-2]
    checksum = 0
    checksum += sum(odd_digits)
    for d in even_digits:
        checksum += sum(digits_of(d*2))
    return checksum % 10

def is_luhn_valid_wikipedia(card_number):
    return luhn_checksum_wikipedia(card_number) == 0

def generate(pref, l):
    nrand = l - len(pref) - 1
    assert nrand > 0
    n = pref
    for i in range(nrand):
        n += str(random.randrange(10))
    n += "0"
    check = luhn_checksum_wikipedia(n)
    if check != 0:
        check = 10 - check
    n = n[:-1] + str(check)
    return n

def check(pref,l,num):
    if len(num) != l:
        return False
    preflen = len(pref)
    if num[:preflen] != pref:
        return False
    return is_luhn_valid_wikipedia(num)

pref = "372542"
# pref = "37"
print(generate(pref, 15))  # American Express system

for i in range(10000):
    n = generate(pref, 15)
    assert check(pref, 15, n)

# N = 100000
# valid = 0
# for i in range(N):
#     n = str(random.randint(0,1000000000000000)).zfill(15)
#     if check(pref, 15, n):
#         valid += 1
# print(str(valid) + " valid out of " + str(N))
```

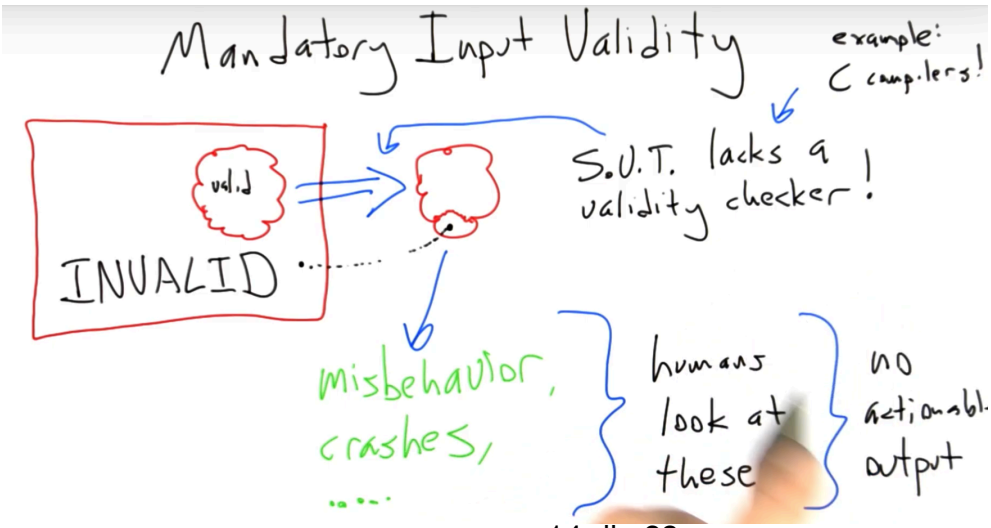
15. Problems with random tests

- Above we have a code from Wikipedia that does the same thing, i.e. the function luhn_checksum_wikipedia; the other functions are an adaptation presented earlier.
- We generate completely random 15-digit numbers (see the above commented code)
- If we consider the prefix 372542 and run the code we get no valid credit card numbers out of 100 000. **The problem is the prefix is too long**.
- With a 6-digit prefix, the chance is one in a million that we'll generate just this prefix and then it goes down to one in 10 million that will meet the prefix and the checksum requirement.
- If we start off with a much smaller prefix like 37 and generate 100 000 credit card numbers, 104 (the no. is different for every execution) of them are valid. So even with just a 2-digit prefix, it's pretty unlikely that we generate valid credit card numbers. If we're generating lots of invalid credit card numbers we're stressing only a very small bit of a transaction processing logic that checks for valid credit card numbers.



16. Mandatory input validity

