Cuprins

1. Testare funcțională

- Datele de test sunt generate pe baza specificației (cerințelor) programului, structura programului nejucând nici un rol.
- Tipul de specificație ideal pentru testarea funcțională este alcătuit din pre-condiții și post-condiții.
- Majoritatea metodelor funcționale se bazează pe o partiționare a datelor de intrare astfel încât datele aparținând unei aceeași partiții vor avea proprietăți similare (identice) în raport cu comportamentul specificat.

(a) Partiționare de echivalență (equivalence partitioning)

- Ideea de bază este de a partiționa domeniul problemei (datele de intrare) în *partiții de echivalență* sau *clase de echivalență* astfel încât, din punctul de vedere al specificației datele dintr-o clasă sunt tratate în mod identic.
- Cum toate valorile dintr-o clasă au specificat același comportament, se poate presupune că toate valorile dintr-o clasă vor fi procesate în același fel, fiind deci suficient să se aleagă câte o valoare din fiecare clasă.
- În plus, domeniul de ieșire va fi tratat în același fel, iar clasele rezultate vor fi transformate în sens invers (reverse engineering) în clase ale domeniului de intrare.
- Clasele de echivalență nu trebuie să se suprapună, deci orice clase care s-ar suprapune trebuie descompuse în clase separate.

- Dupa ce clasele au fost identificate, se alege o valoare din fiecare clasă. În plus, pot fi alese și date invalide (care sunt în afara claselor și nu sunt procesate de nici o clasă).
- Alegerea valorilor din fiecare clasă este arbitrară deoarece se presupune că toate valorile vor fi procesate într-un mod identic.

Exemplu:

Se testează un program care verifică daca un caracter se află într-un șir de cel mult 20 de caractere. Mai precis, pentru un întreg *n* aflat între 1 și 20, se introduc caractere, iar apoi un caracter *c*, care este apoi căutat printre cele *n* caractere introduse anterior. Programul va produce o ieșire care va indica prima poziție din șir unde a fost găsit caracterul *c* sau un mesaj indicand că acesta nu a fost găsit. Utilizatorul are opțiunea să caute un alt caracter tastând *y* (yes) sau să termine procesul tastând *n* (no).

1. Domeniul de intrări:

Exista 4 intrări:

- un întreg pozitiv *n*
- un șir de caractere *x*
- caracterul care se caută *c*
- opțiune de a căuta sau nu un alt caracter *s*
- *n* trebuie să fie între 1 și 20, deci se disting 3 clase de echivalență:

N_1 = 1..20

N_2 = { n | n < 1}

N_3 = {n | n > 20}

- întregul *n* determină lungimea șirului de caractere și nu se precizează nimic despre tratarea diferită a șirurilor de lungime diferită deci a doua intrare nu determină clase de echivalență suplimentare
- *c* nu determină clase de echivalență suplimentare
- opțiunea de a căuta un nou caracter este binară, deci se disting 2 clase de echivalență

S_1 = {y}

S_2 = {n}

2. Domeniul de ieșiri

Constă din urmatoarele 2 răspunsuri:

- Poziția la care caracterul se găseste în șir
- Un mesaj care arată că nu a fost găsit

Acestea sunt folosite pentru a împărți domeniul de intrare în 2

clase: una pentru cazul în care caracterul se află în șirul de caractere și una pentru cazul în care acesta lipsește

C_1(x) = { c | c se află în x}

C_2(x) = { c | c nu se află în x}

Clasele de echivalență pentru întregul program (globale) se pot obține ca o combinatie a claselor individuale:

C_111 = { (n, x, c, s) | n \in N_1, |x| = n, c \in C_1(x), s \in S_1}

C_112 = { (n, x, c, s) | n \in N_1, |x| = n, c \in C_1(x), s \in S_2}

C_121 = { (n, x, c, s) | n \in N_1, |x| = n, c \in C_2(x), s \in S_1}

C_122 = { (n, x, c, s) | n \in N_1, |x| = n, c \in C_2(x), s \in S_2}

C_2 = { (n, x, c, s) | n \in N_2}

C_3 = { (n, x, c, s) | n \in N_3}

Setul de date de test se alcătuiește alegându-se o valoare a intrărilor pentru fiecare clasă de echivalență. De exemplu:

c_111: (3, abc, a, y)

c_112: (3, abc, a, n)

c_121: (3, abc, d, y)

c_122: (3, abc, d, n)

c_2: (0, _, _, _)

c_3: (25, _, _, _)

6 clase

Intrări				Rezultat afișat (expected)
n	**x**	**c**	**s**	
0				Cere introducerea unui întreg între 1 și 20
25				Cere introducerea unui întreg între 1 și 20
3	abc	a	y	Afișează poziția 1; se cere introducerea unui nou caracter
		a	n	Afișează poziția 1
3	abc	d	y	Caracterul nu apare; se cere introducerea unui nou caracter
		d	n	Caracterul nu apare

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void equivalencePartitioning() {
tester.main(new String[]{"0", null, null, null});
tester.main(new String[]{"25", null, null, null});
tester.main(new String[]{"3", "a", "b", "c", "a", "y"});
tester.main(new String[]{"3", "a", "b", "c", "a", "n"});
tester.main(new String[]{"3", "a", "b", "c", "d", "y"});
tester.main(new String[]{"3", "a", "b", "c", "d", "n"});
}
```

Avantaje:

- Reduce drastic numărul de date de test doar pe baza specificației.
- Potrivită pentru aplicații de tipul procesării datelor, în care intrările și ieșirile sunt ușor de identificat și iau valori distincte.

Dezavantaje:

- Modul de definire al claselor nu este evident (nu există nici o modalitate riguroasă sau măcar niște indicații clare pentru identificarea acestora).
- În unele cazuri, deși specificația ar putea sugera că un grup de valori sunt procesate identic, acest lucru nu este adevărat. Acest lucru întărește ideea că metodele funcționale trebuie aplicate împreună cu cele structurale.
- Mai puțin aplicabile pentru situații când intrările și ieșirile sunt simple, dar procesarea este complexă.

(b) Analiza valorilor de frontieră (boundary value analysis)

Analiza valorilor de frontieră este folosită de obicei împreuna cu partiționarea de echivalență. Ea se concentrează pe examinarea valorilor de frontieră ale claselor, care de obicei sunt o sursă importanta de erori.

Pentru exemplul nostru, odată ce au fost identificate clasele, valorile de frontieră sunt ușor de identificat:

- valorile 0, 1, 20, 21 pentru *n*
- caracterul c poate să se gasească în șirul *x* pe prima sau pe ultima poziție

Deci se vor testa următoarele valori:

- N_1 : 1, 20
- N_2 : 0
- N_3 : 21
- C_1: = c_11 se află pe prima poziție în x, c_12 se află pe ultima poziție în x
- Pentru restul claselor se ia câte o valoare (arbitrară)

Deci, pentru C_111 și C_112 vom alege câte 3 date de test (x de 1 caracter și x de 20 de caractere în care c se găsește pe poziția 1 și pe pozitia 20), iar pentru C_121 și C_122 câte 2 date de test (x de 1 caracter și x de 20 de caractere). În total vom avea 12 date de test.

C_111 : (1, a, a, y), (20, abcdefghijklmnoprstu, a, y), (20, abcdefghijklmnoprstu, u, y)

C_112 : (1, a, a, n), (20, abcdefghijklmnoprstu, a, n),
(20, abcdefghijklmnoprstu, u, n)

C_121: (1, a, b, y), (20, abcdefghijklmnoprstu, z, y)

C_122: (1, a, b, n), (20, abcdefghijklmnoprstu, z, n)

C_2 : (0, _, _, _)

C_3 : (21, _, _, _)

Intrari				Rezultat afișat (expected)
n	**x**	**c**	**s**	
0				
21				
1	a	a	y	
		a	n	
1	a	b	y	
		b	n	
20	abcdefghijklmnoprstu	a	y	
		a	n	
20	abcdefghijklmnoprstu	u	y	
		u	n	
20	abcdefghijklmnoprstu	z	y	
		z	n	

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void boundaryValueAnalysis () {
//…
}
```

Avantaje și dezavantaje:

- Aceleași ca la metoda anterioară.
- În plus, această metodă adaugă informații suplimentare pentru generarea setului de date de test și se concentrează asupra unei arii (frontierele) unde de regulă apar multe erori.

(c) Partiționarea în categorii (category-partitioniong)

Această metodă se bazează pe cele două anterioare. Ea caută să genereze date de test care "acoperă" funcționalitatea sistemului și maximizeaza posibilitatea de găsire a erorilor.

Cuprinde următorii pasi:

1. Descompune specificația funcțională în unitati (programe, functii, etc.) care pot fi testate separat.
2. Pentru fiecare unitate, identifică parametrii și condițiile de mediu (ex. starea sistemului la momentul execuției) de care depinde comportamentul acesteia.
3. Găseste categoriile (proprietăți sau caracteristici importante) fiecărui parametru sau condiții de mediu.
4. Partiționează fiecare categorie în alternative. O alternativă reprezintă o mulțime de valori similare pentru o categorie.
5. Scrie specificația de testare. Aceasta constă în lista categoriilor și lista alternativelor pentru fiecare categorie.
6. Creează cazuri de testare prin alegerea unei combinații de alternative din specificația de testare (fiecare categorie contribuie cu zero sau o alternativă).
7. Creează date de test alegând o singură valoare pentru fiecare alternativă.

Pentru exemplul nostru:

1. *Descompune specificatia în unități*: avem o singură unitate.
2. *Identifică parametrii*: *n, x, c, s*
3. *Găsește categorii*:
    - *n*: dacă este în intervalul valid 1..20
    - *x*: dacă este de lungime minimă, maximă sau intermediară
    - *c*: dacă ocupă prima sau ultima poziție sau o poziție în interiorul lui *x* sau nu apare în *x*
    - *s*: dacă este pozitiv sau negativ
4. *Partiționeaza fiecare categorie în alternative*:
    - *n*: <0, 0, 1, 2..19, 20, 21, >21

- *x*: lungime minimă, maximă sau intermediară
- *c*: poziția este prima, în interior, sau ultima sau *c* nu apare în *x*
- *s*: y, n

5. *Scrie specificația de testare*

- *n*
    1) {n | n < 0}
    2) 0
    3) 1 [ok, lungime1
    4) 2..19 [ok, lungime_medie]
    5) 20 [ok, lungime20]
    6) 21
    7) {n | n > 21}
- *x*
    1) {x | |x| = 1}. [if ok and lungime1]
    2) {x | 1 < |x| < 20}. [if ok and lungime_medie]
    3) {x | |x| = 20} [if ok and lungime20]
- *c*
    1) { c | c se afla pe prima poziție în x} [if ok]
    2) { c | c se afla în interiorul lui x} [if ok and not lungime1]
    3) { c | c se afla pe ultima poziție în x} [if ok and not lungime1]
    4) { c | c nu se afla în x} [if ok]
- *s*
    1) y [if ok]
    2) n [if ok]

Din specificația de testare ar trebui să rezulte 7 * 3 * 4 * 2 = 168 de cazuri de testare. Pe de altă parte, unele combinații de alternative nu au sens și pot fi eliminate. Acest lucru se poate face adăugând constrângeri acestor alternative. Constrângerile pot fi sau proprietăți ale alternativelor sau condiții de selectie bazate pe aceste proprietăți. În acest caz, alternativele vor fi combinate doar

dacă condițiile de selecție sunt satisfacute. Folosind acest procedeu, în exemplul nostru vom reduce numărul cazurilor de testare la 24.

6. Creează cazuri de testare

n1	n2	n3x1c1s1	n3x1c1s2
n3x1c4s1	n3x1c4s2	n4x2c1s1	n4x2c1s2
n4x2c2s1	n4x2c2s2	n4x2c3s1	n4x2c3s2
n4x2c4s1	n4x2c4s2	n5x3c1s1	n5x3c1s2
n5x3c2s1	n5x3c2s2	n5x3c3s1	n5x3c3s2
n5x3c4s1	n5x3c4s2	n6	n7

24 cazuri de testare

7. Creează date de test

Intrari				**Rezultat afișat (expected)**
n	**x**	**c**	**s**	
-5				
0				
21				
25				
1	a	a	y	
1	a	a	n	
1	a	b	y	
1	a	b	n	
5	abcde	a	y	
5	abcde	a	n	
5	abcde	c	y	
5	abcde	c	n	
5	abcde	e	y	
5	abcde	e	n	
5	abcde	f	y	
5	abcde	f	n	

20	abcdefghijklmnoprstu	a	y	
20	abcdefghijklmnoprstu	a	n	
20	abcdefghijklmnoprstu	c	y	
20	abcdefghijklmnoprstu	c	n	
20	abcdefghijklmnoprstu	u	y	
20	abcdefghijklmnoprstu	u	n	
20	abcdefghijklmnoprstu	z	y	
20	abcdefghijklmnoprstu	z	n	