- In the two examples that we just looked at, there was a vast space of inputs and some small part of that constituted the actual input domain for the SUT, therefore random testing loop is going to spend a lot of time spinning, i.e. **stressing only a small part of the SUT**. But there's actually something different that can happen that's much worse than that. An **invalid input might not be rejected because the SUT doesn't contain sufficient validity checking logic** in order to distinguish valid inputs from invalid inputs.

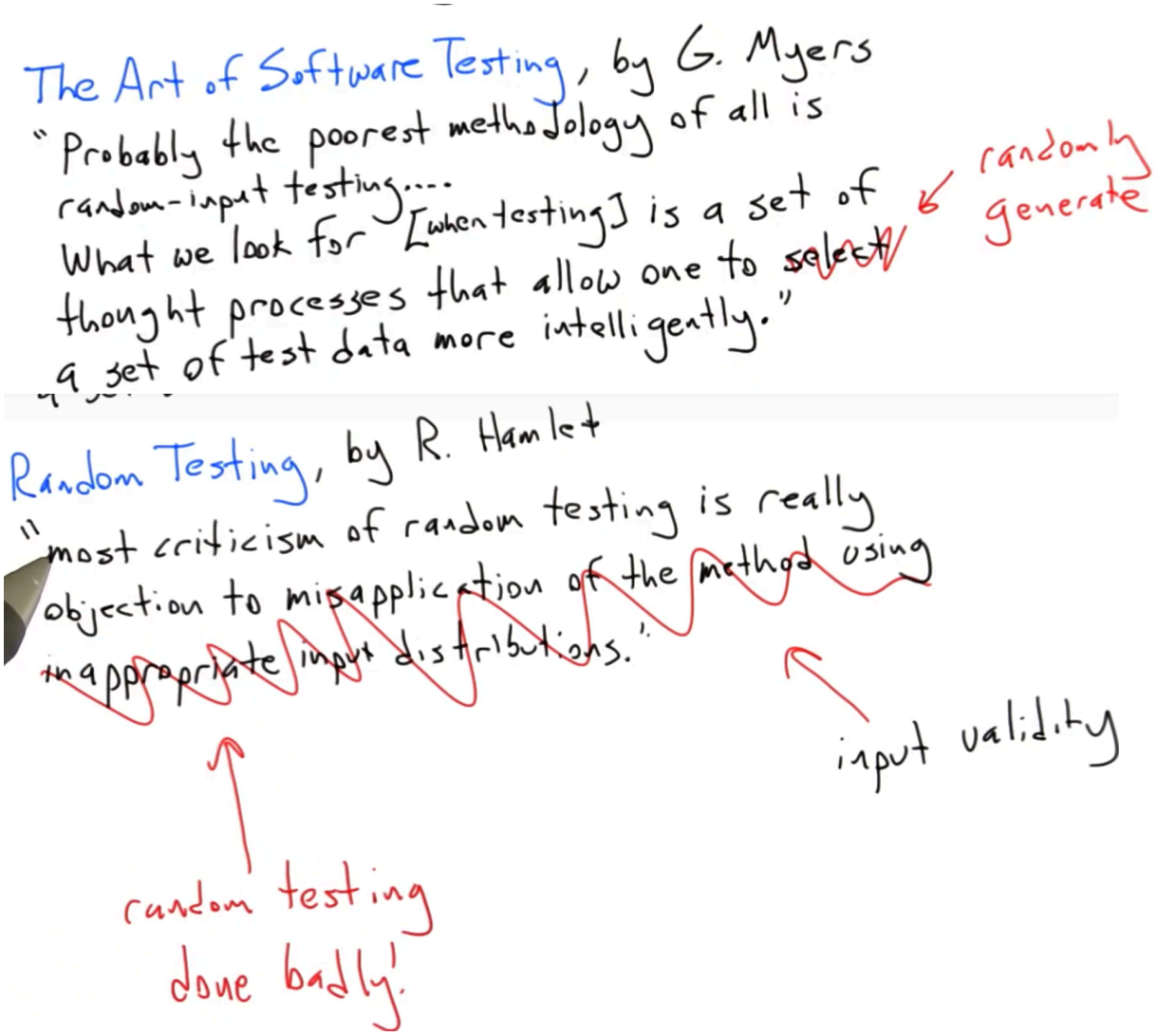


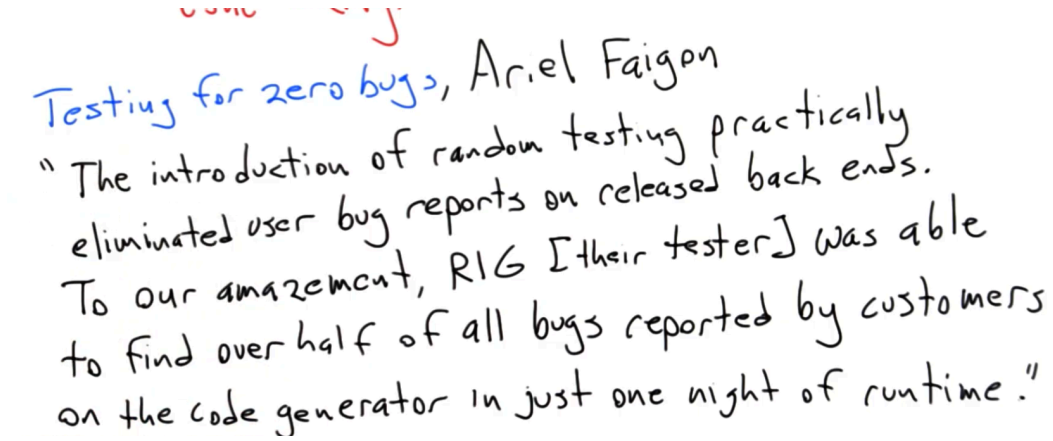
- SUT might lack a validity checker for performance reasons, for code maintainability reasons or there exists software for which it is impossible to construct a validity checker. Compilers look for syntactically invalidacy and fail to check for the dynamic validity properties that are required for certain kinds of miscompilation bugs.
- Therefore, **when we write a random tester, the input validity problem leads us to a performance problem and a much more serious problem when the SUT lacks a reliable validity checker.**

17. Complaints about random testing

- An issue that eventually confronts almost everybody who works on random testing is that random testing gets no respect, i.e. people often think that it's a really stupid way to test software.

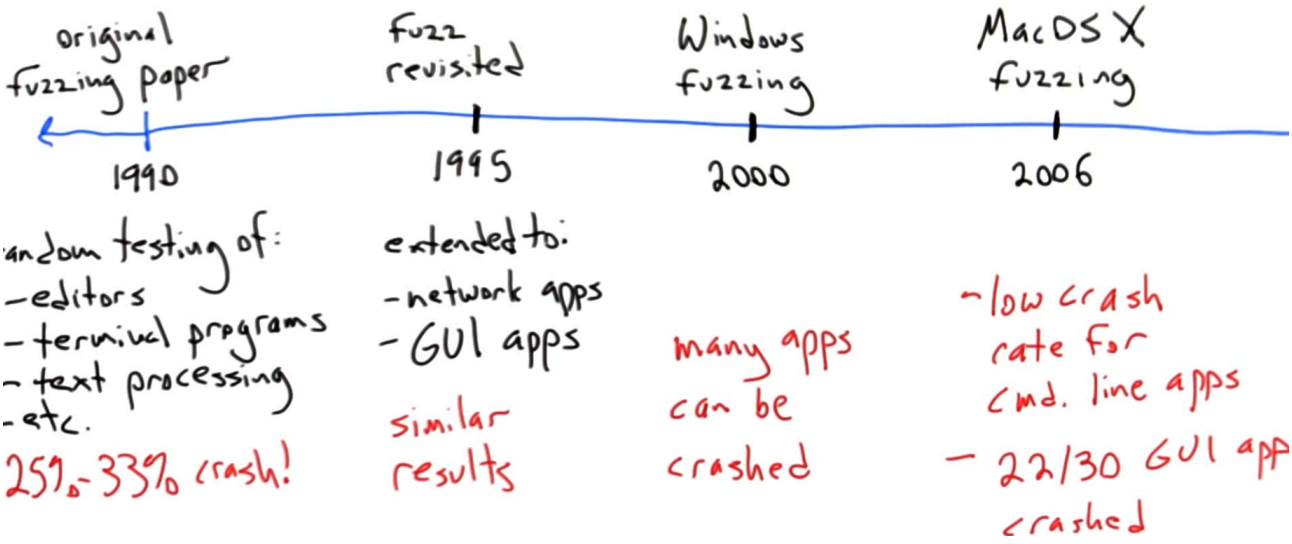
  Corrected excerpts from classic references:





18. Random testing vs fuzzing

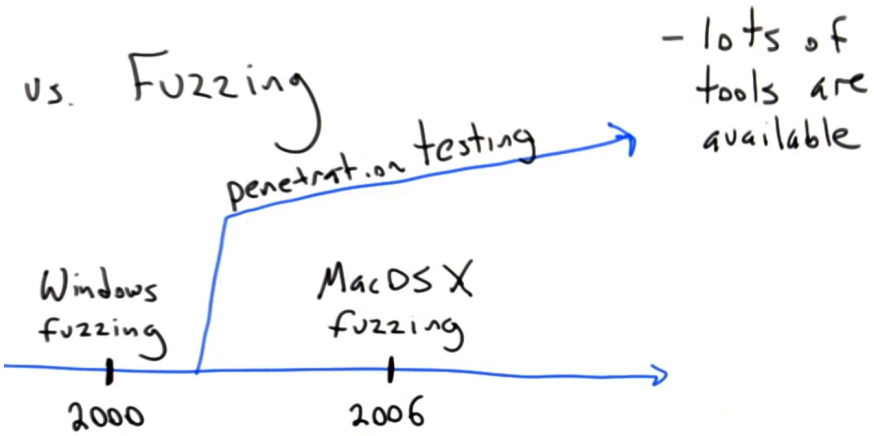
- They are the same thing. In 1990 the professor Bart Miller and his students published a paper called An Empirical Study of the Reliability of Unix Utilities.



- So they had to keep evolving their tools but the input generation methodology that they used, i.e. basically generating random garbage and not really worrying about the input validity problem remained the same across all of these studies.
- What we covered so far was this particular random testing effort by this one research group.
- Something interesting happened **sometime around 2000 or a little after: the term fuzzing took on another use.**

19. Fuzzing for penetration testing

- Around **2000** the connotation of the term **fuzzing** was **penetration testing,** i.e. **finding security vulnerabilities in applications**.