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void categoryPartitioning () {
//…
}
```

Notă: O altă categorie poate fi considerată numarul de apariții a lui c în șirul x. Această categorie poate fi adaugată celor existente.

Avantaje și dezavantaje

- Pașii de inceput (identificarea parametrilor și a condițiilor de mediu precum și a categoriilor) nu sunt bine definiți și se bazează pe experienta celui care face testarea. Pe de altă parte, odata ce acesti pași au fost trecuti, aplicarea metodei este foarte clară.
- Este mai clar definită decat metodele funcționale anterioare și poate produce date de testare mai cuprinzatoare, care testează funcționalitati suplimentare; pe de altă parte, datorită exploziei combinatorice, pot rezulta date de test de foarte mare dimensiune.

Cuprins

Metoda Grafului Cauză-Efect (Cause-Effect Graphing)

- Partiționarea în categorii poate produce un număr mare de combinații de intrări, dintre care o mare parte poate fi nefezabilă.
- Metoda grafului cauză-efect (cunoscuta și ca modelarea dependențelor) se concentrează pe modelarea relațiilor de dependență între condițiile de intrare ale programului (cauze) și condițiile de ieșire (efecte).
- Relația dintre acestea este exprimată sub forma unui graf cauză-efect.
- Graful cauză-efect = reprezentare vizuală a relației logice dintre cauze și efecte, exprimabilă ca o expresie Booleană.

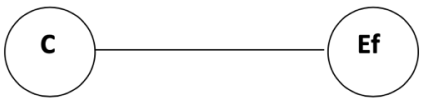
Cauză = orice condiție în specificație (cerințe) care poate afecta răspunsul programului.

Efect = răspunsul programului la o combinație de condiții de intrare. Efectul nu este în mod necesar o ieșire (poate fi un mesaj de eroare, un display, o modificare a unei baze de date sau chiar un punct de testare intern)

Notație:

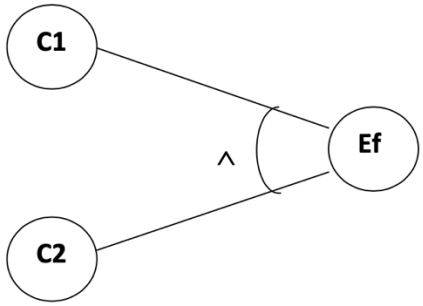
Relații Cauză-Efect

Implies : if C then Ef

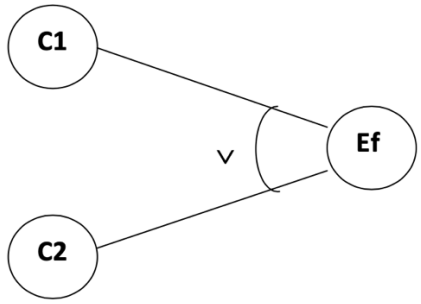


Not – Implies : if (¬C) then Ef

And – Implies: if (C1 && C2) then Ef

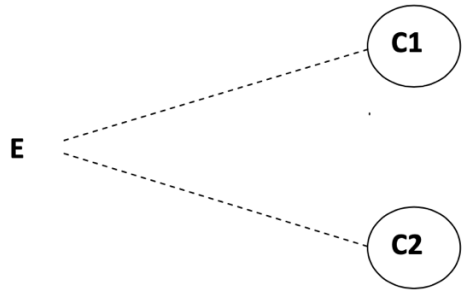


Or – Implies: if (C1 || C2) then Ef

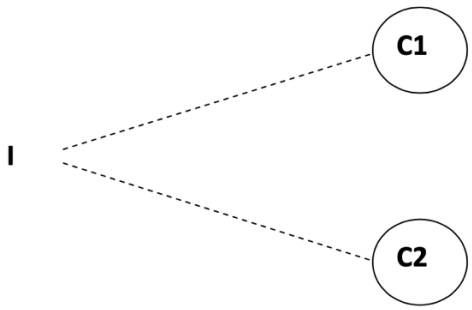


Constrângeri între cauze

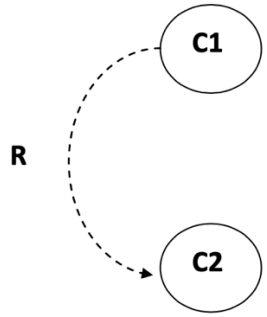
E (exclusive): fie C1 sau C2 (cel mult unul dintre ele)



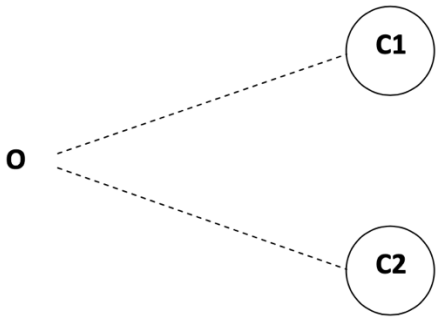
I (Inclusive): cel puțin C1 sau C2



R (Requires) C1 cere C2 (dacă C1 atunci C2)

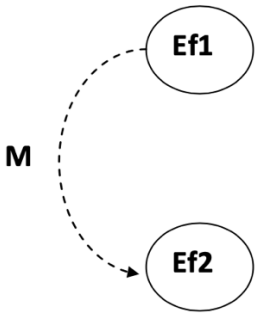


O (one and only one): unul și numai unul dintre C1 și C2



Constrângeri între efecte

M (Masks): Ef1 maschează Ef2 (dacă Ef1 atunci ¬Ef2)



Crearea grafului cauză-efect (exemplu)

după Aditya P. Mathur, Foundations of Software Testing, Pearson Education 2008.

Cerințe: O companie vinde pe web calculatoare (CPU1, CPU2, CPU3), imprimante (PR1, PR2), monitoare (M20, M23, M30) și memorie adițională (RAM256, RAM512, RAM1G). O comandă cuprinde între 1 și 4 articole, cel mult câte unul dintre cele 4 categorii amintite. Interfața grafică constă în 4 ferestre (pentru cele 4 categorii de produse) și o fereastră în care sunt afișate articolele primite cadou.

Monitoarele M20 și M23 pot fi cumpărate cu oricare CPU sau singure. M30 poate fi cumpărat doar împreună cu CPU3. PR1 este oferită cadou la cumpărarea lui CPU2 sau CPU3. Monitoarele și imprimantele, în afară de M30, pot fi cumpărate separat, fără a cumpăra și CPU. La cumpărarea unui CPU1 se primește RAM256 upgrade, iar la cumpărarea unui CPU2 sau CPU3 se primește RAM512 upgrade. La cumpărarea unui CPU3 și a unui M30 se primește RAM1G upgrade și PR2 cadou.

Construirea grafului cauză-efect

C1: Cumpărare CPU1

C2: Cumpărare CPU2

C3: Cumpărare CPU3

C4: Cumpărare PR1

C5: Cumpărare PR2

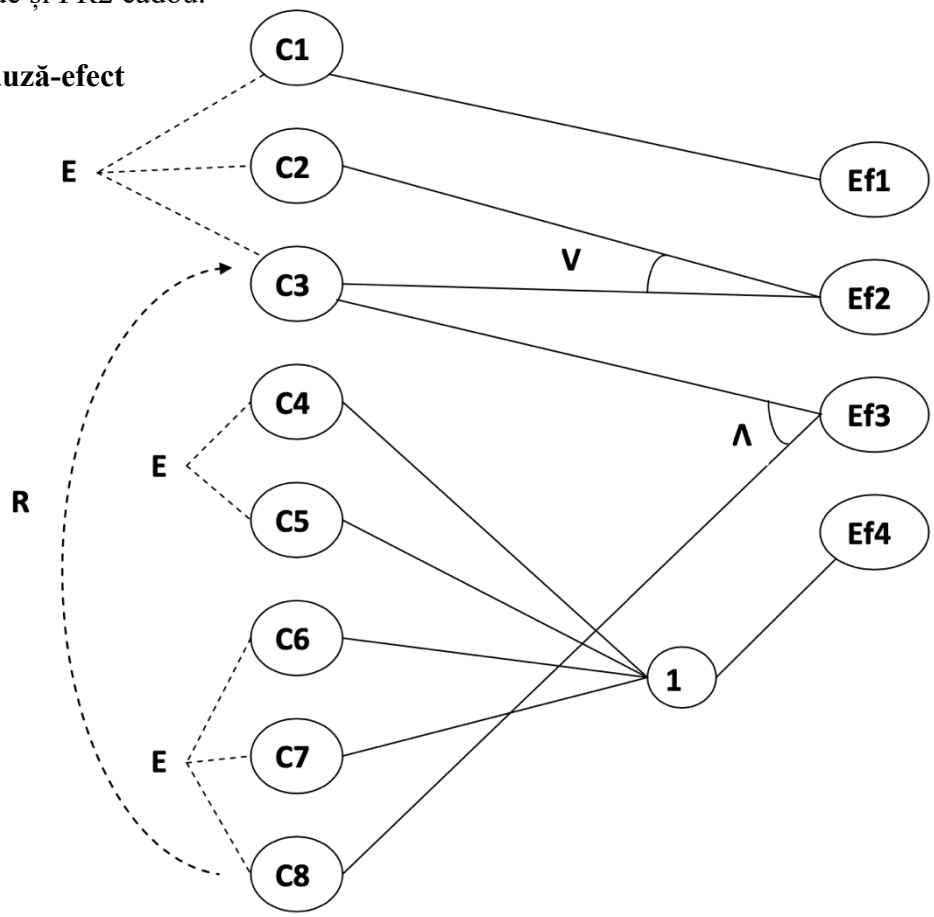
C6: Cumpărare M20

C7: Cumpărare M23

C8: Cumpărare M30

Ef1: RAM256

Ef2: RAM512 și PR1



(pot fi considerate efecte separare, dar se complică fără motiv graful)

Ef3: RAM1G și PR2

Ef4: nici un cadou

Crearea tabelului de decizie din graful cauză-efect

Input: Un graf cauză-efect avand cauze C1, ..., Cp și efecte Ef1, ..., Efq.

Output: Un tabel de decizie avand N = p + q randuri și M coloane, unde M depinde de relația dintre cauze și efect

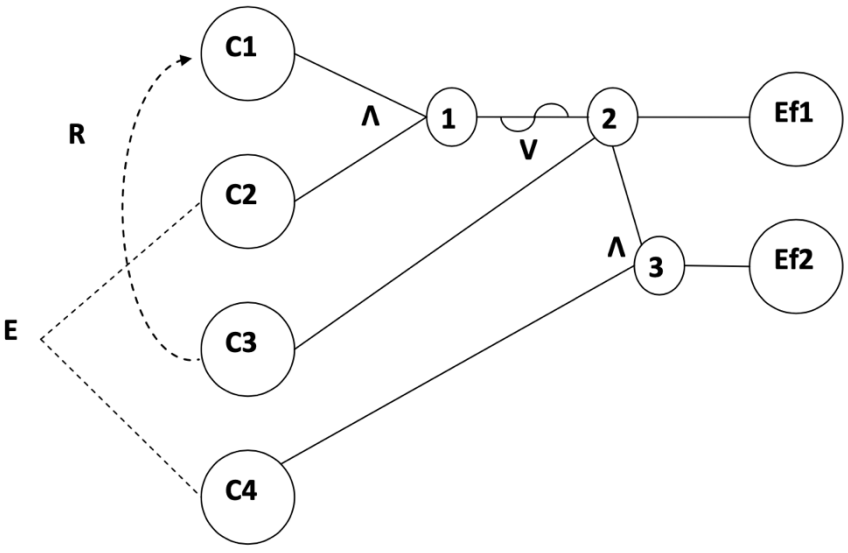
Procedura de creare a tabelului de decizie

1. Inițializează nr_coloane = 0 (tabel de decizie gol)
2. For i = 1 to q
   - 2.1. e = Ef_i (selectează următorul efect pentru procesare)
   - 2.2. Găsește combinațiile de condiții care produc apariția efectului e.

     Fie V1, ..., Vm_i aceste combinații, $m_i > 0$. Setează Vk(j), $p < j \leq p+q$, la 1 dacă efectul Ef_j apare ca urmare a combinației respective și la 0 în caz contrar
   - 2.3. Actualizează tabelul de decizie. Adaugă V1, ..., Vm_i la tabel pe post de coloane succesive începand cu poziția nr_coloane + 1.
   - 2.4. nr_coloane = nr_coloane + m_i.

Nr coloane rezultate este M = nr_coloane.

Exemplu



Pas 1: nr_coloane = 0

Pas 2: i = 1

Pas 2.1: e = Ef1

Pas 2.2:

Se caută valorile lui C1, C2, C3 astfel încât

¬ (C1 ∧ C2) ∨ C3 = 1

1	0	1
0	1	1
0	0	1
1	1	1
1	0	0
0	1	0
0	0	0

În plus, se aplică contrângerea C3 implică C1

1	0	1
1	1	1
1	0	0
0	1	0
0	0	0

Se adaugă C4 = 0 și valorile corespunzătoare pentru Ef1 și Ef2

V1	1	0	1	0	1	0
V2	1	1	1	0	1	0
V3	1	0	0	0	1	0
V4	0	1	0	0	1	0
V5	0	0	0	0	1	0

Pas 2.3: Matricea obținută este transpusă și adaugată la tabelul de decizie începând cu poziția nr_coloane + 1 = 1

	1	**2**	**3**	**4**	**5**
C1	1	1	1	0	0
C2	0	1	0	1	0
C3	1	1	0	0	0
C4	0	0	0	0	0
Ef1	1	1	1	1	1
Ef2	0	0	0	0	0

Pas 2.4: Se actualizează nr_coloane = 0 + 5 = 5

Pas 2: i = 2

Pas 2.1: e = Ef2

Pas 2.2:

Se cauta valorile lui C1, C2, C3, C4 astfel incat

$(\neg (C1 \wedge C2) \vee C3) \wedge C4 = 1$

Folosind combinațiile C2, C2, C3 anterioare pentru Ef1, obținem

1	0	1	1
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	1

În plus, se aplică constrângerea ca C2 și C4 să nu existe simultan

1	0	1	1
1	0	0	1
0	0	0	1

Se adaugă valorile corespunzătoare pentru Ef1 și Ef2

V1	1	0	1	1	1	1
V2	1	0	0	1	1	1
V3	0	0	0	1	1	1

Pas 2.3: Matricea obținută este transpusă și adaugată la tabelul de decizie începand cu poziția nr_coloane + 1 = 6

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C1	1	1	1	0	0	1	1	0
C2	0	1	0	1	0	0	0	0
C3	1	1	0	0	0	1	0	0
C4	0	0	0	0	0	1	1	1
Ef1	1	1	1	1	1	1	1	1
Ef2	0	0	0	0	0	1	1	1

Pas 2.4: Se actualizează nr_coloane = 5 + 3 = 8

Procedura se termină. Nr coloane rezultate este M = 8.

Generarea cazurilor de testare

Fiecare coloană din tabelul de decizie generează cel puțin un caz de testare, corespunzător combinației C1, ..., Cp respective.

Observație: C1, ..., Cp sunt în general expresii care folosesc variabile, etc., deci pentru o combinație pot fi selectate mai multe cazuri de testare.

Problemă: Explozie a stărilor datorită combinației de cauze.

Soluție: Limitarea numărului de cazuri de testare folosind euristici.

Cuprins

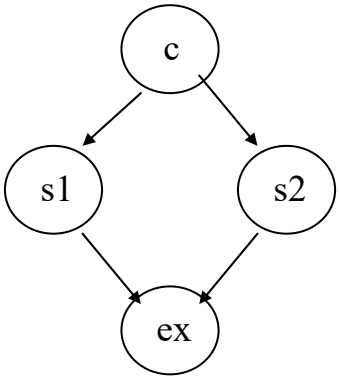
2. Testare structurală

- datele de test sunt generate pe baza implementării (programului), fără a lua în considerare specificația (cerințele) programului
- pentru a utiliza metode structurale de testare programul poate fi reprezentat sub forma unui graf orientat
- datele de test sunt alese astfel încât să parcurga toate elementele (instrucțiune, ramură sau cale) grafului măcar o singură dată. În funcție de tipul de element ales, vor fi definite diferite măsuri de acoperire a grafului: acoperire la nivel de instrucțiune, acoperire la nivel de ramură sau acoperire la nivel de cale

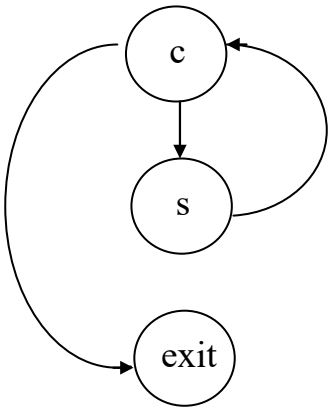
Transformarea programului într-un graf orientat

- Pentru o secvență de instrucțiuni se introduce un nod

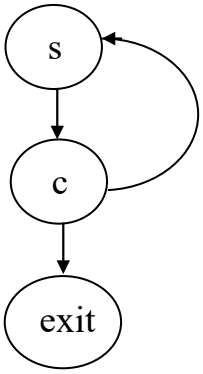
- if c then s1 else s2



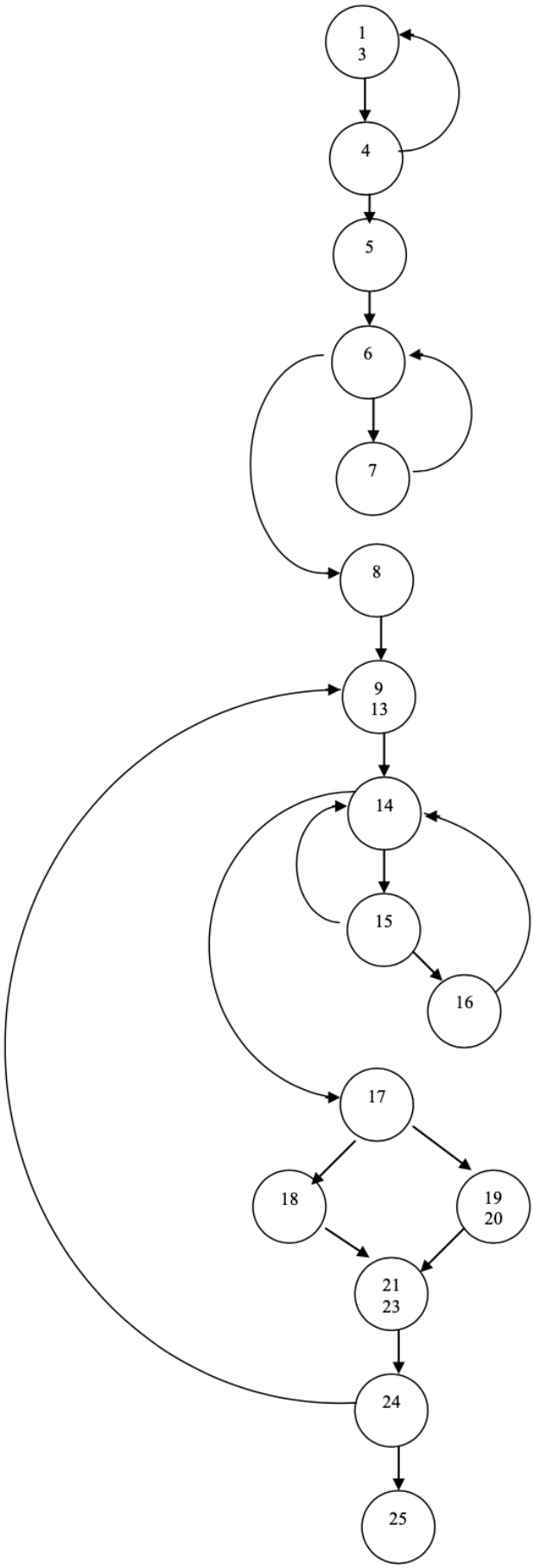
- while c do s



repeat s until c



```
// extras cod clasa de testat implementată în Java (obsolete)
    public class MyClass {
        public static void main(String[] arg) {

            KeyboardInput in = new KeyboardInput();
            char response,c,n1;
            boolean found;
            int n,i;
            char[]a = new char[20];
1           do {
2               System.out.println("Input an integer between 1 and
20: ");
3               n = in.readInteger();
4           } while (n < 1 || n > 20);
5           System.out.println("input "+n+" character(s)");
6           for(i=0;i<n;i++)
7               a[i] = in.readCharacter();
8           n1 = in.readCharacter();
9           do {
10              System.out.println("Input character to search for:
");
11              c = in.readCharacter();
12              n1 = in.readCharacter();
13              found = false;
14              for(i=0;!found && i<n;i++)
15                  if(a[i] == c)
16                      found = true;
17              if(found)
18                  System.out.println("character "+ c +" appears
at position "+i);
19              else
20                  System.out.println("character "+ c +" does not
appear in string");
21              System.out.println("Search for another
character?[y/n]: ");
22              response = in.readCharacter();
23              n1 = in.readCharacter();
24          } while((response == 'y')||(response == 'Y'));
25      }
    }
```


1
3
4
5
6
7
8
9
13
14
15
16
17
18
19
20
21
23
24
25

Pe baza grafului se pot defini diverse acoperiri:

- *Acoperire la nivel de instrucțiune*: fiecare instrucțiune (nod al grafului) este parcursă măcar o dată
- *Acoperire la nivel de ramură*: fiecare ramură a grafului este parcursă măcar o dată
- *Acoperire la nivel de condiție*: fiecare condiție individuală dintr-o decizie să ia atât valoarea adevărat cât și valoarea fals
- *Acoperire la nivel de cale*: fiecare cale din graf este parcursă măcar o dată

(a) Statement coverage (acoperire la nivel de instrucțiune)

Pentru a obține o acoperire la nivel de instrucțiune, trebuie să ne concentram asupra acelor instrucțiuni care sunt controlate de condiții (acestea corespund ramificațiilor din graf)

Intrari				**Rezultat afișat**	**Instrucțiuni parcurse**
N	**x**	**C**	**s**		
1	a	a	y		1..3, 4, 5, 6 7, 6, 8, 9..13 14, 15, 16, 14, 17, 18, 21..23 24, 9..13
		b	n		14, 15, 14, 17, 19..20, 21..23 24, 25

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void statementCoverage() {
//…
}
```

Testarea la nivel de instrucțiune este privita de obicei ca nivelul minim de acoperire pe care îl poate atinge testarea structurală. Acest lucru se datorează sentimentului că este absurd să dai în funcționare un software fără a executa măcar o dată fiecare instrucțiune.

Totuși, destul de frecvent, această acoperire nu poate fi obținută, din următoarele motive:

- Existenta unei porțiuni izolate de cod, care nu poate fi niciodată atinsă. Această situație indică o eroare de design și respectiva porțiune de cod trebuie înlăturată.
- Existența unor porțiuni de cod sau subrutine care nu se pot executa decat în situații speciale (subrutine de eroare, a căror execuție poate fi dificilă sau chiar periculoasă). În astfel de situații, acoperirea acestor instrucțiuni poate fi inlocuită de o inspecție riguroasă a codului

Avantaje:

- Realizează execuția măcar o singură dată a fiecarei instrucțiuni
- În general ușor de realizat

Slăbiciuni:

Nu asigură o acoperire suficientă, mai ales în ceea ce privește condițiile:

- Nu testează fiecare condiție în parte în cazul condițiilor compuse (de exemplu, pentru a se atinge o acoperire la nivel de instrucțiune în programul folosit ca exemplu, nu este necesară introducerea unei valori mai mici ca 1 pentru *n*)
- Nu testează fiecare ramură
- Probleme suplimentare apar în cazul instrucțiunilor *if* a căror clauza *else* lipsește. În acest caz, testarea la nivel de instrucțiune va forța execuția ramurii corespunzătoare valorii adevărat, dar, deoarece nu există clauza *else*, nu va fi necesară și execuția celeilalte ramuri. Metoda poate fi extinsă pentru a rezolva această problemă.

(b) Decision coverage (acoperire la nivel de decizie) sau branch coverage (acoperire la nivel de ramură)

- Este o extindere naturală a metodei precedente.
- Genereaza date de test care testează cazurile când fiecare decizie este adevărată sau falsă.

	Decizii
(1)	while (n<1||n>20)
(2)	for (i=0; i<n; i++)
(3)	for(i=0; !found && i<n; i++)
(4)	if(a[i]==c)
(5)	if(found)
(6)	while ((response=='y') ||(response=='Y'))

Intrari				**Rezultat afișat**	**Decizii acoperite**
N	**x**	**C**	**s**		
25				Cere introducerea unui întreg între 1 și 20	
1	a	A	y	Afișează poziția 1; se cere introducerea unui nou caracter	
		B	n	Caracterul nu apare	

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void branchCoverage() {
//…
}
```

Avantaje:

- Este privită ca etapa superioară a testarii la nivel de instrucțiune; testează toate ramurile (inclusiv ramurile nule ale instrucțiunilor *if/else*)

Dezavantaje:

- Nu testeaza condițiile individuale ale fiecărei decizii

(c) Condition coverage (acoperire la nivel de condiție)

- Genereaza date de test astfel încat fiecare condiție individuală dintr-o decizie să ia atât valoarea adevărat cât și valoarea fals (dacă acest lucru este posibil).

- De exemplu, dacă o decizie ia forma c1 || c2 sau c1 && c2, atunci acoperirea la nivel de condiție se obține astfel încât fiecare dintre condițiile individuale c1 și c2 sa ia atat valoarea adevărat cat și valoarea fals

Nota: decizie inseamna orice ramificare in graf, chiar atunci cand ea nu apare explicit in program. De exemplu, pentru constructia for i := 1 to n din Pascal conditia implicita este i <= n

Decizii	Conditii individuale
while (n<1||n>20)	n < 1, n > 20
for (i=0; i<n; i++)	i < n
for(i=0; !found && i<n; i++)	found, i< n
if(a[i]==c)	a[i] = c
if(found)	Found
while ((response=='y') ||(response=='Y'))	(response=='y') (response=='Y')

Intrari				**Rezultat afișat**	**Conditii individuale acoperite**
n	**x**	**C**	**s**		
0				Cere introducerea unui întreg între 1 și 20	
25				Cere introducerea unui întreg între 1 și 20	
1	a	A	y	Afișează pozitia 1; se cere introducerea unui nou caracter	
		B	Y	Caracterul nu apare; se cere introducerea unui nou caracter	

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void conditionCoverage () {
//…
}
```

Avantaje

- Se concentreaza asupra condițiilor individuale

Dezavantaj

- Poate să nu realizeze o acoperire la nivel de ramură. De exemplu, datele de mai sus nu realizează ieșirea din bucla *while* ((response=='y') || (response=='Y')) (condiția globală este în ambele cazuri adevărată). Pentru a rezolva această slăbiciune se poate folosi testarea la nivel de decizie / condiție.

(d) Condition/decision coverage (acoperire la nivel de condiție/decizie)

- Generează date de test astfel încât fiecare condiție individuală dintr-o decizie să ia atat valoarea adevărat cât și valoarea fals (dacă acest lucru este posibil) și fiecare decizie să ia atât valoarea adevărat cât și valoarea fals.

Intrări				**Rezultat afișat**	**Condiții/Decizii individuale acoperite**
n	**x**	**C**	**s**		
0				Cere introducerea unui întreg între 1 și 20	
25				Cere introducerea unui întreg între 1 și 20	
1	a	A	y	Afișează poziția 1; se cere introducerea unui nou caracter	
		B	Y	Caracterul nu apare; se cere introducerea unui nou caracter	
		B	n	Caracterul nu apare	

(e) Multiple condition coverage (acoperire la nivel de condiții multiple)

Generează date de test astfel încât să parcurga toate combinațiile posibile de adevărat și fals ale conditiilor individuale.

(f) Modified condition/decision (MC/DC) coverage

- Condition/Decision coverage poate să nu testeze unele condiții individuale (care sunt "mascate" de alte condiții)
- Multiple condition coverage poate genera o explozie combinatorică (pentru n condiții pot fi necesare 2^n teste)

Solutie: O formă modificată a condition/decision coverage.
Un set de teste satisface MC/DC coverage atunci când:

- Fiecare condiție individuală dintr-o decizie ia atât valoare True cât și valoare False
- Fiecare decizie ia atât valoare True cât și valoare False
- Fiecare condiție individuală influențează în mod independent decizia din care face parte

Avantaje:

- Acoperire mai puternică decât acoperirea condiție/decizie simplă, testând și influența condițiilor individuale asupra deciziilor
- Produce teste mai puține – depinde liniar de numărul de condiții

AND

Test	**C1**	**C2**	**C1 ∧ C2**
t1	True	True	True
t2	True	False	False
t3	False	True	False

t1 și t3 acoperă C1

t1 și t2 acoperă C2

OR

Test	**C1**	**C2**	**C1 ∨ C2**
t1	False	True	True
t2	True	False	True
t3	False	False	False

t2 și t3 acoperă C1

t1 și t3 acoperă C2

XOR

Test	**C1**	**C2**	**C1 xor C2**
t1	True	True	False
t2	True	False	True
t3	False	False	False

t2 și t3 acoperă C1

t1 și t2 acoperă C2

Exemplu: C = C1 ∧ C2 ∨ C3

Test	**C1**	**C2**	**C3**	**C**	**Efect demonstrat pentru**
1	True	True	False	True	C1
2	False	True	False	False	
3	True	True	False	True	C2
4	True	False	False	False	
5	True	False	True	True	C3
6	True	False	False	False	

Set de teste minimal

Test	**C1**	**C2**	**C3**	**C**
t1	True	True	False	True
t2	False	True	False	False
t3	True	False	False	False
t4	True	False	True	True

t1 și t2 testeaza C1

t1 și t3 testeaza C2

t3 și t4 testeaza C3

(g) Testarea circuitelor independente

Acesta este o modalitate de a identifica limita superioară pentru numărul de căi necesare pentru obținerea unei acoperiri la nivel de ramură.

Se bazează pe formula lui McCabe pentru Complexitate Ciclomatică:

Dat fiind un graf complet conectat *G* cu *e* arce și *n* noduri, atunci numărul de circuite linear independente este dat de:

$$V(G) = e - n + 1$$

La https://en.wikipedia.org/wiki/Cyclomatic_complexity puteți vedea mai multe formule.

i. V(G) = e - n + 2p, unde

e = numărul de muchii ale graficului.

n = numărul de noduri ale graficului.

p = numărul de componente conectate.

Formulă alternativă, unde fiecare punct de ieșire este conectat înapoi la punctul de intrare (adică graful e complet conectat):

ii. V(G) = e − n + p.

Pentru un singur program (sau subrutină sau metodă), p este întotdeauna egal cu 1.

Deci, o formulă mai simplă pentru o singură subrutină este:

iii. V(G) = e − n + 2.