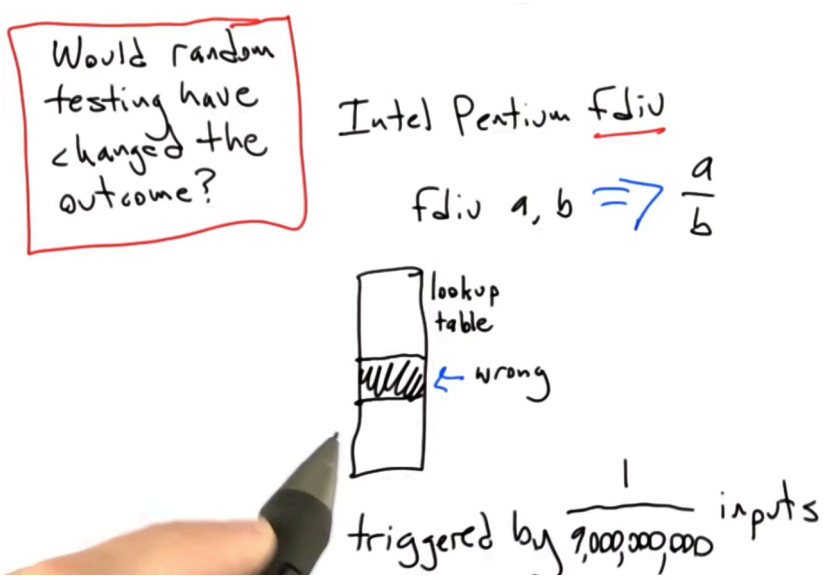
- For example we might find some sort of a machine on the Internet that offers a service and we'll do random testing over the Internet of that service with the goal of finding bugs in the service that will let us mount an attack such as a denial of service or an exploitable vulnerability for mounting an intrusion.

20. Fuzzing for software robustness

- This course is mostly not going to be about fuzzing in the sense of penetration testing.
- Rather, the emphasis of the course is more on random testing in the original sense of fuzzing, i.e. trying to find flaws in software.
- We're concerned with random testing in the general software robustness.

21. Alternate histories

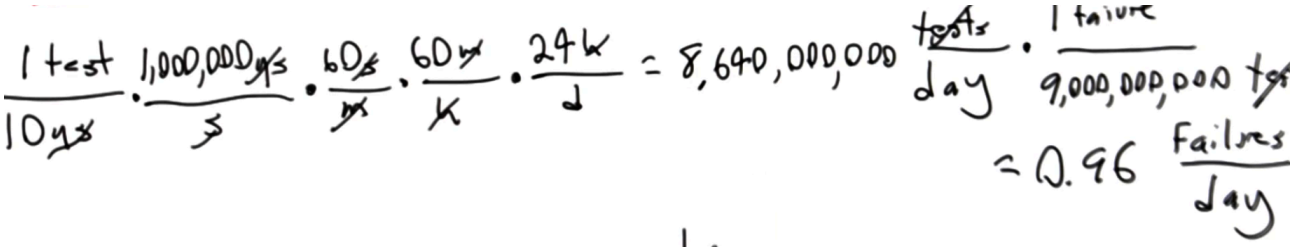
- A genre of fiction where we explore what would have happened to the world had some historical event turned out differently. For example, we might have an alternate history now exploring what would have happened if the Allies hadn't won World War II.
- What we're going to do is look at a few alternate history examples where we're going to try to explore the question, **"What would've happened if certain groups of people in the past had used random testing in addition to whatever other kinds of testing they were doing?",** more specifically, **"Would random testing have been likely to alter the outcome of some famous software bugs but would have not made much difference?**
- Let's look at is the fdiv bug in Intel Pentium chips. In 1994, it came to the world's attention that the fdiv instruction used to perform floating point division was flawed.
- Some of the values were not loaded correctly into the table. 5 entries contained the wrong results. This wasn't an extremely major error in the sense that it would return a million instead of 0 but rather it was off in some number of places after the decimal point.



- Given this relatively low observed **failure rate** on random inputs (1/9 000 000 000), would random testing have been a good way to find the Pentium fdiv bug? The answer is almost certainly yes.
- **How long it would've taken Intel to find this bug using random testing?**
- Let's say, for the sake of argument, that in 1994 it took 10 microseconds to perform an fdiv and verify its result.

22. Fdiv

- By assumption, we can run 1 test every 10 microseconds.

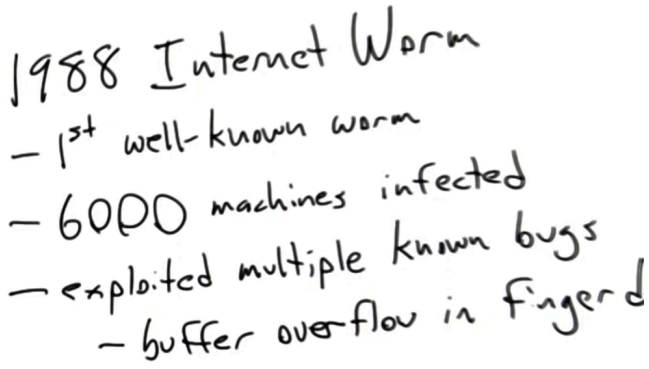


- If we perform completely random testing of the input space for fdiv, we should be able to find this bug in about a day.
- Now this kind of testing is going to need some sort of an oracle to tell if our particular output from fdiv is correct or not, i.e. IEEE floating point.

  Probably **Intel's existing 487 FPU**.
  - FPU stands for floating point unit.

23. 1988 Internet worm

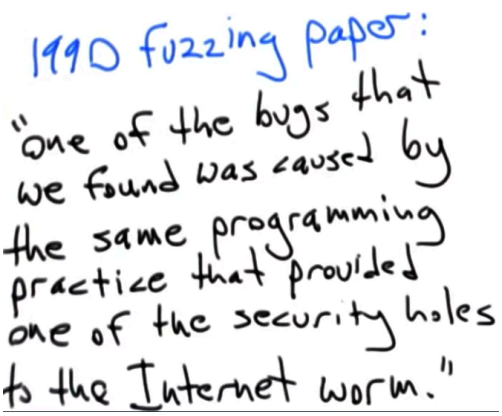
- 1988, the Internet was not particularly well known to the general public, and it had a relatively small number of users. And even so, this worm infected an estimated 6,000



machines, a substantial fraction of the number of machines connected to the Internet at the time.

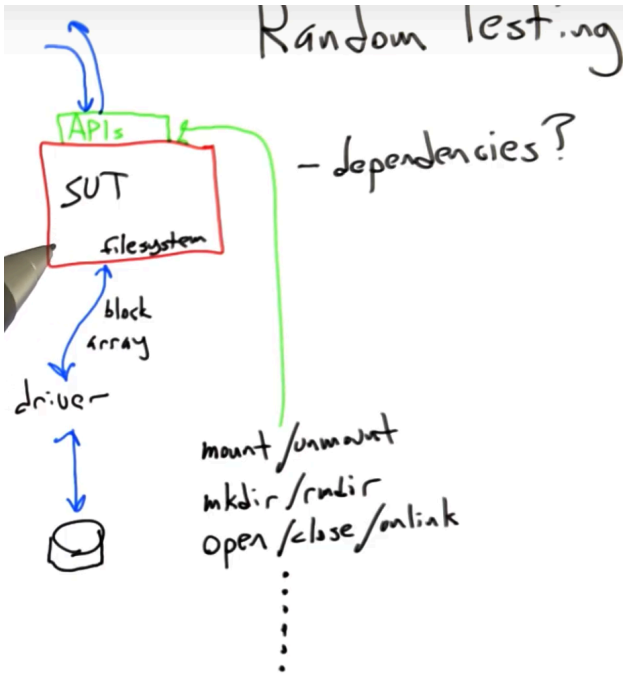
•Finger daemon was a service that ran on UNIX machines at the time, and let you query a remote machine to learn about whether a user was logged into that machine and some other stuff.

- **Would random testing have changed the outcome?**
- **Could this bug in finger daemon and lots of other bugs like it have been found by random testing?** The answer to the question is probably yes.
- If we go back to the original fuzzing paper,



24. Random testing of APIs

- We've been looking at a progression of random testers, from the simpler ones to the more complicated ones.
- One of the major things creating difficulties for us is the structure required in inputs.
- In the following we're going to look at the next level of structured inputs, required for random testing of APIs.



•We have some **SUT, and it's providing an API** (a collection of function calls that can be invoked) or several APIs for other code to use.

•In this case **a single random test is a string of API calls**. The situation is more complicated because of the **dependencies** (în **lb. română** traducem **dependențe**!) among API calls.

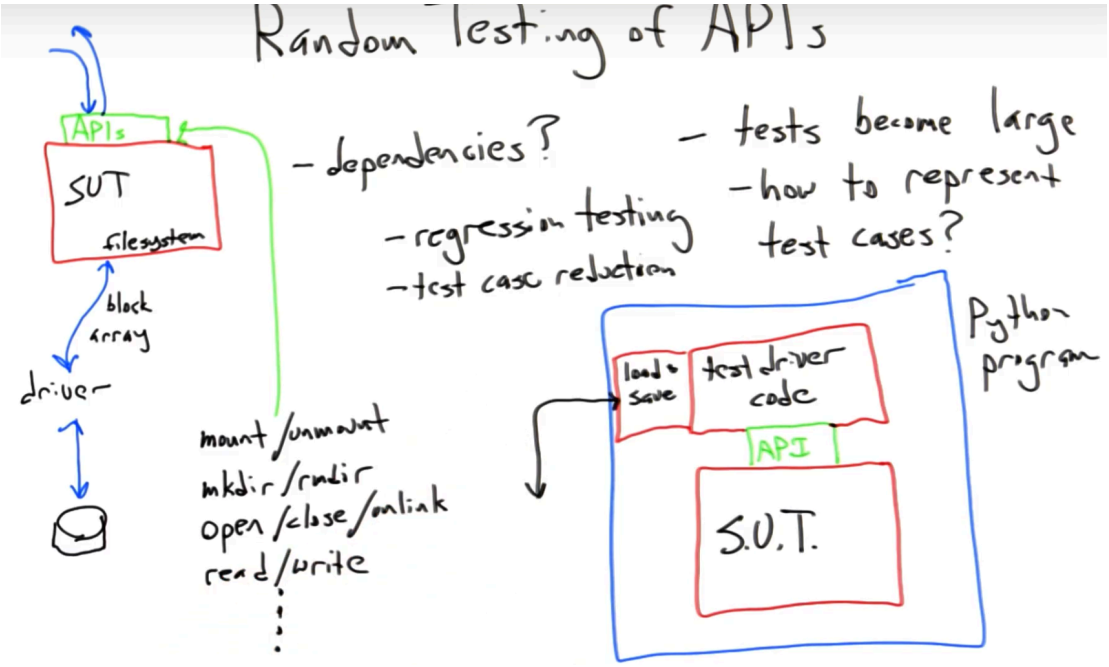
•Let's take the example of randomly testing a file system. It has to manage all of the structure just to manage the free space to efficiently respond to all sorts of