Terminologie:

- Graf complet conectat: există o cale între oricare 2 noduri
  (există un arc între nodul de stop și cel de start)
- Circuit = cale care începe și se termină în același nod
- Circuite linear independente: nici unul nu poate fi obținut ca o combinație a celorlalte

În exemplul nostru, adăugând un arc de la 25 la 1, avem:

n = 16, e = 22, V(G) = 7

Circuite independente:

a) 1..3, 4, 5, 6, 8, 9..13, 14, 17, 18, 21..23, 24, 25, 1..3
b) 1..3, 4, 5, 6, 8, 9..13, 14, 17, 19..20, 21..23, 24, 25, 1..3
c) 1..3, 4, 1..3
d) 6, 7, 6
e) 14, 15, 14
f) 14, 15, 16, 14
g) 9..13, 14, 17, 18, 21..23, 24, 25, 1

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void circuitCoverage() {
//…
}
```

Acesta poartă numele de set de bază.

Orice cale se poate forma ca o combinație din acest set de bază.

De exemplu

1..3, 4, 1..3, 4, 1..3, 4, 5, 6, 7, 6, 8, 9..13, 14, 15, 16, 14 17, 18,

21..23, 24, 25, 1..3

este o combinație din: a, c (de 2 ori), d, f

Avantaje:

Setul de bază poate fi generat automat și poate fi folosit pentru a realiza o acoperire la nivel de ramură.

Dezavantaje:

Setul de bază nu este unic, iar uneori complexitatea acestuia poate fi redusă.

(h) Testare la nivel de cale

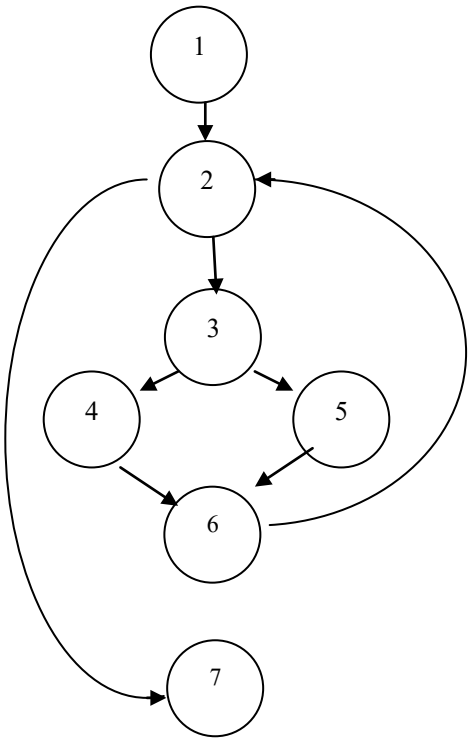
- Generează date pentru executarea fiecarei căi măcar o singura dată
- Problemă: în majoritatea situațiilor există un număr infinit (foarte mare) de căi
- Soluție: Împărțirea căilor în clase de echivalență.

De exemplu: 2 clase pot fi considerate echivalente dacă diferă doar prin numărul de ori de care sunt traversate de același circuit; determină 2 clase de echivalență: traversate de 0 ori și n ori, n > 1.

Aplicație:

Dacă un program este structurat, atunci, folosindu-se o tehnica descrisă de Paige și Holthouse (1977), acesta poate fi caracterizat de o expresie regulata formata din nodurile grafului.

```
1       …
2       while c1 do begin
3       if c2 then
4                       …
5               else …
6       end
7       …
```



Expresia regulată obținută este: 1.2.(3.(4+5).6.2)*.7

Notă: operatorul star (Kleene): R* = R de zero sau mai multe ori la rând (să spunem n)

Pentru n = 0 și n = 1 avem:

1.2.(3.(4+5).6.2 + null).7

de unde rezultă căile:

1.2.7

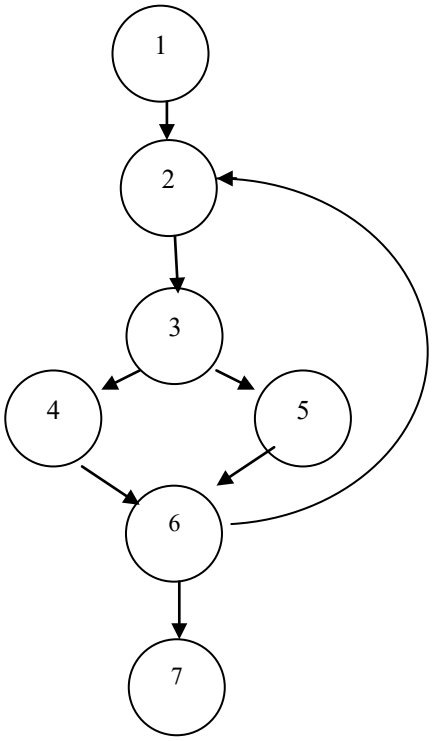
1.2.3.4.6.2.7

1.2.3.5.6.2

Numărul de căi se obține înlocuind în expresia regulată fiecare nod cu 1 (inclusiv pt. null), iar operația de concatenare devine înmulțire:

1.1.(1.(1+1).1.1 + 1).1 = 3 căi

```
1       …
2       repeat s
3               if c2 then
4                       …
5               else …
6       until c1
7       …
```



Expresia regulată: 1.2.3.(4+5).6.(2.3.(4+5).6)*.7

Pentru n = 0 și n = 1 avem:

1.2.3.(4+5).6.(2.3.(4+5).6 + null).7

de unde rezultă căile:

1.2.3.4.6.7

1.2.3.5.6.7

1.2.3.4.6.2.3.4.6.7

1.2.3.4.6.2.3.5.6.7

1.2.3.5.6.2.3.4.6.7

1.2.3.5.6.2.3.5.6.7

Numărul de căi se obține înlocuind în expresia regulată fiecare nod cu 1 (inclusiv pt. null), iar operația de concatenare devine înmulțire:

1.1.1.(1+1).1.(1.1.(1+1).1+1).1 = 2.3 = 6 căi

Pentru exemplul nostru:

1.4.(1.4)*.5.6.(7.6)*.8.9.14.(15.(null+16).14)*17.(18+19).21.24.(
9.14.(15.(null+16).14)*17.(18+19).21.24)*.25

număr de căi:
2.2.3.2.(3.2+1) = 168

Observație: multe căi, din care o mare parte sunt nefezabile (de exemplu, ieșirea de la început din cele două instrucțiuni *for*).

Avantaje:

- Sunt selectate căi pe care alte metode de testare structurală (inclusiv testare la nivel de ramură) nu le ating.

Dezavantaje:

- Multe căi, din care o parte pot fi nefezabile
- Nu exersează condițiile individuale ale deciziilor
- Tehnica descrisă pentru generarea căilor nu este aplicabilă direct programelor nestructurate.

Structural Testing (2)

Linear Code Sequence and Jump (LCSAJ) Coverage

Un LCSAJ este o cale (execuție) a unui program formată dintr-o secvență de cod (Linear Code Sequence) urmată de un salt (Jump) al controlului programului.

Un LCSAJ este definit ca un triplet (X, Y, Z), unde

- X este startul secvenței lineare
- Y este sfarsitul secvenței liniare
- Z este linia de cod unde este transferat controlul dupa sfârșitul secvenței liniare

Exemplu:

(calculul x^y, $x > 0$, $y \geq 0$)

```
1  begin
2      int x, y, z;
3      read(x, y);
4      z = 1;
5      while (y > 0) {
6          z = z*x;
7          y = y -1;
8      }
9      write(z);
10 end
```

LCSAJ	Start	End	Jump to
1	1	8	5
2	5	8	5
3	5	5	9
4	1	5	9
5	9	9	Exit

Considerăm T = {t1, t2}, unde t1 = (x = 3, y = 0), t2 = (x = 3, y = 2)

t1: (1, 5, 9) → (9, 9, exit)

t2: (1, 8, 5) → (5, 8, 5) → (5, 5, 9) → (9, 9, exit)

T acoperă toate cele 5 LCSAJ

Un set de teste care realizează o acoperire la nivel de decizie nu realizează în mod necesar o acoperire la nivel de LCSAJ

Exemplu:

```
1  begin
2      int x, y, z;
3      read(x, y);
4      p = 0;
5      if (x < 0)
6          p = x;
7      if (y < 0)
8          p = p + 1;
9      else
10         p = p + 2;
11 end
```

LCSAJ	**Start**	**End**	**Jump to**
1	1	9	exit
2	1	5	7
3	7	9	exit
4	7	7	10
5	1	7	10
6	10	10	Exit

Considerăm T = {t1, t2}, unde t1 = (x = -1, y = -1), t2 = (x = 0, y = 0)

Pentru t1 ambele condiții sunt satisfăcute, t1: (1, 9, exit)

Pentru t2 ambele condiții sunt false, t2 : (1, 5, 7) → (7, 7, 10) → (10, 10, exit)

Cele două LCSAJ rămase pot fi parcurse de t3 = (x = -1, y = 0) si t4 = (x = 0, y = -1),

t3: (1, 7, 10) → (10, 10, exit)

t4: (1, 5, 7) → (7, 9, exit)

Cuprins

Mutation testing (mutation analysis)

Tehnica de evaluare a unui set de teste pentru un program (având un set de teste generat, putem evalua cât de eficient este, pe baza rezultatelor obținute de acest test asupra mutanților programului)

Mutation = modificare foarte mică (din punct de vedere sintactic) a unui program

Pentru un program P, un mutant M al lui P este un program obținut modificând foarte ușor P; M trebuie să fie corect din punct de vedere sintactic.

Program P

```
begin
      int x, y;
      read(x, y);
      if (x > 0)
            write(x+y)
      else
            write(x*y)
end
```

Mutant M1

```
begin
      int x, y;
      read(x, y);
      if (x >= 0)
            write(x+y)
      else
            write(x*y)
end
```

Mutant M2

```
begin
      int x, y;
      read(x, y);
      if (x > 0)
            write(x-y)
      else
            write(x*y)
end
```

Mutant M3

```
begin
      int x, y;
      read(x, y);
      if (x > 0)
            write(x+y+1)
      else
            write(x*y)
```

```
end
```

Tehnica Mutation testing:

- Generarea mutanților pentru programul P (folosind o mulțime de operatori de mutație)
- Rularea setului de teste asupra programului P și asupra setului de mutanți; dacă un test distinge între P și un mutant M spunem că P omoară mutantul M.

Mutanți de primul ordin / mutanți de ordin mai mare (first-order/higher-order mutants)

First-order mutants = mutanți obținuți făcând o singură modificare în program

n-order mutants = mutanți obținuți făcând n modificari în program

n-order mutant = first-order mutant of a (n-1)-order mutant, n > 1

n-order mutant, n > 1, sunt numiți higher-order mutants

Mutant de ordin 2

```
begin
      int x, y;
      read(x, y);
      if (x >= 0)
            write(x-y)
      else
            write(x*y)
end
```

În general, în practică sunt folosiți doar mutanții de ordin 1. Motive:

- Numarul mare de mutanți de ordin 2 sau mai mare
- Coupling-effect

Principiile de bază ale mutation testing

- Competent programmer hypothesis (CPH):

Pentru o problemă dată, programatorul va scrie un program care se află în vecinatatea unui program care rezolvă în mod corect problema (și deci, erorile vor fi detectate folosind mutanți de ordinul 1)

- Coupling effect

Datele de test care disting orice program care diferă cu puțin de programul corect sunt suficient de puternice pentru a distinge erori mai complexe.

Rezultate experimentale arată că un set de teste care distinge un program de mutanții săi de ordin 1 este foarte aproape de a distinge programul de mutanții de ordin 2

Explicație intuitivă: în general erorile simple sunt mai greu de detectat. Erorile complexe pot fi detectate de aproape orice test

Strong mutation/ weak mutation

Un test t omoară mutantul M (distinge M față de P) dacă cele două se comportă diferit pentru testul t. Întrebare: când observam comportamentul celor două programe ?

- Testul t aduce pe P și M în stari diferite - se observă starea programului (valorile variabilelor afectate) dupa execuția instrucțiunii mutate.
- Schimbarea stării se propagă la sfarșitul programului - se observă valorile variabilelor returnate și alte efecte (schimbarea variabilelor globale, fișiere, baza de date), imediat dupa terminarea programului

Weak mutation: prima condiție este satisfăcută

Strong mutation: ambele condiții sunt satisfăcute

Program P

```
begin
      int x, y;
      read(x, y);
      y := y+1;
      if (x > 0)
```

```
            write(x)
      else
            write(y)
end
```

Mutant M

```
begin
      int x, y;
      read(x, y);
      y := y-1;
      if (x > 0)
            write(x)
      else
            write(y)
end
```

Testul (1, 1) distinge între P și M d.p.d.v. weak mutation, dar nu distinge între P si M d.p.d.v. strong mutation

Testul (0, 1) distinge între P și M d.p.d.v. strong mutation

Strong mutation: mai puternică. Se asigură că testul t detectează cu adevarat problema

Weak mutation: necesită mai puțina putere de calcul; strâns legată de ideea de acoperire

Mutanți echivalenți

Un mutant M a lui P se numește echivalent dacă el se comportă identic cu programul P pentru *orice* date de intrare. Altfel, se spune că M poate fi distins de P.

Din punct de vedere teoretic: în general, problema determinării dacă un mutant este echivalent cu programul părinte este nedecidabilă (este echivalentă cu halting problem)

În practică: determinarea echivalenței se face prin analiza codului

Determinarea mutanților echivalenți poate fi un proces foarte complex – principala problemă practică a tehnicii mutation testing (avem nevoie să decidem dacă mutanții sunt sau nu echivalenți pentru a putea evalua eficiența testelor)

Utilitatea mutation testing

- Evaluarea unui set de date existent (și construirea de noi teste, dacă testele existente nu omoară toți mutanții)
- Detectarea unor erori în cod

Evaluarea unui set de date existent (exemplu)

Program P

```
begin
      int x, y;
      read(x, y);
      if (x > 0)
            write(x+y)
      else
            write(x*y)
end
```

Considerăm următorii operatori de mutație:

\+ inlocuit de –

* inlocuit de /

o variabilă sau o constantă x este înlocuită de x+1

```
begin
      int x, y;
      read(x, y);
      if (x > 0)            M1 if (x+1 > 0)
                            M2 if (x> 0+1)
            write(x+y)      M3 write(x+1+y)
```

```
                              M4 write(x+y+1)

                              M5 write(x-y)

      else

            write(x*y)        M6 write((x+1)*y)

                              M7 write(x*(y+1))

                              M8 write(x/y)

end
```

Set de teste T = {t1, t2, t3, t4} t1 = (0, 0), t2 = (0, 1), t3 = (1, 0), t4 = (-1, -1)

P	**t1**	**t2**	**t3**	**t4**	**Mutant distins**
P(t)	0	0	1	1	
M1(t)	0	**1**	NE	NE	Y
M2(t)	0	0	**0**	NE	Y
M3(t)	0	0	**2**	NE	Y
M4(t)	0	0	**2**	NE	Y
M5(t)	0	0	1	1	**N**
M6(t)	0	1	NE	NE	Y
M7(t)	0	0	1	**0**	Y
M8(t)	ND	NE	NE	NE	Y

Mutanți nedistinși (alive) = {M5}

Întrebare: Este M5 mutant echivalent ?

Răspuns: Nu. (1, 1) distinge între P și M5

Mutation score: MS(T) = D/(L+D), unde

D – numărul de mutanți distinși

L – numărul de mutanți nedistinși (live mutants) neechivalenți

Pentru exemplu: MS(T) = 7/8

Detectarea erorilor folosind mutația (exemplu)

Program P

```
begin
      int x, y
      read(x)
      y = 1
      y = 2            Eroare: instructiune lipsa
      if (x < 0)
            y = 3
      if (x > 2)
            y = 4
      write(y)
end
```

Mutant M

```
...
      if (x < 1)
            y = 3
...
```

Arătăm că mutantul M generează teste care detectează eroarea.

Pentru ca un test t să distingă între P și M trebuie ca:

- Reachability: Instrucțiunea mutată să fie executată la aplicarea lui t
- State infection: Instrucțiunea mutată să afecteze starea programului
- State propagation: Schimbarea de stare să se propage în exterior

Pentru exemplul dat, pentru ca un test t să distingă între P si M:

- Reachability: TRUE
- State infection: $(x < 1 \wedge \neg (x < 0))$
- State propagation: $\neg (x > 2)$

Condiția rezultată: $(x = 0) \wedge (x <= 2) \Leftrightarrow x = 0$

Pentru x = 0 programul corect întoarce 2 în timp ce programul greșit returnează 3

Operator de mutație

Operator de mutație = Regula care se aplică unui program pentru a crea mutanți (e.g. înlocuirea/adăugarea/ștergerea unor operanzi, ștergerea unor instrucțiuni, etc.)

Programul nou obținut trebuie să fie valid din punct de vedere sintactic

Operator de mutație din Java (MuJava //deprecated)

- Traditional mutation operators (method-level operators) – operatori aplicabili oricărui limbaj procedural
- Class mutation operator – operatori specifici paradigmei orientate pe obiect și sintaxei Java
    - Încapsulare
    - Moștenire
    - Polimorfism și dynamic binding
    - Suprascrierea metodelor
    - Java specific

Operatori traditionali

http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf

Operatori de clasă

http://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf

What is testing?

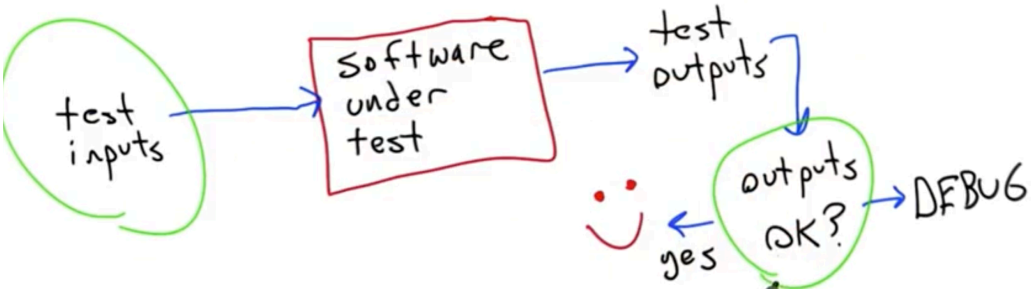
https://www.udacity.com/course/software-testing--cs258

1. Introduction

- In the big picture it's not necessarily failures that we're interested, but rather the fact that **if we can find failures in software and if we can fix these failures, then eventually we're going to run out of bugs and we'll have a software that actually works reliably**.
- Testing software is a large problem. We can see that:
  - Microsoft didn't manage to eliminate all the bugs in their products,
  - Google didn't manage to eliminate all the bugs in their services.
  - **How can we possibly get rid of all the bugs in our own software?**
- Testing problem is not really this large monolithic problem, but rather can be broken down into a lot of smaller sub-problems, and by looking at those sub-problems, **we can apply known techniques and things that people have done before**, and we could sort of pattern match on these problems, and once we've become good at these smaller problems, then we can become much better testers as a whole.

2. What is testing?

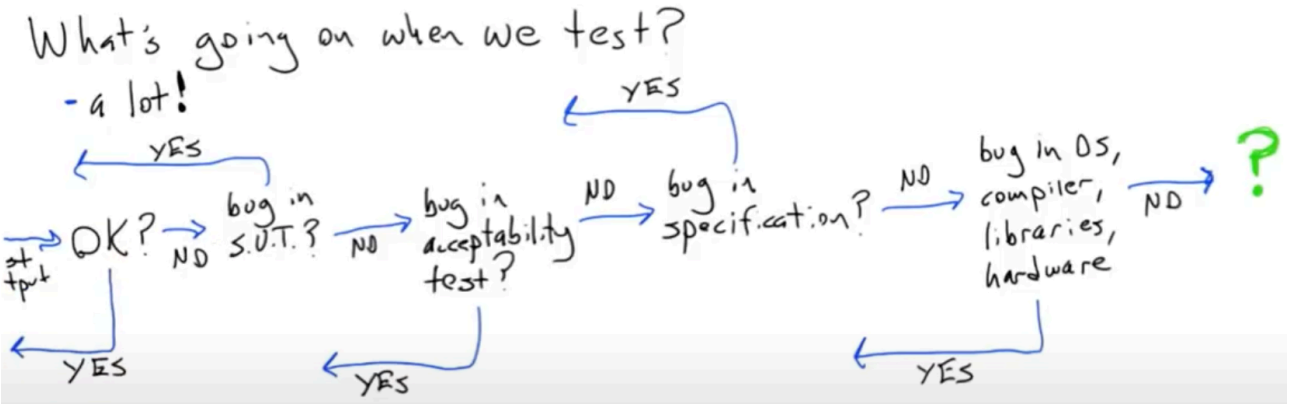
- It's always the case that we have some software under test (SUT).



- On the other hand, **selecting a good set of test inputs, and designing good acceptability test end up being actually really hard**, and basically, these are what we are going to be spending this course talking about.
- The goal of testing isn't so much as finding bugs, but rather it's finding bugs as early as possible. If our goal is just to find some bugs, we go ahead and give the software to our customers and let them find the bugs, but of course, there are huge cost associated with doing this.

- What we rather want to do is to move the time in which to find those bugs early. And the fundamental reason for that is, is that it's almost always the case that **the bug that's found earlier is cheaper to fix.**
- The second fact is that **it's possible to spend a lot of time and effort on testing and still do a really bad job.** Doing testing right requires some imagination and some good taste.
- Third, **more testing is not always better.** In fact, the quality of testing is all about the cost/benefit tradeoff. And fundamentally, testing is an economic activity. We're **spending money or** we're spending **effort** on testing in order to save ourselves money and effort later. Going along with this, testing methods should be evaluated about the cost per defect found.
- Fourth, **testing** can be made much **easier by designing software to be testable**.
- Fifth, **quality cannot be tested into software**. It is surprisingly easy to create software but it's impossible to test effectively at all.
- Finally, testing your own software is really hard.

3. What happens when we test software?



- We start off with the output produced by a test case. It's going to pass through the acceptability check.
- What we're doing when we check the output of a test case for acceptability is we're running a little experiment, and this experiment can have two possible results. One result is the output is okay. In which case, we are to go run another test case. And the question is, what do we learn in that case? And the answer is unfortunately not very much. What we might have done at best is increased our confidence just a tiny, tiny bit but the SUT is correct. And as it so happens, we stand to learn a whole lot more when the output is not okay. And the process I'm going to talk about right now is if the

acceptability check fails that is to say the test output is not okay, we have to discover what's happened. And so of course what we expect is much of the time we'll find a **bug in the SUT**. If that's the case, we're going to go fix it. And if not, there's still plenty of other possibilities: **a bug in** our **acceptability check,** or in our **specification**, in **OS, compiler, libraries, hardware**, etc.

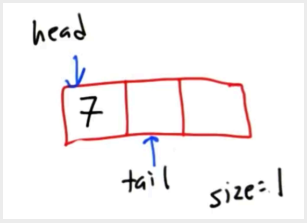
4. **Mars climate orbiter**

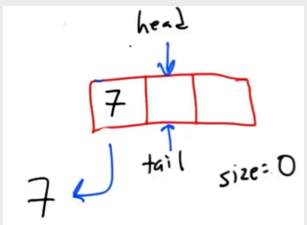
- **A famous bug** that happened to the Mars Climate Orbiter that was sent off to Mars in 1998, and there were **some miscommunications**, between NASA and the people they contracted out to which was Lockheed Martin. By the time Mars Climate Orbiter actually got to Mars which was in 1999, quite a while later, there had been some problems that caused the orbiter to drift off course enough that it basically ran into a suicide mission and crashed into the Martian atmosphere and broke up and crashed in the planet. What happened was **a basic unit error.**
- NASA expected units in metric, for example - meters per second, and Lockheed Martin programmed in English units, for example - feet per second.

5. **Fixed sized queue**

- enqueue
- dequeue
- FIFO order