calls that perform file system operations.

- **What are the contents of the file system API?** Mount/unmount, etc., i.e. UNIX style file system interface.
- So, if we want to do random testing of a file system, we're going to be issuing sequences of calls in the file system API.
- File systems end up being substantially large chunks of code, and what's more, the integrity of our data depends on the correctness of that file system code. For example, if we save some critical data to the disk and the file system messes it up, i.e. it saves it in some wrong place or corrupts it in some other way, then we're not going to get that data back ever. So it's really, really important the file systems work well.

25. Fuzzing filesystems

- We have to do something more systematic than completely random stream of API calls or providing random arguments to those calls.



- For example, open a file and keep track of the fact that it's open so we can randomly generate read and write calls into it. Therefore, we need **to track these dependencies**, in order to issue a sequence of API calls that's going to do reasonably **effective random testing of a file system**.
- The driver code creates the test cases, passes them to the SUT, looks at the results, and keeps going until it finishes. There are a couple cases in which that's not so good.
- One of them is where we find a particular test case that makes our system fail and we'd like to **save off that test case for later use in regression testing**.

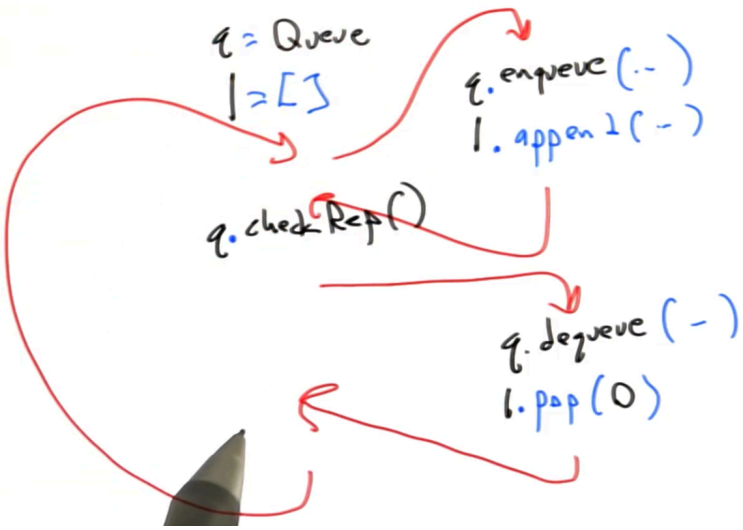
- It's often the case that when random tests become extremely large, we need to turn them into first class objects, i.e. objects living on disk using a save and load routine in order to perform **test case reduction,** a piece of technology that's very often combined with random testing, that takes a large test case that makes the SUT fail and turns it into a smaller test case that still makes the SUT fail.

26. Random testing the bounded queue

- Let’s use the bounded queue presented in previous lectures.
- **checkRep is our assertion checker that does sanity checking over the internal state of the queue.** Below is a simple test routine, a non-random test for the queue.

```
def simple_test():
    q = Queue(2)
    q.checkRep()

    res = q.enqueue(10)
    assert res
    q.checkRep()

    res = q.enqueue(11)
    assert res
    q.checkRep()

    res = q.enqueue(12)
    assert not res
    q.checkRep()

    res = q.enqueue(13)
    assert not res
    q.checkRep()

    x = q.dequeue()
    assert x == 10
    q.checkRep()

    x = q.dequeue()
    assert x == 11
    q.checkRep()

    x = q.dequeue()
    assert x is None
    q.checkRep()

    x = q.dequeue()
    assert x is None
    q.checkRep()

    print("All tests complete")

simple_test()
```

- **How to write a (small) random tester for the queue?**
- One thing we can do is randomly invoke queue operations in a loop and wait for checkRep to fail.
- On the other hand, we know how many elements the queue holds, and we also know whether any particular addition or removal from the queue succeeds or fails. And so it's pretty easy using an integer to keep track of how many elements are currently used in the queue.
- The third thing we can do is actually keep track of what values should be coming out of dequeue operations.
- The bounded queue that we're testing is based on a Python array, but it turns out that it's really easy to emulate the operation of that queue using native Python data structures.
- **l is a list which is mirroring the contents of the queue q.**

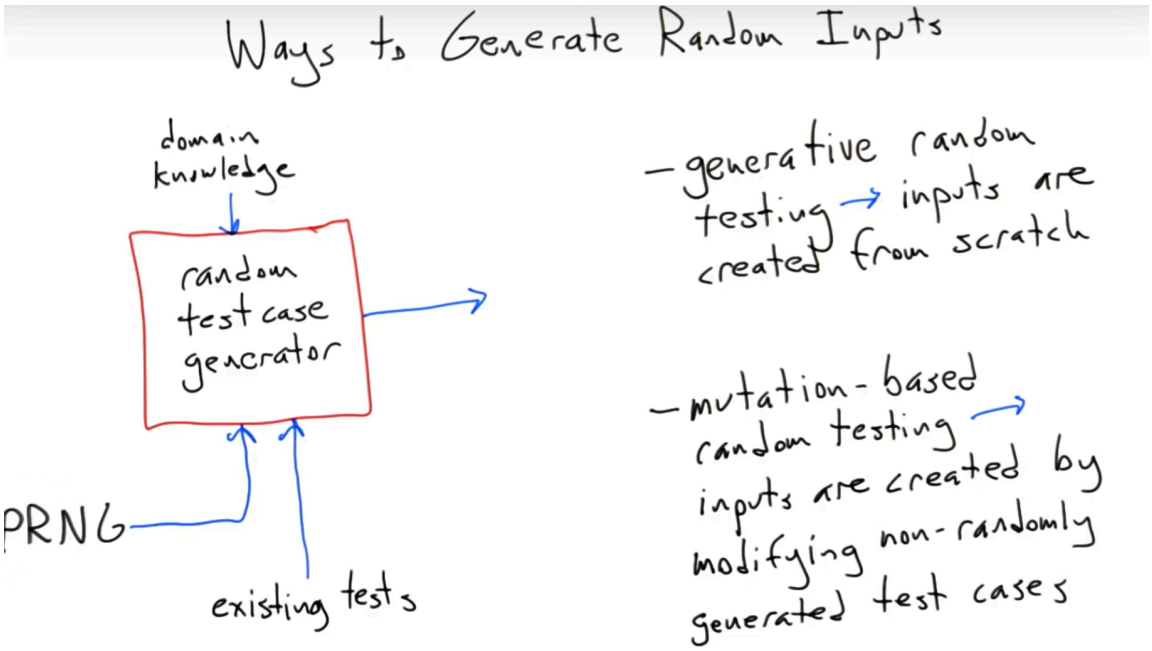



```
# TASK:
# Write a random tester for the Queue class.
# The random tester should repeatedly call the Queue methods
# on random input in a semi-random fashion.
# For instance, if you wanted to randomly decide between
# calling enqueue and dequeue, you would write something like this:
#
# q = Queue(500)
# if (random.random() < 0.5):
#     q.enqueue(some_random_input)
# else:
#     q.dequeue()
#
# You should call the enqueue, dequeue, and checkRep methods
# several thousand times each.
```