```
# the Queue class provides a fized-size FIFO queue of
integers
# the constructor takes a single parameter: an integer >0
that
# is the maximum number of elements the queue can hold
# empty() returns True iff the queue holds no elements
# full() returns True iff the queue cannot hold any more
elements
# enqueue(i) attempts to put the integer i into the queue;
it returns
# True if successful and False if the queue is full
# dequeue() removes an integer from the queue and returns
it,
# or else returns None if the queue is empty


import array
import random

class Queue:
      def __init__(self,size_max):
            assert size_max > 0
            self.max = size_max
            self.head = 0
            self.tail = 0
            self.size = 0
            self.data = array.array('i', range(size_max))

      def empty(self):
            return self.size == 0

      def full(self):
            return self.size == self.max

      def enqueue(self, x):
            if self.size == self.max:
                  return False
            self.data[self.tail] = x
            self.size += 1
            self.tail += 1
            if self.tail == self.max:
                  self.tail = 0
            return True

      def dequeue(self):
            if self.size == 0:
                  return None
            x = self.data[self.head]
            self.size -= 1
            self.head += 1
            if self.head == self.max:
                  self.head = 0
            return x
```





6. Filling the queue

```
q = Queue(2)
c1 = enqueue(6)
c2 = enqueue(7)
c3 = enqueue(8)
c4 = dequeue()
c5 = dequeue()
c6 = dequeue()
```

The values in c1, c2, c3, c4, c5, c6 are:

True, True, False, 6, 7, None

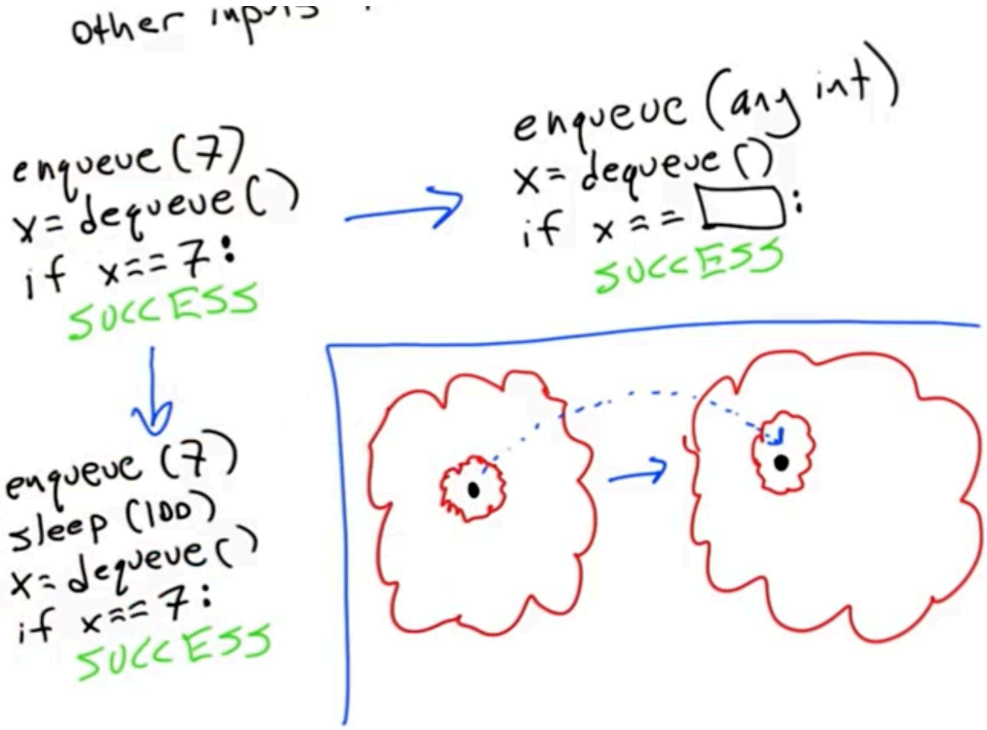
7. What we learn

```
enqueue(7)
x = dequeue()
if x == 7:
    print "Success!"
else:
    print "Error!"
```

If we pass this test case, what have we learned about our code?

- [x] Our code passes this test case.
- [ ] Our code passes any test case where we replace 7 with a different integer.
- [x] Our code passes many test cases where we replace 7 with a different integer.

8. Equivalent tests



- The question is given the above test case, is it possible to conclude without trying it that this test case will succeed?
- We're trying to make arguments that **a single test case is a representative of a whole class of actual executions of the system that we're testing**.
- The idea is we have a single test case that represents a point in the input space for the system that we're testing. By running the code, we mapped that point in the input space to a single point in the output space. But the problem is, there is of extremely large input space and we can't test it all.
- What we're trying to do here is build up an intuition for when we can make an argument that, in fact, we haven't made an argument about the mapping of a single input to a single output. But rather, **some class of inputs that are closely related, that are somehow equivalent for purposes of the SUT, and that if the system executes correctly for the single input that we've tried, it's going to execute correctly for all of the inputs in this particular bit of the input space**.

9. Testing the queue

- We can enqueue some numbers and dequeue them and check if they are correct.

- We can, also, write test functions that are not testing equivalent functionality in the queue but the if statements. For example, if the size is equal to max, then we will return false and if the tail is equal to max, then we will reset the tail to zero.
- Below is the code for 3 test functions for the fixed size queue mentioned before.

```
def test1():
    q = Queue(3)
    res = q.empty()
    if not res:
        print "test1 NOT OK"
        return
    res = q.enqueue(10)
    if not res:
        print "test1 NOT OK"
        return
    res = q.enqueue(11)
    if not res:
        print "test1 NOT OK"
        return
    x = q.dequeue()
    if x != 10:
        print "test1 NOT OK"
        return
    x = q.dequeue()
    if x != 11:
        print "test1 NOT OK"
        return
    res = q.empty()
    if not res:
        print "test1 NOT OK"
        return
    print "test1 OK"
```

```
def test2():
   q = Queue(2)
   res = q.empty()
   if not res:
      print "test2 NOT OK"
      return
   res = q.enqueue(1)
   if not res:
      print "test2 NOT OK"
      return
   res = q.enqueue(2)
   if not res:
      print "test2 NOT OK"
      return
   res = q.enqueue(3)
   if q.tail != 0:
      print "test2 NOT OK"
      return
print "test2 OK"

def test3():
   q = Queue(1)
   res = q.empty()
   if not res:
      print "test3 NOT OK"
      return
   x = q.dequeue()
      if not x is None:
      print "test3 NOT OK"
      return
   res = q.enqueue(1)
      if not res:
      print "test3 NOT OK"
      return
   x = q.dequeue()
      if x != 1 or q.head != 0:
      print "test3 NOT OK"
      return
print "test3 OK”

test1()
test2()
test3()
```

10. Creating testable software

- SUT:
  - clean code
  - refactor
  - should always be able to describe what a module does & how it interacts with other code
  - no extra threads
  - no swamp of global variables
  - no pointer soup
  - Modules should have unit tests
  - when applicable, support fault injection
  - assertions, assertions, assertions

11. Assertions

Assertion: Executable check for a property that must be true for your code

Rule 1: Assertions are not for error handling

```
def sqrt (arg):
      ... compute result ...
      assert result >= 0
      return result
```

Rule 2: NO SIDE EFFECTS

```
assert foo() == 0
where:
foo() changes a global variable
```

Rule 3: No silly assertions

```
assert (1 + 1) == 2
```

- The best assertions are those which check a nontrivial property that could be wrong but only if we actually made a mistake in our logic. It's not something that could be wrong if the user did something wrong, and it's not something that's wrong that’s just completely silly to check.

12. Checkrep

- Let’s see what assertions we can add to the queue code presented before that would make it more robust with respect to mistakes.

- We add a checkRep function that stands for check representation, and this is a function that we commonly add to a data structure or to other functions that checks the variables in the program for self-consistency. And so what it's going to do is basically try and terminate the program if some invariant that we know should hold over the program's data structures fails to hold.
- Next we are going to break our queue by making the enqueue fail to properly reset the tail variable when it overflows. What we have here is logic that when self.tail is equal to self.max, we reset it back to 0. And we are going to set it to 1 instead. What we want is to write a tighter set of assertions for our queue so that our checkRep can catch this before we actually return the wrong thing to the user.

```
def enqueue(self, x):
      if self.size == self.max:
            return False
      self.data[self.tail] = x
      self.size += 1
      self.tail += 1
      if self.tail == self.max:
            self.tail = 1 # code defect
      return True

def checkRep(self):
      assert self.size >= 0 and self.size <= self.max
      if self.tail > self.head:
            assert (self.tail - self.head) == self.size
      if self.tail < self.head:
            assert (self.head - self.tail) == (self.max - self.size)
      if self.tail == self.head:
            assert (self.size == 0) or (self.size == self.max)
       return
```

13. Why assertions?

- Make code self-checking, leading to more effective testing
- Make code fail early, closer to the bug
- Assign blame
- Document assumptions, preconditions, postconditions, & invariants

14. Are assertions used in production?

- The GCC source code base contains more than 9000 assertions.
- The LLVM compiler suite contains about 13,000 assertions, and that's over about 1.4 billion lines of code for a total of about 1 assertion per 110 lines of code.
- We are counting raw lines of code, not source lines of code. This includes blanks, comments, and everything.

- The LLVM and GCC developers have made a pretty serious commitment to checking assumptions, preconditions, post conditions, and invariant in the code that they wrote.
- Much of the time the bugs show up as assertion violations.

15. Disabling assertions

- "`Python -O`" disable assertions
- Advantages:
  - Code runs faster
  - Code keeps going

    What is it that we're trying to do with our system? Is it better to keep going or is it better to stop? Keeping going after some condition is true that will lead to an assertion violation may lead to a completely erroneous execution. On the other hand, possibly, that's better than actually stoping.
- Disadvantages:
  - What if our code relies on a side-effecting assertion?
  - Even in production code may be better to fail early

    Do our users want the system to die or do they want the system give them some completely wrong result?

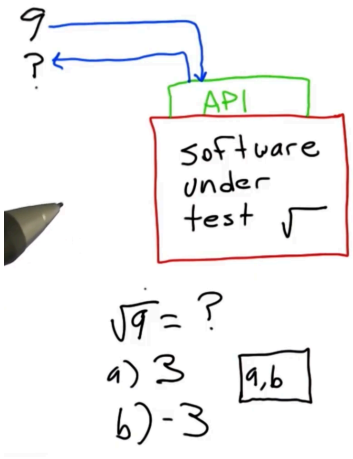
16. When to use assertions

- Disadvantages:
  - What if our code relies on a side-effecting assertion?
  - Even în production code, may be better to fail early
- **Julian Seward**, Valgrind:

  "This code is absolutely loaded with assertions, and these are permanently enabled … as Valgrind has become more widely used, they have shown their worth, pulling up various bugs which otherwise have appeared as hard-to-find segmentation faults.
  I am of the view that it's acceptable to spend 5% of the total running time … doing assertion checks."
- On the other hand… if you're doing something so critical that it keeps going that it resembles landing a spaceship on Mars, then go ahead and turn off assertions in your production software.

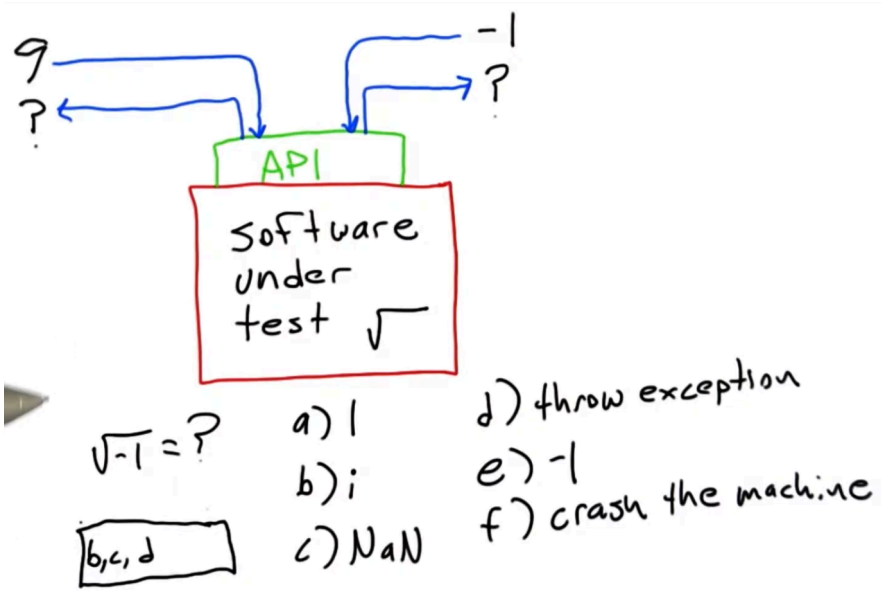
17. Specifications

- The SUT is providing some set of APIs, i.e. a set of function calls that can be called by another software.

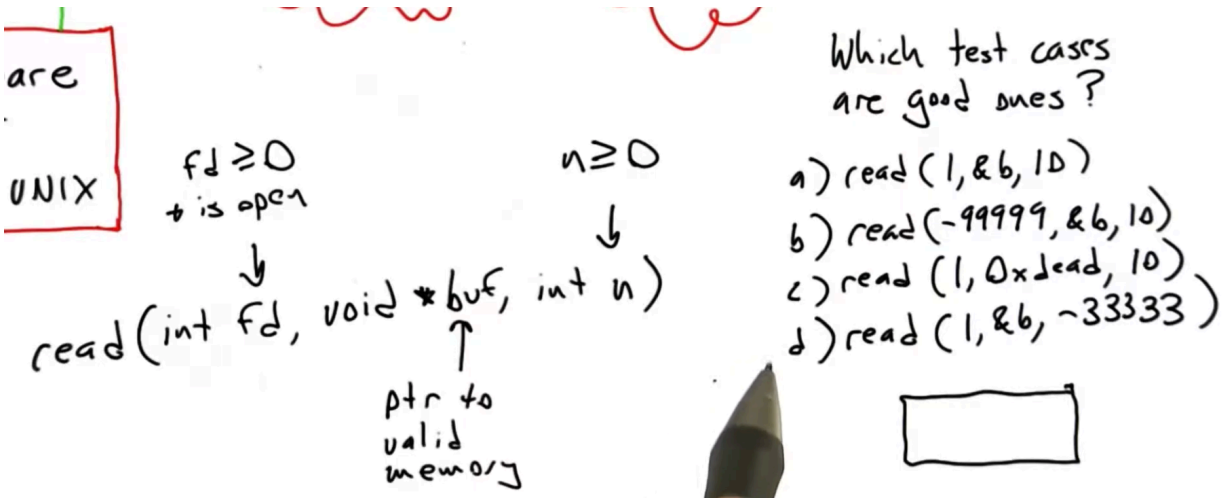


- The answer to this quiz is any answer is acceptable. The issue that we're getting at here is what's the specification for the SUT, for our square root routine?
  Is it defined to return the positive value? the negative value? Can it return either of them? And this is what we're getting at here is that software always operates under some sort of a specification.

18. Refining the specification



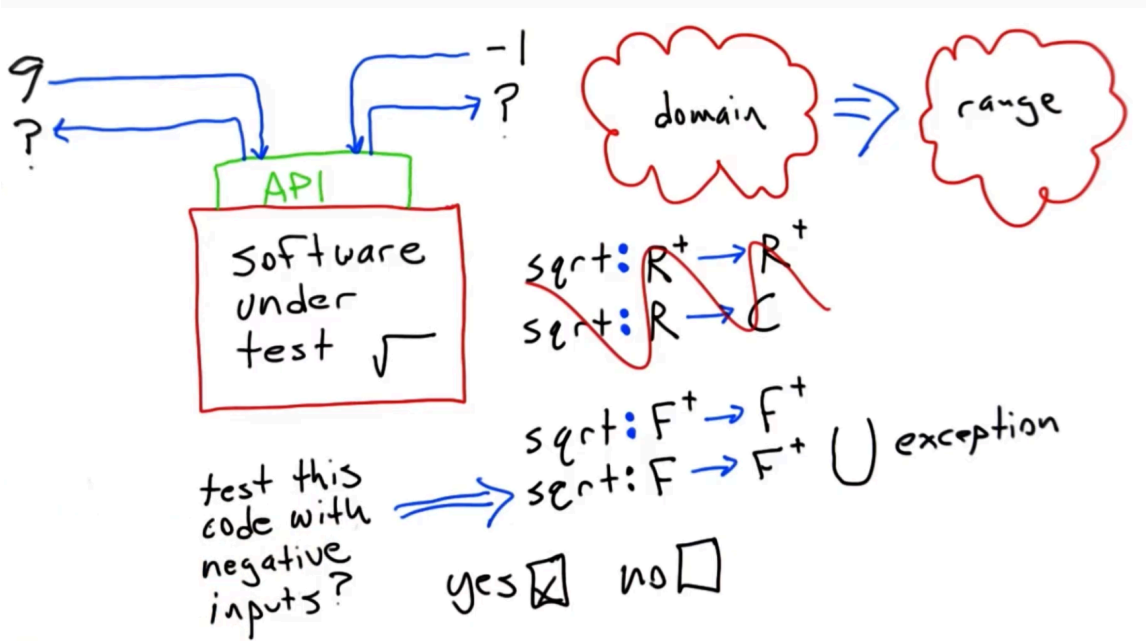
- Running a simple test case is forcing us to think about the specification for the SUT. And in fact, this is really common that as soon as we start testing a piece of software, we start to think about what the software is actually supposed to be doing.



- Often when we're testing software, we're not so much just looking for bugs in the software, but we're helping to refine the specification for the SUT.

19. Domains and ranges

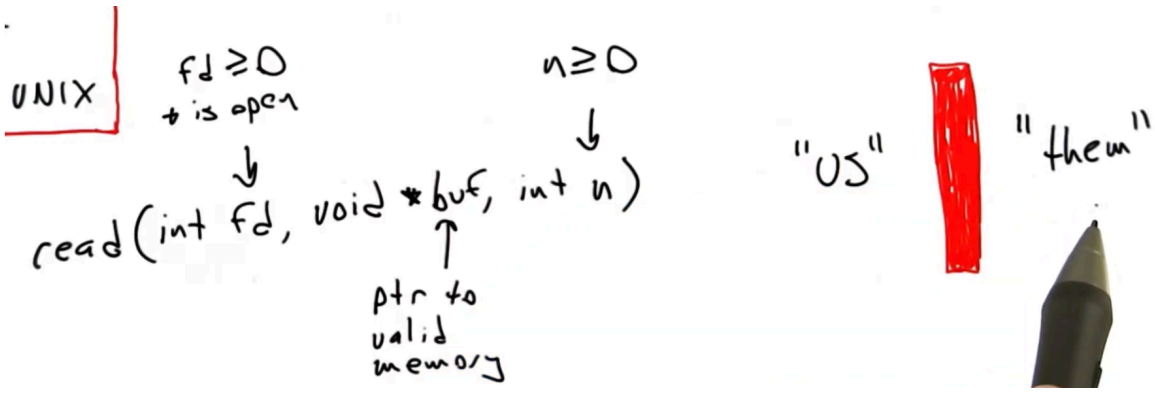
- If we think of a piece of software as a mathematical object, we'll find the software has a **domain of values**. Correspondingly, every piece of software also has a **range**.
- Sometimes as a software tester, you'll test code with an input that looks like it should be part of the domain and the code will malfunction, will crash with some sort of a bad error, perhaps maybe not throw an exception but rather actually exit abnormally.
- **Restrictions on the domain of functions** are actually a very **valuable tool in practice** because otherwise, every function or piece of software that we implement, has to contain maximal defensive code against illegal inputs. And in practice, this kind of defensive coding is not generally possible.



20. Good test cases

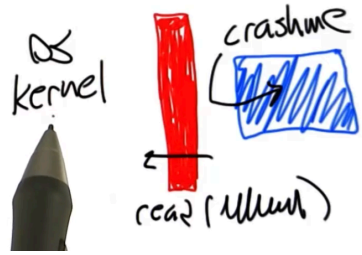
- Testing UNIX platform. All that read system call does is takes a file that's already open and reads some bytes out of it into the address space of the process that calls a read.

- Answer: all of the above.
- We're the kernel implementers, and our job is to keep the machine running to provide isolation between users and basically to enforce all of the security policies that operating systems are designed to enforce.
- On the other side of the boundary we have them. These are our users. They might not actually be malicious but writing buggy codes.
- Therefore, if we're testing the operating system interface, we really need to be issuing calls like read with garbage.



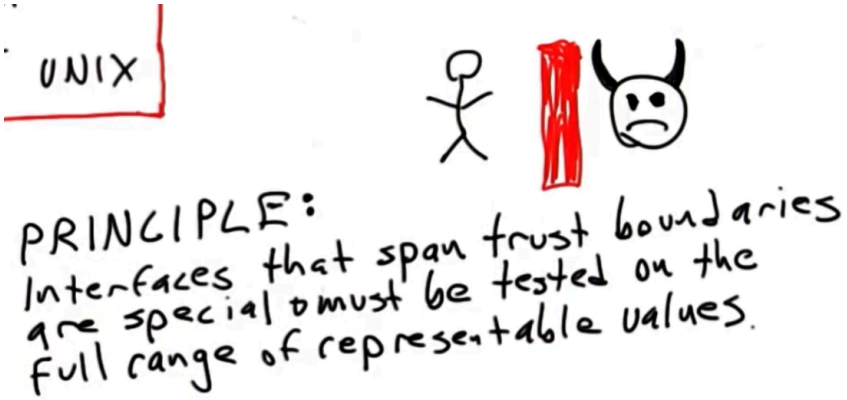
21. Crashme

- This is one of the ways that we actually test operating systems: using a tool called crashme that allocates a block of memory, writes totally random garbage into it, then it masks off all signal handlers, i.e. system level exception handlers and it jumps into the block of garbage, i.e. it starts executing completely garbage bytes.

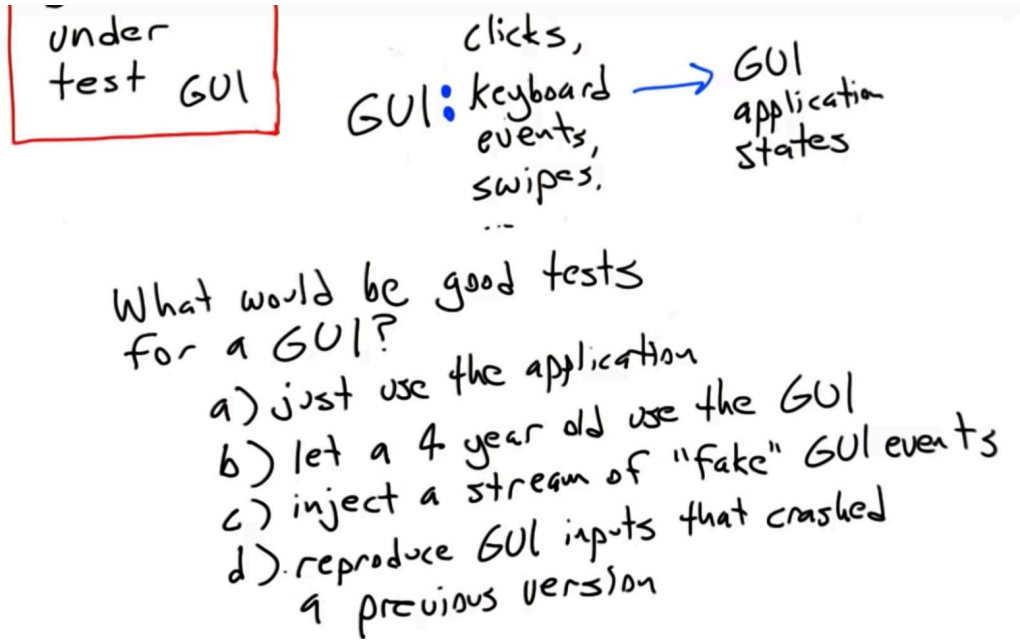


•So over time what we end up doing is exploring, i.e. testing a lot of operating system calls, that contain really weird and unexpected values.

- And if the kernel goes ahead and keeps running properly and keeps trying to kill the crashme process, then the operating system kernel is succeeding.
- If the kernel ever falls over, if it crashes, then we've discovered a bug.



22. Testing a GUI



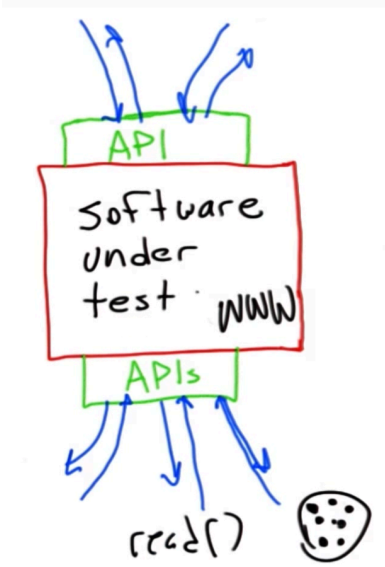
- Answer: all of the above.

23. Trust relationships

- The question: can I trust my teammates, and can my teammates trust me to always generate inputs when using the various APIs that remain within the domain of those inputs? And of course the answer is generally no.

•In fact, I probably can't even trust myself to always generate inputs that are within the domain of acceptable inputs for APIs.

- This brings us to is the idea of **defensive coding** i.e, error checking for its own sake to detect internal inconsistencies.
- Software doesn’t just provide APIs, it also uses them.



- Let’s say the **SUT** is something like a **web browser**. One thing we can do is test the web browser using the APIs that it provides, i.e. using its GUI and not worry about testing it with respect to the APIs that it uses.
- Let's take the case where our web browser is storing cookies. Most of the time during testing, we expect the storage and retrieval of cookies to operate perfectly normally. But what happens if, for example, the hard disk is full when the web browser tries to store a cookie?
- If we just hope that the software does the right thing, then one of the **golden rules of testing is we shouldn't ever just hope that it does something; we need to actually check this.**

- Back to the UNIX read system call. Before, we were concerned with the domain of the read system call, i.e. the set of possible valid arguments to the read system call and now we're concerned with the range because now we're not testing the UNIX operating system anymore; we're testing a program that runs on top of the UNIX OS.
- read is allowed to read less bytes than you actually asked for. It's going to return some number between 0 and count, but we don't know what number it's going to return.
- Another thing that read can do is just fail outright, i.e. it can return -1 to the application but it turns out that there are a whole lot of different reasons for that kind of a failure.