```
def random_test():
    N = 4
    add = 0
    remove = 0
    addFull = 0
    removeEmpty = 0
    q = Queue(N)
    q.checkRep()
    l = []
    for i in range(100000):
        if random.random() < 0.5:
            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))

def time_test():
    q = Queue(50000)
    for i in range(50000):
        q.enqueue(0)
        q.checkRep()
    for i in range(50000):
        q.dequeue()
        q.checkRep()

random_test()
time_test()
```

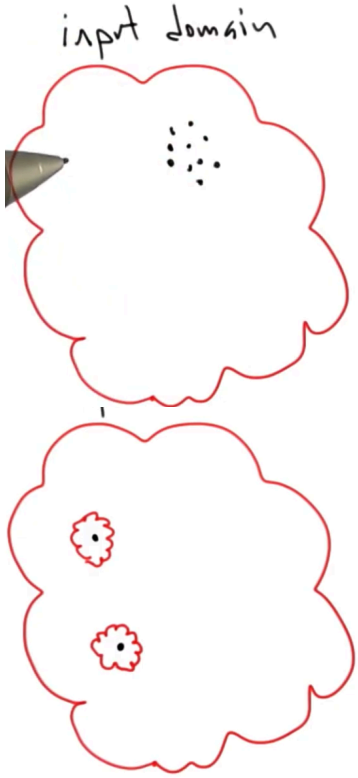
27. Generating random inputs

- The test case generator has 2 inputs: pseudo-random numbers from a pseudo-random number generator, and it's also predicted on some amount of knowledge of the input domain, and that's supplied by a human.
- We looked at one example, testing the adder, where essentially no domain knowledge was needed, i.e. pseudo-random numbers were used directly as test cases.
- We looked at the credit card number generator where pseudo-random numbers parameterized an algorithm which created valid credit card numbers.
- In the file system testing example we need to generate valid sequences of API calls.
- In the web browser testing example we need to actually generate well-formed HTML.
- All of these ways of generating inputs are variations on a single theme which we call **generative random testing**, i.e. inputs are created from scratch.
- There's an entirely different approach called **mutation-based random testing**, i.e. inputs are created by randomly modifying existing non-randomly created inputs by the SUT.



- Which approach is better? Sometimes generative works really well, other times mutation-based testing works really well. It just depends on the situation and how much time we have to spend.

28. Mutation based random testing



- A generative random tester will test inputs from a cluster found in some part of the input domain.
- A mutation-based random tester will start with some known input, will randomly modify it ending with test cases that are in the same neighborhood as the original input. So we're exploring interesting parts of the input domain, that we could have never reached this part of the input domain using any kind of a generative random test case generator.
- The last approach is very useful and valid, but on the other hand, it's generally limited to exploring a region of the input space that's close to the starting point.

29. Generating mutators

- **What are the operators that we use to mutate test cases to create new random test cases?** The most obvious thing to do is start flipping bits.
- So, for example, we take a Word document and we flip 10, 100, maybe a couple thousand bits in it, we load it up in Microsoft Word, and we see if we can find a part of the range for Microsoft Word that causes it to crash.
- Another thing we can do, and this is one of the techniques often used by penetration testing tools based on fuzzing, is modify selected fields. Let's say our test case has some known structure: a valid HTTP request. We will target parts of the HTTP request that are known to frequently lead to vulnerabilities in servers and we will selectively modify those.

How should we mutate test cases?
- flip bits
- modify selected fields
- add / remove / replace tokens, sub-trees of the AST

- If we have a parser we modify our test case using its grammar. For example, add or remove or replace tokens in a test case or also subtrees of the AST. AST stands for abstract syntax tree.

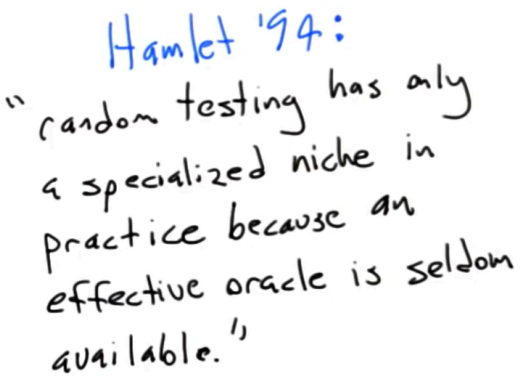
- **Mutational fuzzer**: a 5-line Python program by Charlie Miller who claims that he found an enormous number of bugs with it.
  - "**Babysitting an Army of Monkeys**: An analysis of fuzzing 4 products with 5 lines of Python" was the title of a talk by Charlie Miller in CodenomiCON 2010 (see the YouTube video).
- It turns out that the 5 lines of Python are missing quite a bit of code. See the added comments explaining what these are.
  - We load the file we want to mutate for purposes of creating a random test case into a buffer in memory.

```
# load a file into buffer
# at a random position of the buffer

# change the byte to a random one (5 lines of python code)
numwrites = random.randrange(math.ceil((float(len(buf)) / FuzzFactor))) + 1
for j in range(numwrites):
      rbyte = random.randrange(256)
      rn = random.randrange(len(buf))
      buf[rn] = "%c"%(rbyte)