```
READ(2)                          Linux Programmer's Manual                          READ(2)


NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

       If  count is zero, read() returns zero and has no other results.  If count is greater than SSIZE_MAX,
       the result is unspecified.

RETURN VALUE
       On success, the number of bytes read is returned (zero indicates end of file), and the file  position
       is  advanced  by  this number.  It is not an error if this number is smaller than the number of bytes
       requested; this may happen for example because fewer bytes are actually available  right  now  (maybe
       because  we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or
       because read() was interrupted by a signal.  On error, -1 is returned, and  errno  is  set  appropri-
       ately.  In this case it is left unspecified whether the file position (if any) changes.

ERRORS
       EAGAIN The  file  descriptor fd refers to a file other than a socket and has been marked non-blocking
              (O_NONBLOCK), and the read would block.
```

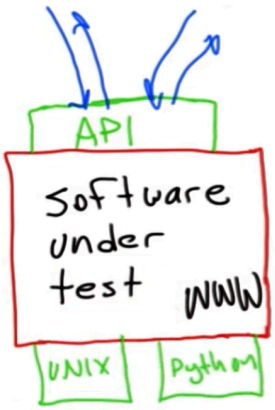
- We can see here that there are at least 9 different error conditions that read can return.

```
ERRORS
       EAGAIN The  file  descriptor fd refers to a file other than a socket and has been marked non-blocking
              (O_NONBLOCK), and the read would block.

       EAGAIN or EWOULDBLOCK
              The file descriptor fd refers to a socket and has been marked non-blocking  (O_NONBLOCK),  and
              the read would block.  POSIX.1-2001 allows either error to be returned for this case, and does
              not require these constants to have the same value, so a portable application should check for
              both possibilities.

       EBADF  fd is not a valid file descriptor or is not open for reading.

       EFAULT buf is outside your accessible address space.

       EINTR  The call was interrupted by a signal before any data was read; see signal(7).

       EINVAL fd  is  attached to an object which is unsuitable for reading; or the file was opened with the
              O_DIRECT flag, and either the address specified in buf, the value specified in count,  or  the
              current file offset is not suitably aligned.

       EINVAL fd  was created via a call to timerfd_create(2) and the wrong size buffer was given to read();
              see timerfd_create(2) for further information.

       EIO    I/O error.  This will happen for example when the process is in a  background  process  group,
              tries  to  read from its controlling tty, and either it is ignoring or blocking SIGTTIN or its
              process group is orphaned.  It may also occur when there is a low-level I/O error while  read-
              ing from a disk or tape.

       EISDIR fd refers to a directory.
```

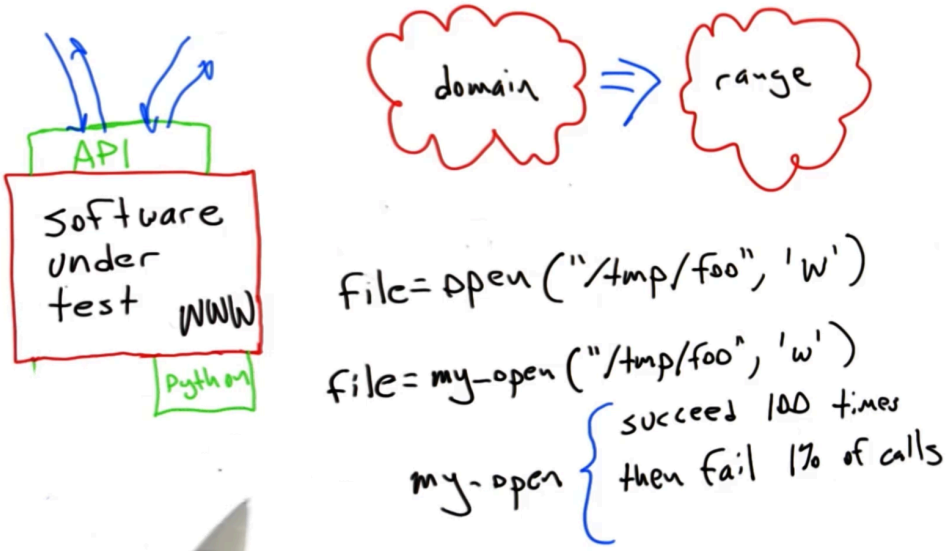
- The application might have to do different things depending on which of these values it gets. And the point is it might be very hard as people testing the web browser to actually make the operating system read call return all of those different values.
- And until we've tested it with all of those different values, we're left with software whose behavior we probably don't understand, and, therefore, it's software that hasn't been tested very well.

24. Fault injection

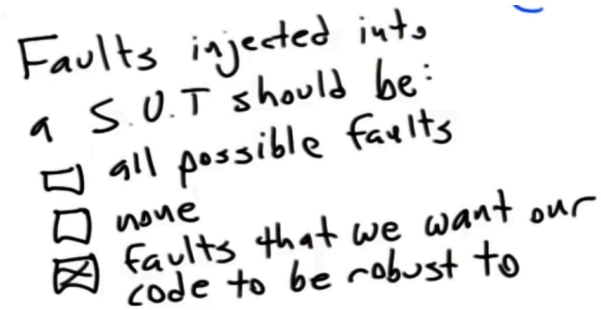
- What we have is a fairly difficult testing problem. In practice, there are a couple of different ways to deal with it.
- First, you should always try to use low level programming or interfaces that are predictable and that return friendly error codes. Given a choice between using the UNIX system call and using the Python libraries, you'd almost always choose the Python libraries.



- We don't always have the option of doing this so we're forced to use these bad style APIs sometimes. From a testing point of view, we can often use the technique called **fault injection** to deal with these kind of problems.
- Let's assume for the moment that we're using the Python library to create a file. We might have a fairly hard time testing the case where the open call fails because this call might almost always succeed. So what we can do instead is call a different function, my_open, which has an identical interface and almost an identical implementation.
- So we have a stub function and we can sometimes cause the open system call to fail.



- In practice, you have to be pretty careful with fault injection. One thing that can happen if you make my_open fail too often, for example, if it fails 50% of the time, then a program that's using it probably will never get off the ground.
- We would have to experiment with what kind of failure rates are good at testing the SUT:



25. Timing dependent problems

- We have our SUT, and it provides some APIs. Each API's collection of functions and most of the work that we have during testing is going to be calling these functions with various values and looking at the results.
- The issue is that the overt explicit interfaces that we see here don't represent all the possible inputs and outputs that we might care about.
- For example, on the input side it's completely possible that our SUT cares about the time at which inputs arrive.
- It might be the case that our software responds differently if 2 inputs arrive very close together than it does if 2 inputs arrive separated by a large amount of time.
- Another example is a web browser where if the data corresponding to a web page is returned in short time, this data will get rendered as a web page. But if the data that comes from the network is scattered across too much time, this is going to result in some sort of a timeout, i.e. SUT will render some sort of an error page.

26. Therac 25

- An extreme example of timing-dependent input being difficult to deal with is presented in the following. In the 1980s, there was a r**adiation therapy machine** called a Therac-25.

•It was used to deliver a highly concentrated beam of radiation to a part of a human body where that beam could be used to destroy cancerous tissue without harming tissue that's nearby.

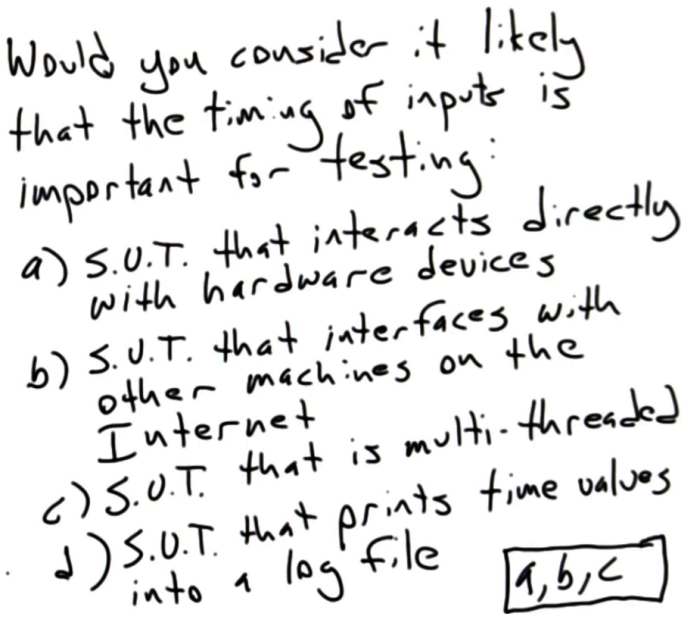
•This was not a safe technology as it depended on skilled operators and also highly safe software.

• There was a tragic series of mistakes where 6 people were subjected to massive radiation overdoses and several of these people died.

•The Therac-25 had **serious issues with its software**.

- It turned out that the people developing the software put a number of bugs into it.
- The particular bug was a software bug called a **race condition** that involved the keyboard input to the radiation therapy machine. If the operator of the machine typed slowly, the bug was very unlikely to be triggered. As they treated hundreds and hundreds of patients, they typed faster and faster, and eventually they started triggering this bug.
- The kind of quandary that this scenario raises for us as software testers is **do we have to care about the time at which inputs arrive at our SUT, or can we not worry about that?**
- And so obviously, for the Therac-25 and obviously, also for something like a Linux kernel, the time at which inputs arrive is relevant.
- On the other hand, unless we've been extremely sloppy, the square root function that we've been talking about won't care about the time at which its inputs arrive.

27. Testing timing

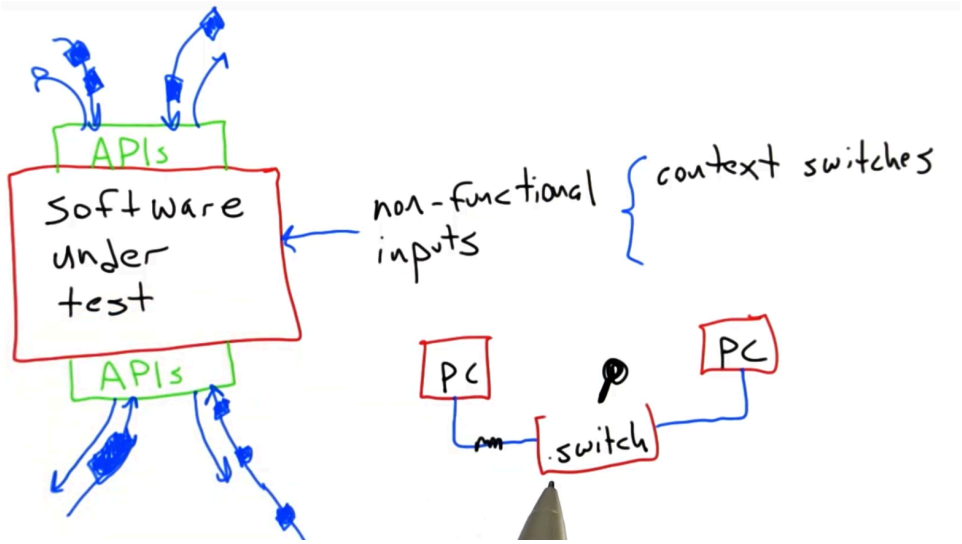


28. Taking time into account

- In order to figure out if timing matters for the inputs for the SUT is think about its specification, its requirements, and also look at the source code for things like timeout, time sleep, and basically values or computations that depend on the time at which things happen.

29. Nonfunctional inputs

- These are inputs that affect the operation of a SUT that have nothing to do with the APIs provided or that are used by the software that we're testing.
- Context switches are switches between different threads of execution in a multi-threaded SUT. The issue is that multiple threads of execution are scheduled along different processors on the physical machine that we're running on at different times, and it's the OS that makes the decisions about what thread goes on what processor at what time, and depending on the scheduling of these threads bugs in the SUT can either be concealed or revealed, and the problem is that the timing of these context switches is completely not under the control of our application.
- For example, in the late 1990s a company made very, very fast networking hardware. A problem was that the network was extremely reliable implying a real difficulty in testing the end-to-end reliability software.
- So the tester opened up a switch, exposing all of the electrical contacts inside, and then took a metal key, and run it across the contacts that were exposed. Either the network would glitch for a moment or else it would fail to resume properly:



30. Testing survey

- It's not intended to be exhaustive because there's going to be a bunch of overlap between these different categories.

•**White box testing** refers to is the fact that the tester is using detailed knowledge about the **internals of the system** in order to construct better test cases and **black box testing** refers to the fact that we're rather testing the system based on what we know about **how it's supposed to respond** to our test cases.

31. Unit testing

- Unit testing means looking at some **small software module** at a time and **testing** it **in an isolated fashion**.
- The main thing that distinguishes unit testing from other kinds of testing is that we're **testing a smaller amount of software**.
- Often the person performing the unit testing is the same person who implemented the module, and in that case we may well be doing white box testing but unit testing can also be black box testing.
- The goal of unit testing is to **find defects in the internal logic of the SUT as early as possible**, in order to create more robust software modules that we can compose later with other modules and end up with a system that actually works.
- Another thing that distinguishes unit testing from other kinds of testing is that generally, at this level, **we have no hypothesis about the patterns of usage of SUT**.

- In other words, we're going to try to test the unit with inputs from all different parts of its domain (the set of possible inputs).
- Python has a number of frameworks for unit testing and also for mock objects (fake objects).

32. Integration testing

- Integration testing refers to taking **multiple software modules that have already been unit tested and testing them in combination with each other**.

- What we're really testing are the interfaces between modules, and the question is did we define them tightly enough and specify them tightly enough that the different groups of people implementing the different modules were able to make compatible

assumptions, which are necessary for the software modules to all actually work together.

- Coming up with a software module that survives integration testing is a lot harder than creating reliable small units.

33. System testing

- Here we're asking the question **does the system as a whole meet its goals**?
- And often at this point we're doing black box testing, and that's for a couple of reasons:
  - the system is probably large enough,
  - we're not so much concerned with what's going on inside the system,
  - at this level we are often concerned with how the system will be used,
  - we may not care about asking the system work for all possible use cases. Rather, we would simply like to make sure that it performs acceptably for the important use cases.

34. Other kinds of testing

- In **differential testing** we are taking the same test input delivering it to 2 different implementations of the SUT and comparing them for equality.

- **Stress testing** is a kind of testing where a system is tested at or beyond its normal usage limits, and it's probably best described through a couple of examples.
  - With the square root function we might test it with very large numbers or very tiny numbers.
  - For an OS we might test it by making a huge number of system calls or by making very large memory allocations or by creating extremely large files.
  - For a web server we could stress test it by making many requests, or even better, by making many requests, all of which require he database to communicate with its backend.
- Stress testing is typically done to assess the robustness and reliability of the SUT.

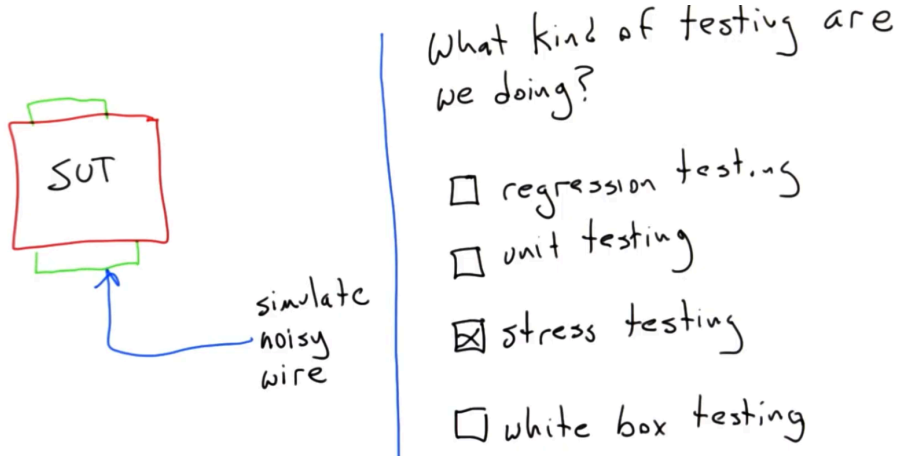
- In **random testing** we use the results of a pseudo-random number generator to randomly create test inputs, and we deliver those to the SUT.
- Very often this can be much more subtle and more

sophisticated than just throwing random bits at the software. And random testing is very often useful for finding corner cases in software systems, and the crashme program for Linux and other Unix kernels is a great example of a random tester.

- **Regression testing** always involves taking inputs that previously made the system fail and replaying them against the system.

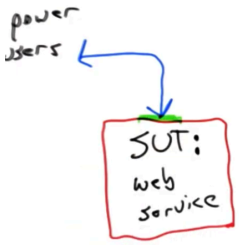
35. Testing car software

- We have a piece of SUT that is some sort of a critical embedded system, for example, it's controlling airbags on a vehicle. And this vehicle is going to be out in the field for many years, and so cars get subjected to fairly extreme conditions.
  - They have a lot of heating and cooling cycles.
  - They get exposed to salt and mud and water, and so the cars are in a fairly harsh environment.
- We want to know how the software responds to defects that get introduced into the automobile's wiring as this vehicle ages in the field.
- We're going to use some sort of a simulation module to simulate a noisy wire, a wire whose insulation has rubbed off and that is not perfectly conducting signals anymore.
- What kind of testing are we doing?



36. Testing a web service

- SUT is some sort of a web service, exposed to some small fraction of users who have been selected, based on their willingness and desire to use new features.
- What kind of testing are we doing?



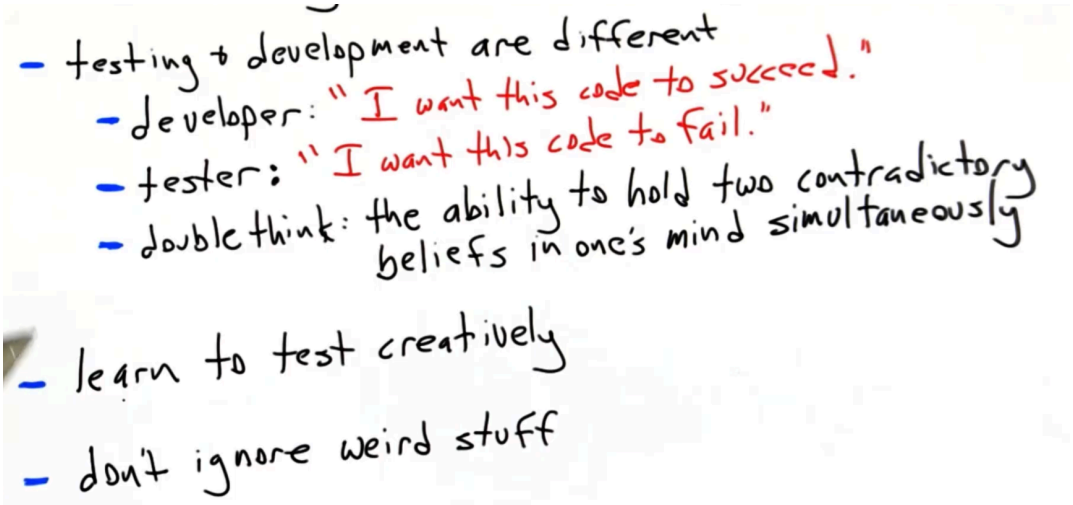
- [ ] integration testing
- [ ] differential testing
- [ ] random testing
- [x] system/validation testing

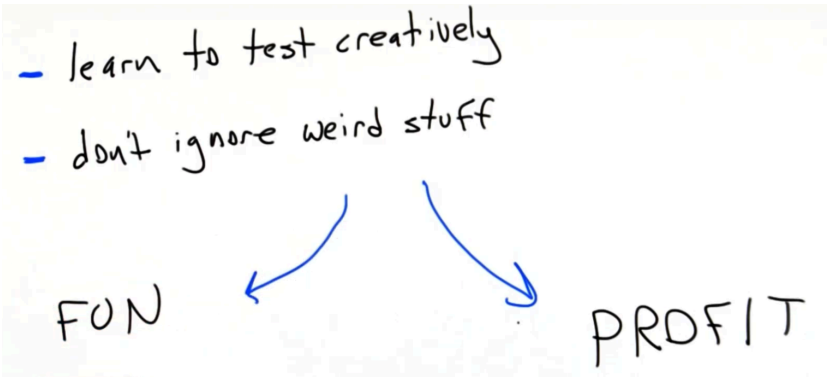
37. Testing a new library

- In this scenario we have some large SUT, and part of it is a library that has been giving us problems.
- Let's say this library is implementing numerical functions, and these numerical functions have been sometimes throwing floating point exceptions and crashing our system.
- The vendor has given us a new version and we're going to spend some time checking and trying to make sure that even if it's not that great it's at least not worse than the previous version.
- What kind of testing are we doing?

- [x] unit testing
- [ ] white box testing
- [ ] black box testing
- [ ] stress testing

38. Being great at testing




- learn to test creatively
- don't ignore weird stuff
FUN
PROFIT

Blackbox testing

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

Problem

- Break 5 incorrect implementations of the fixed-size queue using assert statements (i.e. catch buggy queue implementations)
- The queue class is a fixed-size queue of integers. There are 4 methods that we're going to need to concern ourselves with and these method calls are all we're going to have access to during testing (we don't actually know the code that we're running against, we don't have access to it).

```
# the Queue class provides a fized-size FIFO queue of integers

# the constructor takes a single parameter: an integer >0 that
# is the maximum number of elements the queue can hold

# empty() returns True iff the queue currently
# holds no elements, and False otherwise.

# full() returns True iff the queue cannot hold
# any more elements, and False otherwise.

# enqueue(i) attempts to put the integer i into the queue; it returns
# True if successful and False if the queue is full

# dequeue() removes an integer from the queue and returns it,
# or else returns None if the queue is empty

# Example:
# q = Queue(1)
# is_empty = q.empty()
# succeeded = q.enqueue(10)
# is_full = q.full()
# value = q.dequeue()
#
# 1. Should create a Queue q that can only hold 1 element
# 2. Should then check whether q is empty, which should return True
# 3. Should attempt to put 10 into the queue, and return True
# 4. Should check whether q is now full, which should return True
# 5. Should attempt to dequeue and put the result into value, which
#    should be 10
#
# Your test function should run assertion checks and throw an
# AssertionError for each of the 5 incorrect Queues.


def correct_test():
    ###Your code here.
```

Queue1

```
import array


######
###### this queue is buggy: silently holds 2 byte integers only
######

class Queue1:
    def __init__(self,size_max):
        assert size_max > 0
        self.max = size_max
        self.head = 0
        self.tail = 0
        self.size = 0
        self.data = array.array('i', range(size_max))

    def empty(self):
        return self.size == 0

    def full(self):
        return self.size == self.max

    def enqueue(self, x):
        if self.size == self.max:
            return False
        x = x % (2**16)  # only stores integers up to 2^16
        self.data[self.tail] = x
        self.size += 1
        self.tail += 1
        if self.tail == self.max:
            self.tail = 0
        return True

    def dequeue(self):
        if self.size == 0:
            return None
        x = self.data[self.head]
        self.size -= 1
        self.head += 1
        if self.head == self.max:
            self.head = 0
        return x
```

Queue2

```
######
###### this queue is buggy: it silently fails to create queues with
###### more than 15 elements
######

class Queue2:
    def __init__(self,size_max):
        assert size_max > 0
        if size_max > 15:  # max_size is set to 15 elements
            size_max = 15
        self.max = size_max
        self.head = 0
        self.tail = 0
        self.size = 0
        self.data = array.array('i', range(size_max))

    def empty(self):
        return self.size == 0

    def full(self):
        return self.size == self.max

    def enqueue(self, x):
        if self.size == self.max:
            return False
        self.data[self.tail] = x
        self.size += 1
        self.tail += 1
        if self.tail == self.max:
            self.tail = 0
        return True

    def dequeue(self):
        if self.size == 0:
            return None
        x = self.data[self.head]
        self.size -= 1
        self.head += 1
        if self.head == self.max:
            self.head = 0
        return x
```

Queue3

```
######
###### this queue is buggy: its empty() method is implemented by seeing
###### if an element can be reached
######

class Queue3:
    def __init__(self,size_max):
        assert size_max > 0
        self.max = size_max
        self.head = 0
        self.tail = 0
        self.size = 0
        self.data = array.array('i', range(size_max))

    def empty(self):
        return self.dequeue() is None  # tricky; trying to dequeue an element
and checking if is None

    def full(self):
        return self.size == self.max

    def enqueue(self, x):
        if self.size == self.max:
            return False
        self.data[self.tail] = x
        self.size += 1
        self.tail += 1
        if self.tail == self.max:
            self.tail = 0
        return True

    def dequeue(self):
        if self.size == 0:
            return None
        x = self.data[self.head]
        self.size -= 1
        self.head += 1
        if self.head == self.max:
            self.head = 0
        return x
```

Queue4

```
######
###### this queue is buggy: dequeue of empty returns False instead of None
######

class Queue4:
    def __init__(self,size_max):
        assert size_max > 0
        self.max = size_max
        self.head = 0
        self.tail = 0
        self.size = 0
        self.data = array.array('i', range(size_max))

    def empty(self):
        return self.size == 0

    def full(self):
        return self.size == self.max

    def enqueue(self, x):
        if self.size == self.max:
            return False
        self.data[self.tail] = x
        self.size += 1
        self.tail += 1
        if self.tail == self.max:
            self.tail = 0
        return True

    def dequeue(self):
        if self.size == 0:
            return False  # returns False where it should, according to the
specification return None
        x = self.data[self.head]
        self.size -= 1
        self.head += 1
        if self.head == self.max:
            self.head = 0
        return x
```

Queue5

```
######
###### this queue is buggy: it holds one less item than intended
######

class Queue5:
    def __init__(self,size_max):
        assert size_max > 0
        size_max -= 1  # holds one less element than you tell it to hold
        self.max = size_max
        self.head = 0
        self.tail = 0
        self.size = 0
        self.data = array.array('i', range(size_max))

    def empty(self):
        return self.size == 0

    def full(self):
        return self.size == self.max

    def enqueue(self, x):
        if self.size == self.max:
            return False
        self.data[self.tail] = x
        self.size += 1
        self.tail += 1
        if self.tail == self.max:
            self.tail = 0
        return True

    def dequeue(self):
        if self.size == 0:
            return None
        x = self.data[self.head]
        self.size -= 1
        self.head += 1
        if self.head == self.max:
            self.head = 0
        return x
```

Tests

```
def correct_test():
    # Queue1 silently holds only 2 byte unsigned integers, than wraps around
    q = Queue1(2)
    succeeded = q.enqueue(100000)  # value greater than 2^16
    assert succeeded
    value = q.dequeue()
    assert value == 100000  # test1 failed

    # Queue2 silently fails to hold more than 15 elements
    q = Queue2(30)
    # try to enqueue more than 15 elements
    for i in range(20):
        succeeded = q.enqueue(i)
        assert succeeded  # test2 failed

    # Queue3 implements empty() by checking if dequeue() succeeds.
    # This changes the state of the queue unintentionally.
    q = Queue3(2)
    succeeded = q.enqueue(10)
    assert succeeded
    assert not q.empty()  # the function checks by trying to dequeue
    value = q.dequeue()
    assert value == 10  # test3 failed

    # Queue4 dequeue() of an empty queue returns False instead of None
    q = Queue4(2)
    value = q.dequeue()
    assert value is None  # test4 failed

    # Queue5 holds one less item than intended
    q = Queue5(2)
    for i in range(2):
        succeeded = q.enqueue(i)
        assert succeeded  # test4 failed


correct_test()
```

Coverage testing

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

1. Introduction

- Usually problems in released software are coming from things that people forgot to test, i.e. testing was inadequate and the developers were unaware of that fact.
- In the following we present a collection of techniques called **code coverage** where automated tools can tell us places where our testing strategy is not doing a good job.

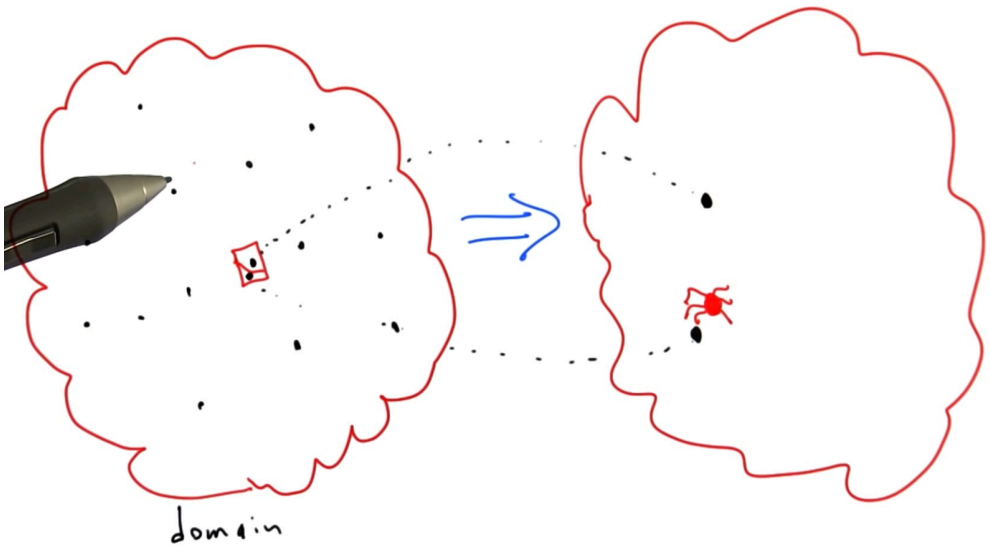
2. How much testing is enough?

- One of the trickiest things about testing software is that it's hard to know when you've done enough testing and the fact is, it is really easy to spend a lot of time testing and to start to believe that you did a good job and then to have some really nasty bug show up that are triggered by parts of the input space that we just didn't think to test.
- We consider some test cases of the domain, i.e. testing is being compliant to some small part of the input domain and the problem, is that even the small part of the domain may contain an infinite number of test cases.
- Next we consider test cases for other parts of the domain we didn't think to test, putting the results and outputs that are not okay (bad range).
- Assume we have a small Python program that cause the Python runtime to crash.
  - The distinguishing feature of it seem to be a large number of cascaded if statements.
  - It's easy if we're testing to remain in the part of the space where, for example, we have < 5 nested ifs.
  - Another region contains >= 5 nested ifs that cause the Python virtual machine to crash.

- Let's say that we're testing some software that somebody has inserted a back door so that can’t be triggered it accidentally (extremely bad range). We didn't tested inputs triggering the back door because we just didn't know it was there.
- We need some sort of a tool (automated scoring system) or methodology that if we are in fact testing only a small part of the input domain for a system to look at our testing effort and to say that the score is, for example, 14 out of 100. Reasons:
  - Our testing efforts will improve by helping us find the input domain that need more testing.
  - We can argue that we've done enough testing.
  - We can identify parts of the test suite that are completely redundant.

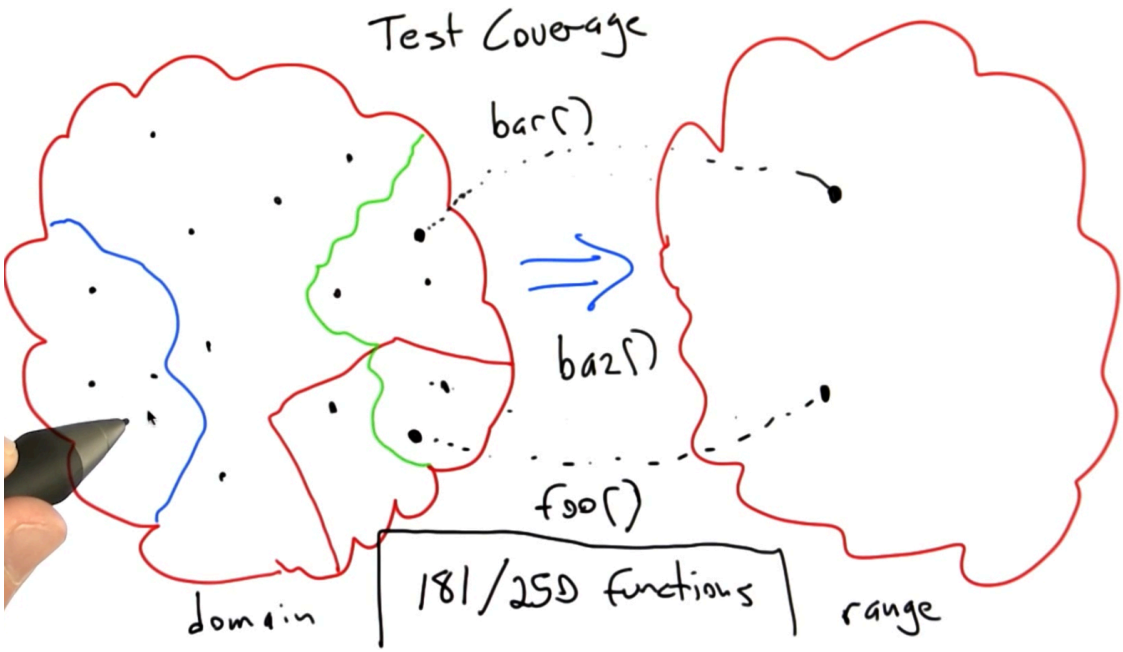
3. Partitioning the input domain

- We start with some SUT and it’s going to have a set of possible inputs, i.e. an input domain, and usually it consists of many possible test cases, that there is no way we can possibly test them all.
- People were often interested in ways to partition the input domain into a no. of different classes so that all of the points within each class are treated the same by the SUT.
- Let’s consider a subset of the input domain. For purposes of finding defects in the SUT, we pick an arbitrary point and execute the system on it. We look at the output, and if it is acceptable, then we’re done testing that class of inputs.
- In practice sometimes what we thought was a class of inputs that are all equivalent might be in different classes. We can blame the partitioning and the unfortunate fact is the original definition of this partitioning scheme didn’t give us good guidance in how to actually do the partitioning.



4. Coverage

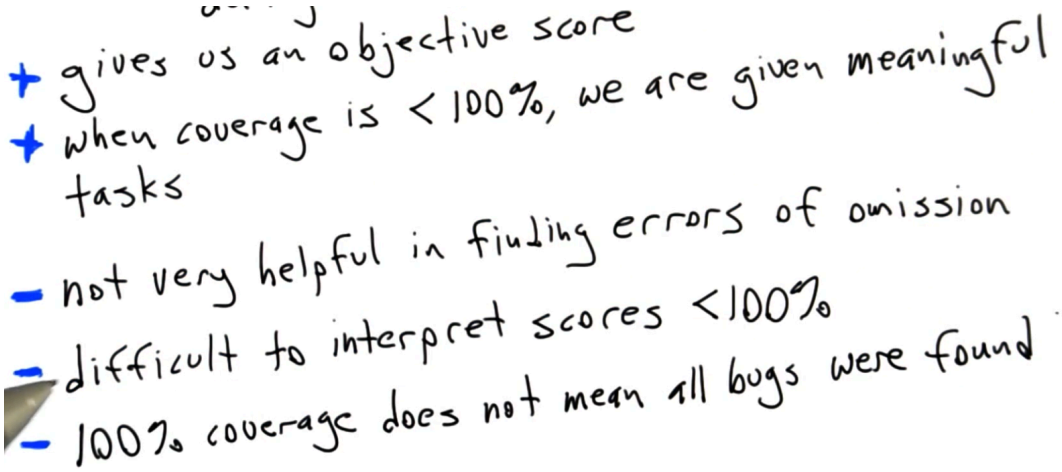
- In practice we ended up with the notion of test coverage instead of a good partitioning of the input domain.
- **Test coverage** is trying to accomplish exact the same thing that partitioning was accomplishing, but it goes about in a different way.
- Test coverage is an **automatic way of partitioning the input domain with some observed features of the source code.**
- One particular kind of test coverage is called **function coverage** and is achieved when every function in our source code is executed during testing.
- We subdivide the input domain for the SUT until we have split it into parts that results in every function being called.



- In practice, we start with a set of test of cases, and we run them all through the SUT. We see which functions are called and then we end up with some sort of a **score** called a **test coverage metric**.
- The score assigned to a collection of test cases is very useful.
  - For each of the functions that wasn't covered, we can go and look at it and we can try to come up with the test input that causes that function to execute.
  - If there is some function baz() for which we can't seem to devise an input that causes it to execute, then there are a couple of possibilities. One possibility is that it can't be called at all. It's **dead code**. Another possibility is that we simply don't understand our system well enough to be able to trigger it.

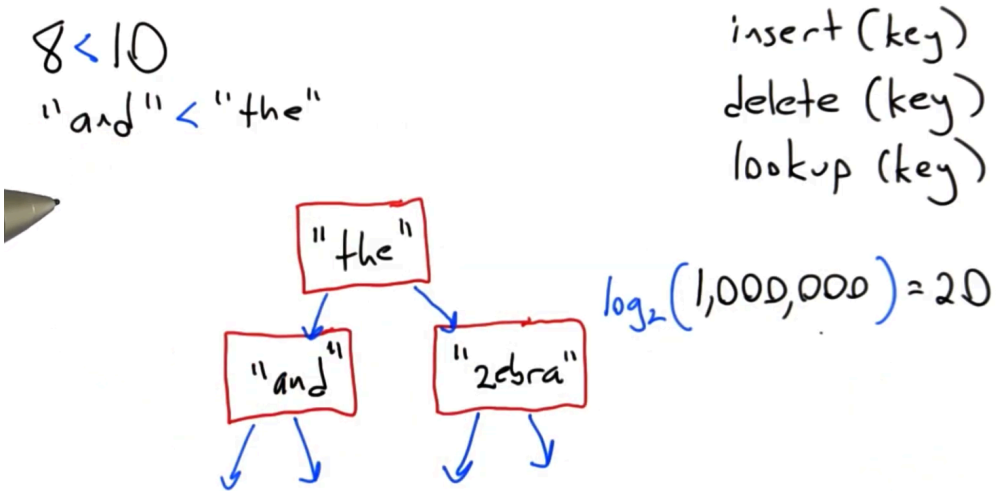
5. Test coverage

- Test coverage is **a measure of the proportion of a program exercised during testing**.



6. Splay tree

- Let's take a concrete look of what a coverage can do for us in practice.
- We'll look at some random open source Python codes that implements a splay tree, a kind of binary search tree.
- A binary search tree is a tree where every node has 2 leaves and it supports operations such as insert, delete, and lookup. The main important thing is the keys have to support an order in relation.
  - If we're using integers for keys then we can use < for order in relation.
  - If we're using words as our keys, then we can use dictionary order.



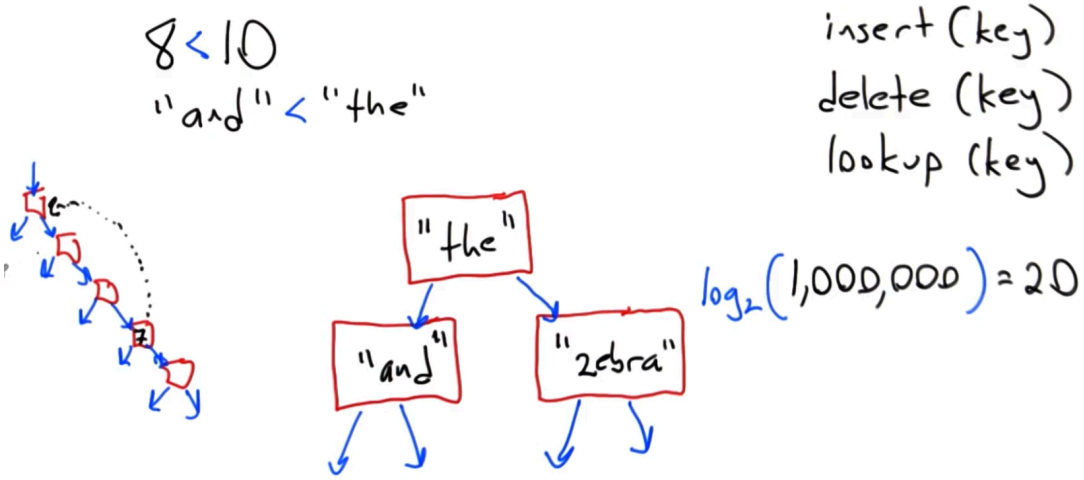
- The way the binary search tree is going to work is, we're going to build up a tree under the invariant that the left child of any node always has a key that's ordered before the

key of the parent node and the right child is always ordered after the parent node using the ordering.

- For a large tree with this kind of shape we have a procedure for fast lookup.
- The way that these trees are set up means that, on average, each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree, O(log n).

7. Splay tree issues

- Splay tree is a the simplest example of self-balancing binary search tree. As we add elements a procedure keeps the tree balanced so that tree operations remain very fast.
- It has a really cool property that when we access the nodes, let's say we do a lookup of this node which contains 7, what's going to happen is as a side-effect of the lookup that node is going to get migrated up to the root and then whatever was previously at the root is going to be pushed down and possibly some sort of a balancing operation is going to happen.
  - The point is, that frequently accessed elements end up being pushed towards the root of a tree and therefore, future accesses to these elements become even faster.



- In the following we will look at an open source splay tree implemented in Python, found on the web, that comes with its own test suite and we're going to look at what kind of code coverage this test suite gets on the splay tree.

8. Splay tree example

https://codereview.stackexchange.com/questions/209904/unit-testing-for-splay-tree-in-python