# save the buffer

# run the process

# look at the exit code

# if it doesn't die (no bug found), kill it

# start over
```

- It turns out that a lot of people have been using this sort of tool on the common utility programs like Acrobat Reader or MS Office Suite. We may not find anything on the latest versions. If we want to actually get some bugs we should find old versions and fuzz them. Almost certainly we will find problems using this kind of infrastructure.

30. Oracles

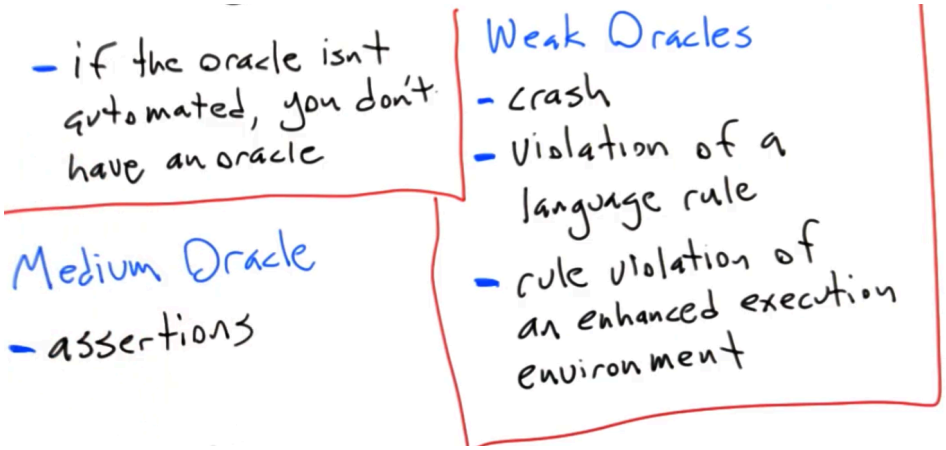
- Oracles are extremely important for random testing because **if you don't have** an automated **oracle**, i.e. if you don't have an automated way to tell if a test case did something interesting **then you've got nothing**.
- There are quite few oracles available but almost always we can make something work even if it's just a weak oracle like watching out for crashes.



- Let's organize the oracles that are suitable for random testing into a collection of categories.
- **Weak oracles** are some of the ones that are most useful in practice. They can only enforce fairly generic properties about SUT.

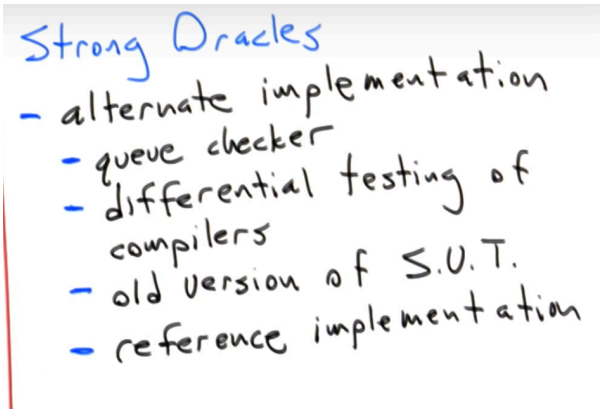
31. Medium oracles

- In C or C++ we could run the compiled program under a **runtime monitor** like **Valgrind** which provides an enhanced execution environment that checks a lot of rules unchecked by the regular C and C++ runtime or by the OS. For example, Valgrind will terminate our process if we access beyond the end of an array.
- Another example of an enhanced execution environment would be one that checks, for example, integer overflow or floating point problems.



- One example of a medium power oracle is assertion checks that the programmer has put into the software.
- A medium power oracle doesn't guarantee anything even remotely close to actual correct operation.

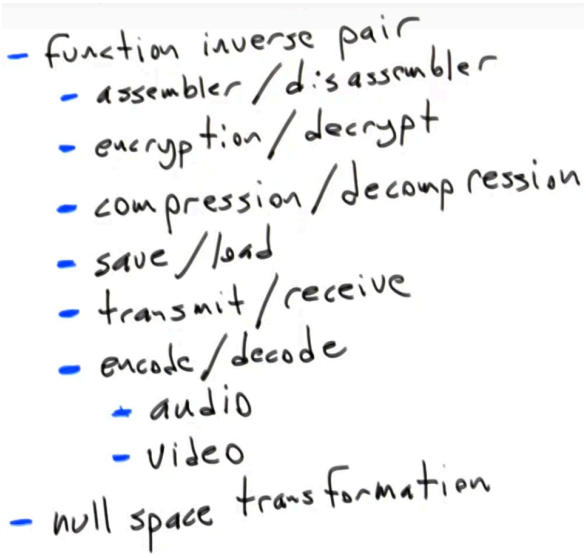
32. Strong oracles



- Strong oracles are extremely useful and we should always use one if we can possibly find one.
- An important one is having an alternate implementation. The random tester for the bounded queue included a 2nd implementation of the same queue abstraction, i.e. a Python list used to actually check that the queue gave the right answers.

- Differential testing of compilers means we have multiple implementations of the same compiler specification and we expect them to behave the same given equivalent inputs.
- Looking at an older version of SUT means that we've broken it.
- The best alternate implementation we could have is a **reference implementation**, i.e. an implementation of the specification that we trust.

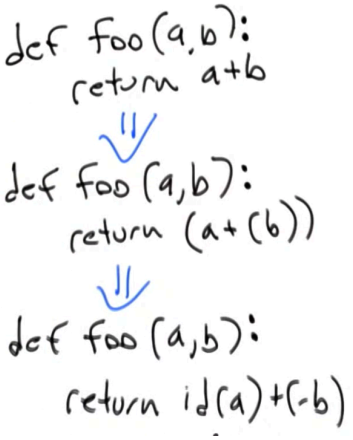
33. Function inverse pairs



•Another strong oracle is function inverse pair. We have available some function and also its inverse and we can use these as a pair to do strong checking of correct behavior of the SUT.

•The final strong oracle we talk about is null space transformation.

34. Null space transformations



- We take a random test case or any test case and we make some change to it that shouldn't affect how it's treated by the SUT.
- Let's consider a simple Python function, namely foo. One possible null space transformation would be to add a level of parentheses or maybe several.
- We can do something very much like differential testing but instead of taking the same program and running it through 2 implementations of Python we're going to take 2 programs which are semantically identical and run them through the same implementation of Python. If it interprets this code in such a way that it returns some different answer, then we've found a bug in it.
- We could always do things even more elaborate (using the identity function, etc.)
- So, there are a lot of potential oracles available for performing random testing. If we're potentially willing to invest time in creating a good oracle, then we can very often find something to use for random testing.

Notă[1]

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