```
class Node:
      def __init__(self, key):
            self.key = key
            self.left = self.right = None

      def equals(self, node):
            return self.key == node.key

class SplayTree:
      def __init__(self):
            self.root = None
            self.header = Node(None)  # for splay()

      def insert(self, key):
            if (self.root == None):
                  self.root = Node(key)
            return

            self.splay(key)
            if self.root.key == key:
                  # If the key is already there in the tree, don't do
anything.
                  return

            n = Node(key)
            if key < self.root.key:
                  n.left = self.root.left
                  n.right = self.root
                  self.root.left = None
            else:
                  n.right = self.root.right
                  n.left = self.root
                  self.root.right = None
            self.root = n

      def remove(self, key):
            self.splay(key)
            if key != self.root.key:
                  raise 'key not found in tree'

            # Now delete the root.
            if self.root.left == None:
                  self.root = self.root.right
            else:
                  x = self.root.right
                  self.root = self.root.left
                  self.splay(key)
                  self.root.right = x
```

```
    def findMin(self):
        if self.root == None:
            return None
        x = self.root
        while x.left != None:
            x = x.left
        self.splay(x.key)
        return x.key

    def findMax(self):
        if self.root == None:
            return None
        x = self.root
        while x.right != None:
            x = x.right
        self.splay(x.key)
        return x.key

    def find(self, key):
        if self.root == None:
            return None
        self.splay(key)
        if self.root.key != key:
            return None
        return self.root.key

    def isEmpty(self):
        return self.root == None
```

- The splay operation moves a particular key up to the root of the binary search tree. This serves as both the balancing operation and also the look up:

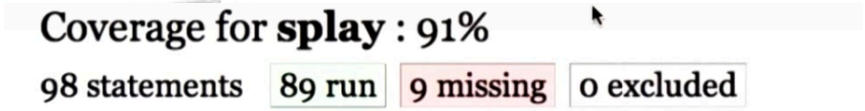
```
    def splay(self, key):
        l = r = self.header
        t = self.root
        self.header.left = self.header.right = None
        while True:
            if key < t.key:
                if t.left == None:
                    break
                if key < t.left.key:
                    y = t.left
                    t.left = y.right
                    r.right = t
                    t = y
                    if t.left == None:
                        break
                r.left = t
                r = t
                t = t.left
            elif key > t.key:
                if t.right == None:
                    break
                if key > t.right.key:
                    y = t.right
                    t.right = y.left
                    r.left = t
                    t = y
                    if t.right == None:
                        break
                l.right = t
                l = t
                t = t.right
            else:
                break
        l.right = t.left
        r.left = t.right
        t.left = self.header.right
        t.right = self.header.left
        self.root = t
```

- Now, let's look at the test suite:

```
import unittest  # module for unit testing
from splay_tree import SplayTree



class TestCase(unittest.TestCase):
    def setUp(self):
        self.keys = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        self.t = SplayTree()
        for key in self.keys:
            self.t.insert(key)

    def testInsert(self):
        for key in self.keys:
            self.assertEquals(key, self.t.find(key))

    def testRemove(self):
        for key in self.keys:
            self.t.remove(key)
            self.assertEquals(self.t.find(key), None)

    def testLargeInserts(self):
        t = SplayTree()
        nums = 40000
        gap = 307  # stress testing
        i = gap
        while i != 0:
            t.insert(i)
            i = (i + gap) % nums

    def testIsEmpty(self):
        self.assertFalse(self.t.isEmpty())
        t = SplayTree()
        self.assertTrue(t.isEmpty())

    def testMinMax(self):
        self.assertEquals(self.t.findMin(), 0)
        self.assertEquals(self.t.findMax(), 9)


if __name__ == "__main__":
    unittest.main()
```

- After running the tests we can use a **code coverage tool** to measure coverage. We can generate a HTML report. In our case, it's telling us that when we run the splay tree on its own unit test suite, out of the 98 statements in the file, 89 of them got run, 9 of them failed to run.

```
udacitys-imac:files udacity$ emacs splay.py
udacitys-imac:files udacity$ python splay_test.py
.....
----------------------------------------------------------------------
Ran 5 tests in 1.686s

OK
udacitys-imac:files udacity$ coverage erase ; coverage run splay_test.py ; coverage html -i
```




```
1  class Node:
2      def __init__(self, key):
3          self.key = key
4          self.left = self.right = None
5
6      def equals(self, node):
7          return self.key == node.key
8
9  class SplayTree:
10     def __init__(self):
11         self.root = None
12         self.header = Node(None) #For splay()
13
14     def insert(self, key):
15         if (self.root == None):
16             self.root = Node(key)
17             return
```

- The test suite failed to test the case where we inserted an element into the tree and it was already there.
- In the splay tree's removing function the first thing this function does is, splays the tree based on the key to be removed and so this is intended to draw that key up to the root note of the tree. If the root node of the tree does not have the key that we're looking for, then we're going to raise an exception saying that this key wasn't found. But this wasn't tested.
- If we look in the body of the delete function, we see a pretty significant chunk of code that wasn't tested, so we have to go back and revisit this.

- In conclusion, the tool is showing us what we didn't think to test with the unit test suite that we wrote so far.

9. Improving coverage

- We will modify in the test suite, testRemove function:

```
    def testRemove(self):
        for key in self.keys:
            self.t.remove(key)
            self.assertEquals(self.t.find(key), None)
        self.t.remove(-999)
```

- After running the tool the line that we just added causes an exception re-thrown in the splay function. So we have found a bug not anticipated by the developer of the splay tree.
- It often turns out that the stuff that we thought was going to run might be running only some of it. So the coverage tool told us something interesting.
- On the other hand, if the coverage tool hasn't told us anything interesting, i.e. everything we hoped was executed well when we run the unit test suite then that's good, too.
- Another thing we noticed is that the bug was somewhere completely different buried in the splay routine and if we go back and look at the coverage information, it turns out that the splay routine is entirely covered, i.e. every line of code was executed during the execution of the unit test for the splay tree.
- We deduce that just because some code was covered, especially at the statement level, this does not mean that it doesn’t contain a bug in it.
  - We have to ask the question, "What do we want to really read into the fact that we failed to cover something?” The coverage tool has given an example suggesting that our test suite is poorly thought out.
  - So, when coverage fails its better to try to think about why went wrong rather than just blindly writing a test case and just exercise the code which wasn't covered.

10. Problems with coverage

- We looked to an example where measuring coverage was useful in finding a bug in a piece of code. The coverage is not particularly useful in spotting the bug.
- In the following we will discuss about another piece of code, a broken function whose job is to determine whether a number is prime.

```
# CORRECT SPECIFICATION:
#
# isPrime checks if a positive integer is prime.
#
# A positive integer is prime if it is greater than
# 1, and its only divisors are 1 and itself.


import math

def isPrime(number):
    if number<=1 or (number%2)==0:
        return False
    for check in range(3,int(math.sqrt(number))):
        if (number%check) == 0:
            return False
    return True

def test():
    assert isPrime(1) == False
    assert isPrime(2) == False
    assert isPrime(3) == True
    assert isPrime(4) == False
    assert isPrime(5) == True
    assert isPrime(20) == False
    assert isPrime(21) == False
    assert isPrime(22) == False
    assert isPrime(23) == True
    assert isPrime(24) == False
```

- isPrime function has successfully identified whether the input is prime or not for the 10 numbers in the assertions.
- When we run the code coverage tool we can see out of the 20 statements in the file, all of them run, and none of them failed to be covered. Statement coverage gives us a perfect result for this particular code and yet another set is wrong.

```
# TASKS:
#
# 1) Add an assertion to test() that shows
#    isPrime(number) to be incorrect for
#    some input.
#
# 2) Write isPrime2(number) to correctly
#    check if a positive integer is prime.

# Note that there is an error where isPrime() incorrectly returns False for
an input of 2, despite 2 being a prime number.
# This has been fixed in the following.
# In isPrime function the range loops between 3 and the square of the input-1


import math

def isPrime2(number):
    if number == 2:
        return True
    if number<=1 or (number%2)==0:
        return False
    for check in range(3,int(math.sqrt(number)) + 1):
        if (number%check) == 0:
            return False
    return True

def test():
    assert isPrime(6) == False
    assert isPrime(7) == True
    assert isPrime(8) == False
    assert isPrime(9) == True
    assert isPrime(10) == False
    assert isPrime(25) == True
    assert isPrime(26) == False
    assert isPrime(27) == False
    assert isPrime(28) == False
    assert isPrime(29) == True

def test2():
    assert isPrime2(6) == False
    assert isPrime2(7) == True
    assert isPrime2(8) == False
    assert isPrime2(9) == False
    assert isPrime2(10) == False
    assert isPrime2(25) == False
    assert isPrime2(26) == False
    assert isPrime2(27) == False
    assert isPrime2(28) == False
    assert isPrime2(29) == True
```

- Why did test coverage fail to identify the bug? Statement coverage is a rather crude metric that only checks whether each statement executes once. Each statement executes at least once that lets a lot of bugs slip through.

- The lesson here is we should not let complete coverage plus a number of successful test cases fool as into thinking that a piece of code of right. It's often the case that deeper analysis is necessary.

TO BE CONTINUED…

Coverage testing

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

1. Introduction

- Usually problems in released software are coming from things that people forgot to test, i.e. testing was inadequate and the developers were unaware of that fact.
- In the following we present a collection of techniques called **code coverage** where automated tools can tell us places where our testing strategy is not doing a good job.

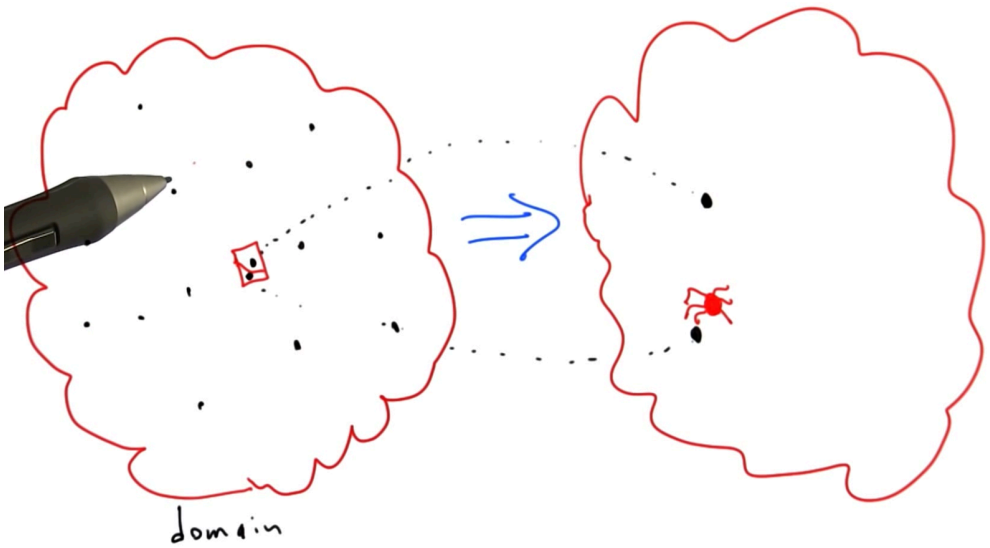
2. How much testing is enough?

- One of the trickiest things about testing software is that it's hard to know when you've done enough testing and the fact is, it is really easy to spend a lot of time testing and to start to believe that you did a good job and then to have some really nasty bug show up that are triggered by parts of the input space that we just didn't think to test.
- We consider some test cases of the domain, i.e. testing is being compliant to some small part of the input domain and the problem, is that even the small part of the domain may contain an infinite number of test cases.
- Next we consider test cases for other parts of the domain we didn't think to test, putting the results and outputs that are not okay (bad range).
- Assume we have a small Python program that cause the Python runtime to crash.
  - The distinguishing feature of it seem to be a large number of cascaded if statements.
  - It's easy if we're testing to remain in the part of the space where, for example, we have < 5 nested ifs.
  - Another region contains >= 5 nested ifs that cause the Python virtual machine to crash.

- Let's say that we're testing some software that somebody has inserted a back door so that can’t be triggered it accidentally (extremely bad range). We didn't tested inputs triggering the back door because we just didn't know it was there.
- We need some sort of a tool (automated scoring system) or methodology that if we are in fact testing only a small part of the input domain for a system to look at our testing effort and to say that the score is, for example, 14 out of 100. Reasons:
  - Our testing efforts will improve by helping us find the input domain that need more testing.
  - We can argue that we've done enough testing.
  - We can identify parts of the test suite that are completely redundant.

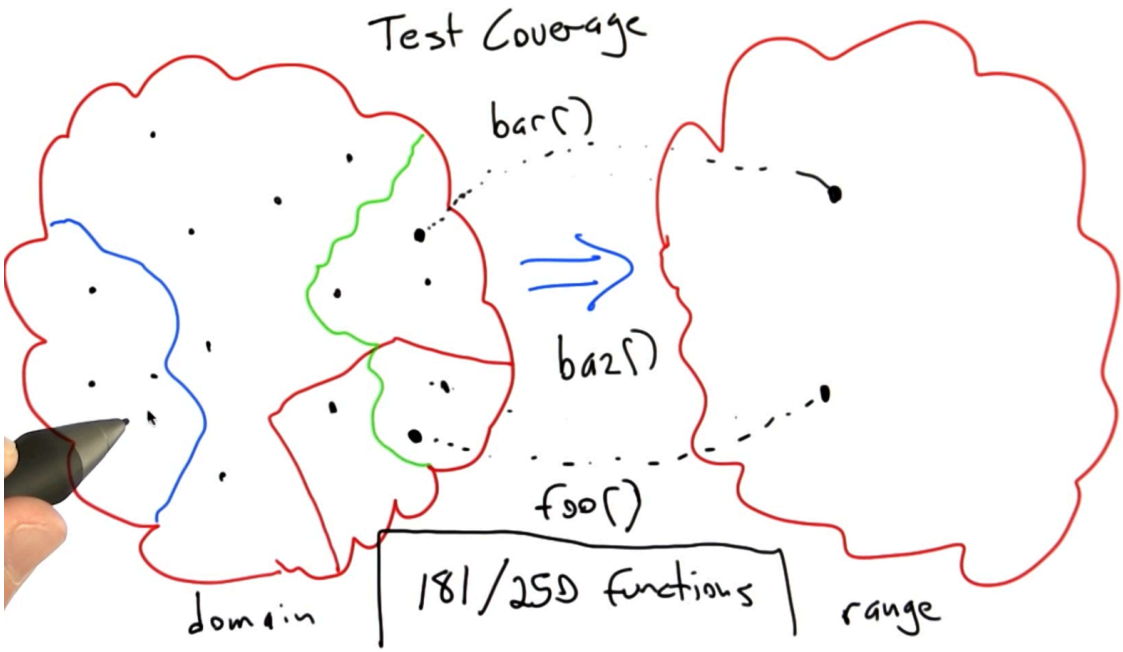
3. Partitioning the input domain

- We start with some SUT and it’s going to have a set of possible inputs, i.e. an input domain, and usually it consists of many possible test cases, that there is no way we can possibly test them all.
- People were often interested in ways to partition the input domain into a no. of different classes so that all of the points within each class are treated the same by the SUT.
- Let’s consider a subset of the input domain. For purposes of finding defects in the SUT, we pick an arbitrary point and execute the system on it. We look at the output, and if it is acceptable, then we’re done testing that class of inputs.
- In practice sometimes what we thought was a class of inputs that are all equivalent might be in different classes. We can blame the partitioning and the unfortunate fact is the original definition of this partitioning scheme didn’t give us good guidance in how to actually do the partitioning.



4. Coverage

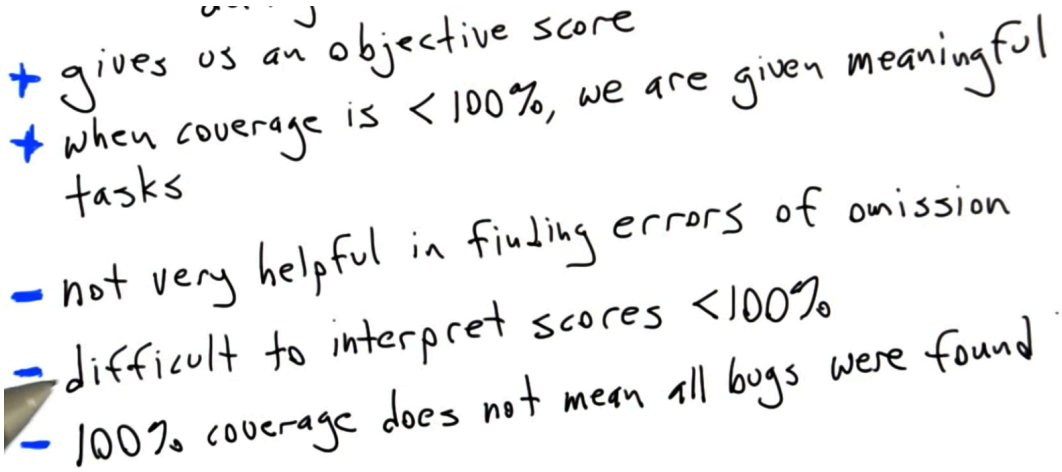
- In practice we ended up with the notion of test coverage instead of a good partitioning of the input domain.
- **Test coverage** is trying to accomplish exact the same thing that partitioning was accomplishing, but it goes about in a different way.
- Test coverage is an **automatic way of partitioning the input domain with some observed features of the source code.**
- One particular kind of test coverage is called **function coverage** and is achieved when every function in our source code is executed during testing.
- We subdivide the input domain for the SUT until we have split it into parts that results in every function being called.



- In practice, we start with a set of test of cases, and we run them all through the SUT. We see which functions are called and then we end up with some sort of a **score** called a **test coverage metric**.
- The score assigned to a collection of test cases is very useful.
  - For each of the functions that wasn't covered, we can go and look at it and we can try to come up with the test input that causes that function to execute.
  - If there is some function baz() for which we can't seem to devise an input that causes it to execute, then there are a couple of possibilities. One possibility is that it can't be called at all. It's **dead code**. Another possibility is that we simply don't understand our system well enough to be able to trigger it.

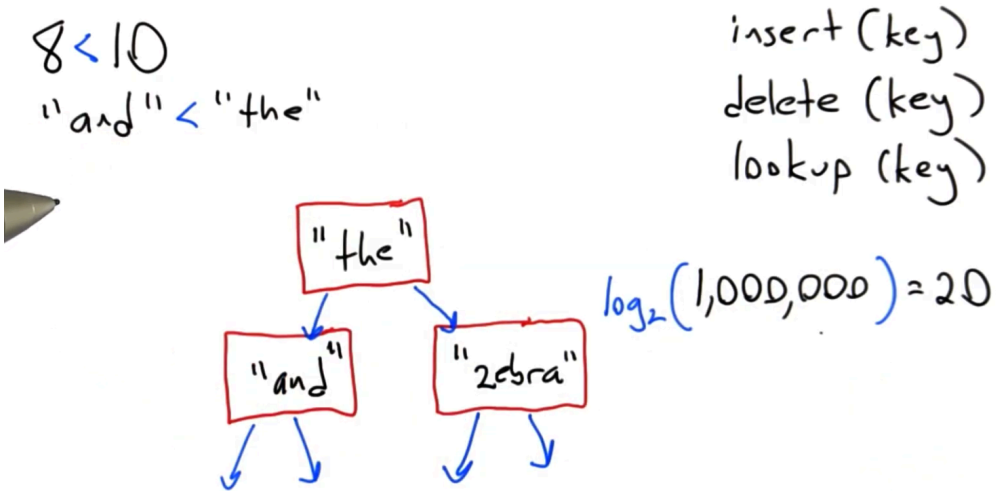
5. Test coverage

- Test coverage is **a measure of the proportion of a program exercised during testing**.



6. Splay tree

- Let's take a concrete look of what a coverage can do for us in practice.
- We'll look at some random open source Python codes that implements a splay tree, a kind of binary search tree.
- A binary search tree is a tree where every node has 2 leaves and it supports operations such as insert, delete, and lookup. The main important thing is the keys have to support an order in relation.
  - If we're using integers for keys then we can use < for order in relation.
  - If we're using words as our keys, then we can use dictionary order.



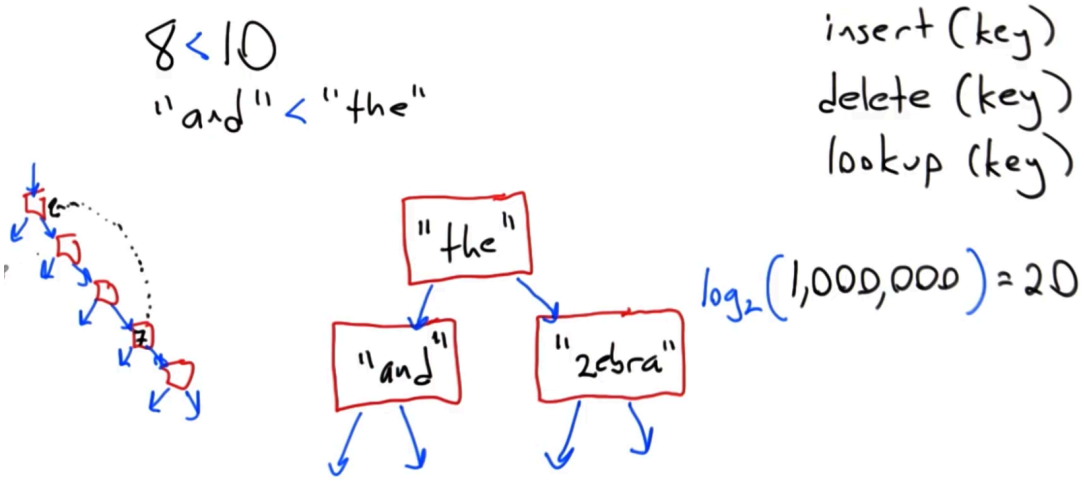
- The way the binary search tree is going to work is, we're going to build up a tree under the invariant that the left child of any node always has a key that's ordered before the

key of the parent node and the right child is always ordered after the parent node using the ordering.

- For a large tree with this kind of shape we have a procedure for fast lookup.
- The way that these trees are set up means that, on average, each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree, O(log n).

7. Splay tree issues

- Splay tree is a the simplest example of self-balancing binary search tree. As we add elements a procedure keeps the tree balanced so that tree operations remain very fast.
- It has a really cool property that when we access the nodes, let's say we do a lookup of this node which contains 7, what's going to happen is as a side-effect of the lookup that node is going to get migrated up to the root and then whatever was previously at the root is going to be pushed down and possibly some sort of a balancing operation is going to happen.
  - The point is, that frequently accessed elements end up being pushed towards the root of a tree and therefore, future accesses to these elements become even faster.



- In the following we will look at an open source splay tree implemented in Python, found on the web, that comes with its own test suite and we're going to look at what kind of code coverage this test suite gets on the splay tree.

8. Splay tree example

https://codereview.stackexchange.com/questions/209904/unit-testing-for-splay-tree-in-python

```
class Node:
      def __init__(self, key):
            self.key = key
            self.left = self.right = None

      def equals(self, node):
            return self.key == node.key

class SplayTree:
      def __init__(self):
            self.root = None
            self.header = Node(None)  # for splay()

      def insert(self, key):
            if (self.root == None):
                  self.root = Node(key)
            return

            self.splay(key)
            if self.root.key == key:
                  # If the key is already there in the tree, don't do
anything.
                  return

            n = Node(key)
            if key < self.root.key:
                  n.left = self.root.left
                  n.right = self.root
                  self.root.left = None
            else:
                  n.right = self.root.right
                  n.left = self.root
                  self.root.right = None
            self.root = n

      def remove(self, key):
            self.splay(key)
            if key != self.root.key:
                  raise 'key not found in tree'

            # Now delete the root.
            if self.root.left == None:
                  self.root = self.root.right
            else:
                  x = self.root.right
                  self.root = self.root.left
                  self.splay(key)
                  self.root.right = x
```

```
    def findMin(self):
        if self.root == None:
            return None
        x = self.root
        while x.left != None:
            x = x.left
        self.splay(x.key)
        return x.key

    def findMax(self):
        if self.root == None:
            return None
        x = self.root
        while x.right != None:
            x = x.right
        self.splay(x.key)
        return x.key

    def find(self, key):
        if self.root == None:
            return None
        self.splay(key)
        if self.root.key != key:
            return None
        return self.root.key

    def isEmpty(self):
        return self.root == None
```

- The splay operation moves a particular key up to the root of the binary search tree. This serves as both the balancing operation and also the look up:

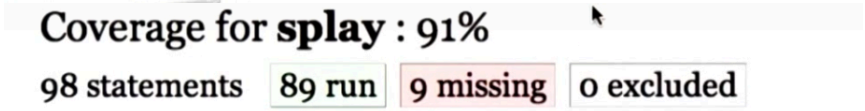
```
    def splay(self, key):
        l = r = self.header
        t = self.root
        self.header.left = self.header.right = None
        while True:
            if key < t.key:
                if t.left == None:
                    break
                if key < t.left.key:
                    y = t.left
                    t.left = y.right
                    r.right = t
                    t = y
                    if t.left == None:
                        break
                r.left = t
                r = t
                t = t.left
            elif key > t.key:
                if t.right == None:
                    break
                if key > t.right.key:
                    y = t.right
                    t.right = y.left
                    r.left = t
                    t = y
                    if t.right == None:
                        break
                l.right = t
                l = t
                t = t.right
            else:
                break
        l.right = t.left
        r.left = t.right
        t.left = self.header.right
        t.right = self.header.left
        self.root = t
```

- Now, let's look at the test suite:

```
import unittest  # module for unit testing
from splay_tree import SplayTree



class TestCase(unittest.TestCase):
    def setUp(self):
        self.keys = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        self.t = SplayTree()
        for key in self.keys:
            self.t.insert(key)

    def testInsert(self):
        for key in self.keys:
            self.assertEquals(key, self.t.find(key))

    def testRemove(self):
        for key in self.keys:
            self.t.remove(key)
            self.assertEquals(self.t.find(key), None)

    def testLargeInserts(self):
        t = SplayTree()
        nums = 40000
        gap = 307  # stress testing
        i = gap
        while i != 0:
            t.insert(i)
            i = (i + gap) % nums

    def testIsEmpty(self):
        self.assertFalse(self.t.isEmpty())
        t = SplayTree()
        self.assertTrue(t.isEmpty())

    def testMinMax(self):
        self.assertEquals(self.t.findMin(), 0)
        self.assertEquals(self.t.findMax(), 9)


if __name__ == "__main__":
    unittest.main()
```

- After running the tests we can use a **code coverage tool** to measure coverage. We can generate a HTML report. In our case, it's telling us that when we run the splay tree on its own unit test suite, out of the 98 statements in the file, 89 of them got run, 9 of them failed to run.
  `coverage erase ; coverage run splay_tree_test.py ; coverage html -i`

```
udacitys-imac:files udacity$ emacs splay.py
udacitys-imac:files udacity$ python splay_test.py
.....
----------------------------------------------------------------------
Ran 5 tests in 1.686s

OK
udacitys-imac:files udacity$ coverage erase ; coverage run splay_test.py ; coverage html -i
```




```
1  class Node:
2      def __init__(self, key):
3          self.key = key
4          self.left = self.right = None
5
6      def equals(self, node):
7          return self.key == node.key
8
9  class SplayTree:
10     def __init__(self):
11         self.root = None
12         self.header = Node(None) #For splay()
13
14     def insert(self, key):
15         if (self.root == None):
16             self.root = Node(key)
17             return
```

- The test suite failed to test the case where we inserted an element into the tree and it was already there.
- In the splay tree's removing function the first thing this function does is, splays the tree based on the key to be removed and so this is intended to draw that key up to the root note of the tree. If the root node of the tree does not have the key that we're looking for, then we're going to raise an exception saying that this key wasn't found. But this wasn't tested.
- If we look in the body of the delete function, we see a pretty significant chunk of code that wasn't tested, so we have to go back and revisit this.

- In conclusion, the tool is showing us what we didn't think to test with the unit test suite that we wrote so far.

9. Improving coverage

- We will modify in the test suite, testRemove function:

```
    def testRemove(self):
        for key in self.keys:
            self.t.remove(key)
            self.assertEquals(self.t.find(key), None)
        self.t.remove(-999)
```

- After running the tool the line that we just added causes an exception re-thrown in the splay function. So we have found a bug not anticipated by the developer of the splay tree.
- It often turns out that the stuff that we thought was going to run might be running only some of it. So the coverage tool told us something interesting.
- On the other hand, if the coverage tool hasn't told us anything interesting, i.e. everything we hoped was executed well when we run the unit test suite then that's good, too.
- Another thing we noticed is that the bug was somewhere completely different buried in the splay routine and if we go back and look at the coverage information, it turns out that the splay routine is entirely covered, i.e. every line of code was executed during the execution of the unit test for the splay tree.
- We deduce that just because some code was covered, especially at the statement level, this does not mean that it doesn’t contain a bug in it.
  - We have to ask the question, "What do we want to really read into the fact that we failed to cover something?” The coverage tool has given an example suggesting that our test suite is poorly thought out.
  - So, when coverage fails its better to try to think about why went wrong rather than just blindly writing a test case and just exercise the code which wasn't covered.

10. Problems with coverage

- We looked to an example where measuring coverage was useful in finding a bug in a piece of code. The coverage is not particularly useful in spotting the bug.
- In the following we will discuss about another piece of code, a broken function whose job is to determine whether a number is prime.

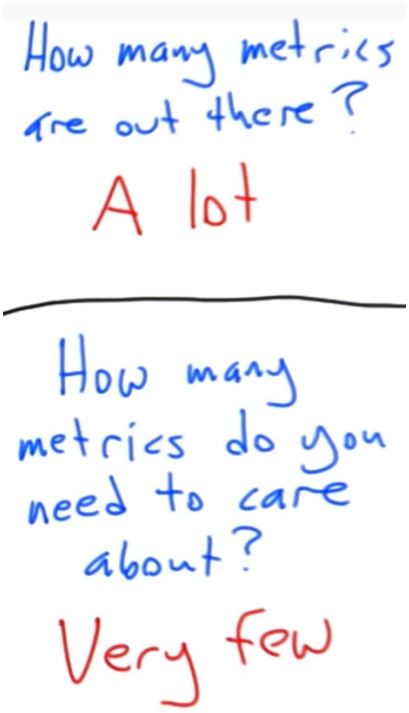
```
# CORRECT SPECIFICATION:
#
# isPrime checks if a positive integer is prime.
#
# A positive integer is prime if it is greater than
# 1, and its only divisors are 1 and itself.


import math

def isPrime(number):
    if number <= 1 or (number % 2) == 0:
        return False
    for check in range(3, int(math.sqrt(number))):
        if (number % check) == 0:
            return False
    return True

def check(n):
    print("isPrime(" + str(n) + ") = " + str(isPrime(n)))

check(1)
check(2)
check(3)
check(4)
check(5)
check(20)
check(21)
check(22)
check(23)
check(24)

def test():
    assert isPrime(1) == False
    assert isPrime(2) == False
    assert isPrime(3) == True
    assert isPrime(4) == False
    assert isPrime(5) == True
    assert isPrime(20) == False
    assert isPrime(21) == False
    assert isPrime(22) == False
    assert isPrime(23) == True
    assert isPrime(24) == False
```

- isPrime function has successfully identified whether the input is prime or not for the 10 numbers in the assertions.
- When we run the code coverage tool we can see out of the 20 statements in the file, all of them run, and none of them failed to be covered. Statement coverage gives us a perfect result for this particular code and yet another set is wrong.
- `coverage erase ; coverage run --branch prime.py ; coverage html -i`

```
# TASKS:
#
# 1) Add an assertion to test() that shows
#    isPrime(number) to be incorrect for
#    some input.
#
# 2) Write isPrime2(number) to correctly
#    check if a positive integer is prime.

# Note that there is an error where isPrime() incorrectly returns False for
an input of 2, despite 2 being a prime number.
# This has been fixed in the following.
# In isPrime function the range loops between 3 and the square of the input-1


import math

def isPrime2(number):
    if number == 2:
        return True
    if number <= 1 or (number % 2)==0:
        return False
    for check in range(3, int(math.sqrt(number)) + 1):
        if (number % check) == 0:
            return False
    return True

def test():
    assert isPrime(6) == False
    assert isPrime(7) == True
    assert isPrime(8) == False
    assert isPrime(9) == True
    assert isPrime(10) == False
    assert isPrime(25) == True
    assert isPrime(26) == False
    assert isPrime(27) == False
    assert isPrime(28) == False
    assert isPrime(29) == True

def test2():
    assert isPrime2(6) == False
    assert isPrime2(7) == True
    assert isPrime2(8) == False
    assert isPrime2(9) == False
    assert isPrime2(10) == False
    assert isPrime2(25) == False
    assert isPrime2(26) == False
    assert isPrime2(27) == False
    assert isPrime2(28) == False
    assert isPrime2(29) == True
```

- Why did test coverage fail to identify the bug? Statement coverage is a rather crude metric that only checks whether each statement executes once. Each statement executes at least once that lets a lot of bugs slip through.
- The lesson here is we should not let complete coverage plus a number of successful test cases fool as into thinking that a piece of code of right. It's often the case that deeper analysis is necessary.

11. Coverage metrics

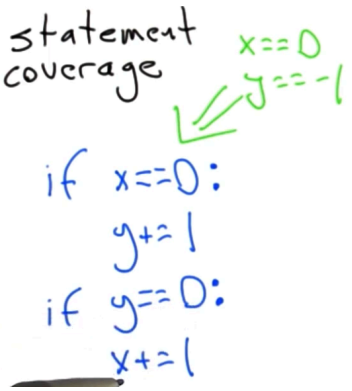


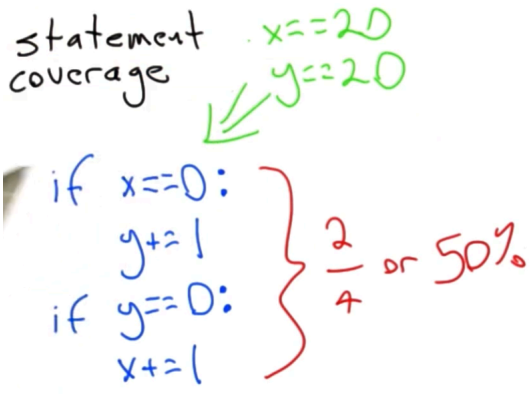
•There is a large number of test coverage metrics. An article lists **101 test coverage metrics**.

•Here we talk about a fairly small number of coverage metrics that matter for everyday programming life.

•The first is statement coverage measured by the Python test coverage tool where we looked at.

•Let's use a very simple 4 line codes and try to measure the statement coverage. Let's say we call this code with x=0 and y=-1. All 4 will be executed and this will give us a statement coverage of **100%**.





•Let's call this code with x=20 and y=20. Both tests will fail, and so, we will end up with a code coverage of 2/4 statements or **50%**.

- **Line coverage** is very similar to statement coverage but the metric is tied to actual physical lines in the source code. In this case, there is only one statement for each line so statement coverage and line coverage would be exactly identical.

- But on the other hand, if we decided to write some code that had multiple statements per line, line coverage would conflate them whereas statement coverage would consider them individual statements. For most practical purposes, these are very similar. So, statement coverage has a slightly finer granularity.

12. Testing coverage

- The function stats is defined to take a list of numbers as input and what the function does is computes the smallest element, the largest element, the median element and the mode, i.e. the element which occurs the most frequently in the list.

```
def stats(lst):
    min = None
    max = None
    freq = {}
    for i in lst:
        if min is None or i < min:
            min = i
        if max is None or i > max:
            max = i
        if i in freq:
            freq[i] += 1
        else:
            freq[i] = 1
    lst_sorted = sorted(lst)
    if len(lst_sorted) % 2 == 0:
        middle = len(lst_sorted)/2
        median = (lst_sorted[middle] + lst_sorted[middle-1]) / 2
    else:
        median = lst_sorted[len(lst_sorted)/2]
    mode_times = None
    for i in freq.values():
        if mode_times is None or i > mode_times:
            mode_times = i
    mode = []
    for (num, count) in freq.items():
        if count == mode_times:
            mode.append (num)
    print "list = " + str(lst)
    print "min = " + str(min)
    print "max = " + str(max)
    print "median = " + str(median)
    print "mode(s) = " + str(mode)

def test():
    # Change l to something that manages full coverage. You may
    # need to call stats twice with different input in order
    # to achieve full coverage.
    l = [31]
    stats(l)

test()
```

- We can see above that the bad test managed to cover 29 statements in the stats function.
- In the following we'll write a collection of test cases in order to get 100% statement coverage.

```
def test():
    l = [31,32,33,34]. # even number of elements
    stats(l)
    l = [31,32,33,33]. # odd number of elements
    stats(l)
test()
```

13. Fooling coverage

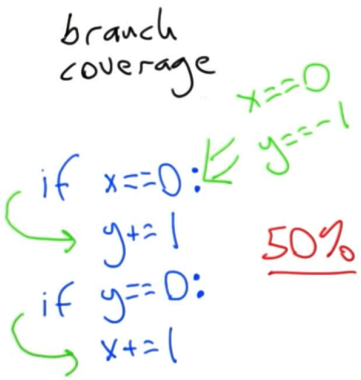
- Below we insert a bug into the stats module, i.e. make it wrong but in a way that's undetectable by test cases that get full statement coverage.
-

```
# TASK:

# Achieve full statement coverage on the stats function.
# All you should have to do is modify the test function
# to call stats with different lists of values.

# You will need to:
# 1) Insert a bug into the stats function.
# 2) Modify test1 so that it still achieves full test
#    coverage, but does not trigger your bug. Depending
#    on the bug you insert, you may not need to modify
#    test1 at all.
# 3) Write test2 so that it also achieves full test
#    coverage, but does trigger your bug.
```

```
import math

def stats(lst):
    min = None
    max = None
    freq = {}
    for i in lst:
        # i = abs(i)  => in test2 we consider l = [-33,-34]
        if min is None or i < min:
            min = i
        if max is None or i > max:
            max = 31  # bug
        if i in freq:
            freq[i] += 1
        else:
            freq[i] = 1
    lst_sorted = sorted(lst)
    if len(lst_sorted) % 2 == 0:
        middle = len(lst_sorted)/2
        median = (lst_sorted[middle] + lst_sorted[middle-1]) / 2
    else:
        median = lst_sorted[len(lst_sorted)/2]
    mode_times = None
    for i in freq.values():
        if mode_times is None or i > mode_times:
            mode_times = i
    mode = []
    for (num, count) in freq.items():
        if count == mode_times:
            mode.append (num)
    print "list = " + str(lst)
    print "min = " + str(min)
    print "max = " + str(max)
    print "median = " + str(median)
    print "mode(s) = " + str(mode)

# test1 should achieve full statement coverage of the stats function
# without triggering the bug you've inserted into the stats function.
def test1():
    ###Your test1 code here. Depending on what
    # bug you choose to put in the stats function,
    # you may or may not need to modify test1.
    l = [31, 31, 1, 2, 2, 1]
    stats(l)
    l = [31]
    stats(l)

# test2 should also achieve full statement coverage of the stats function,
# but should trigger the bug you've inserted into the stats function.
def test2():
    l = [31, 31, 1, 2, 2, 1]
    stats(l)
    l = [32]
    stats(l)

test1()
test2()
```

14. Branch coverage

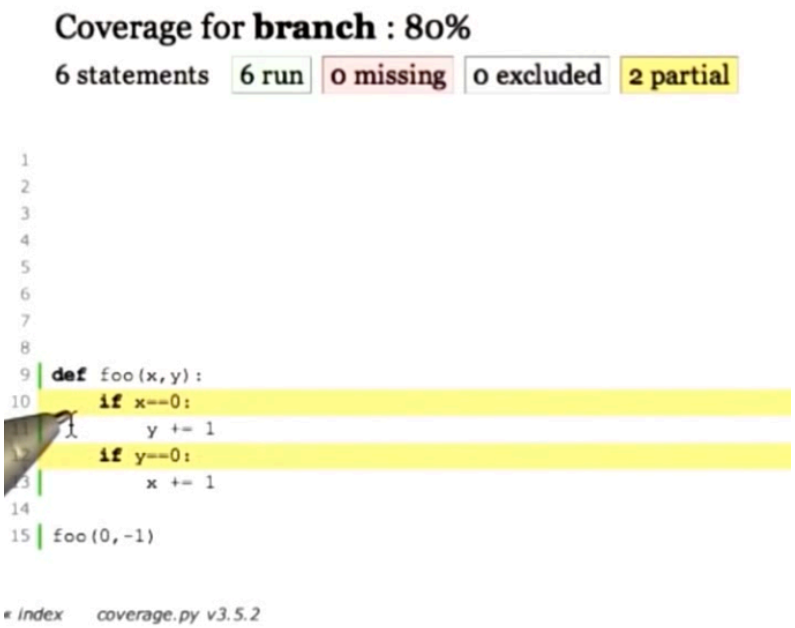
- We just talked about statement coverage, which is closely related to line coverage, but it's a bit more fine-grained, and now let's talk about what is probably the only other test coverage metric that will matter in your day-to-day life unless you go build avionics software.
- Branch coverage is a metric where a branch in a code is covered, if it executes both ways.
- In many cases, branch coverage and statement coverage have the same effect. For example, if our code only contained if-then-else loops, the metrics would be equivalent. On the other hand for code like on the right that's missing the else branches they're not quite equivalent.



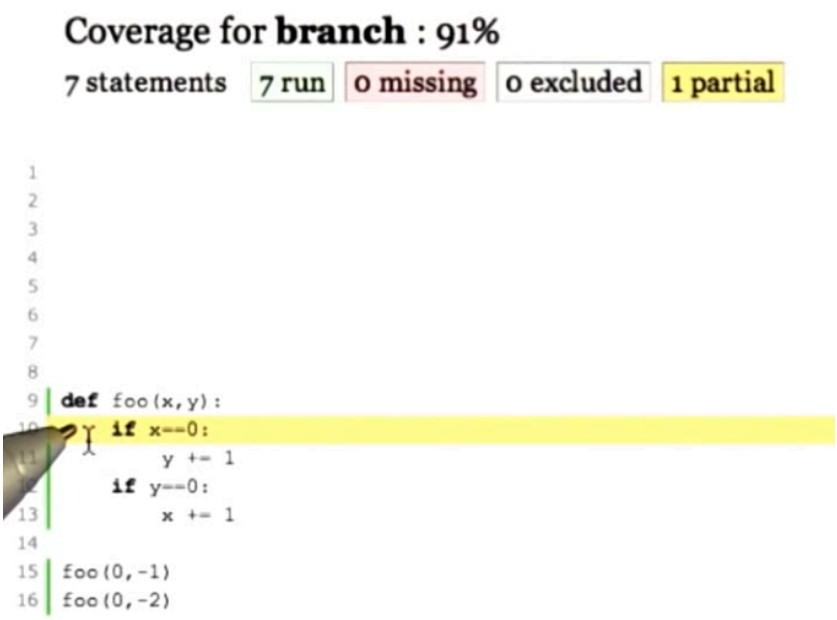
- These inputs are sufficient to get 100% statement coverage and 50% branch coverage:
- We're going run this under the coverage tool, but this time we're going to give the coverage run command an argument --branch:

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

- That simply tells it to measure branch coverage instead of just measuring statement coverage.

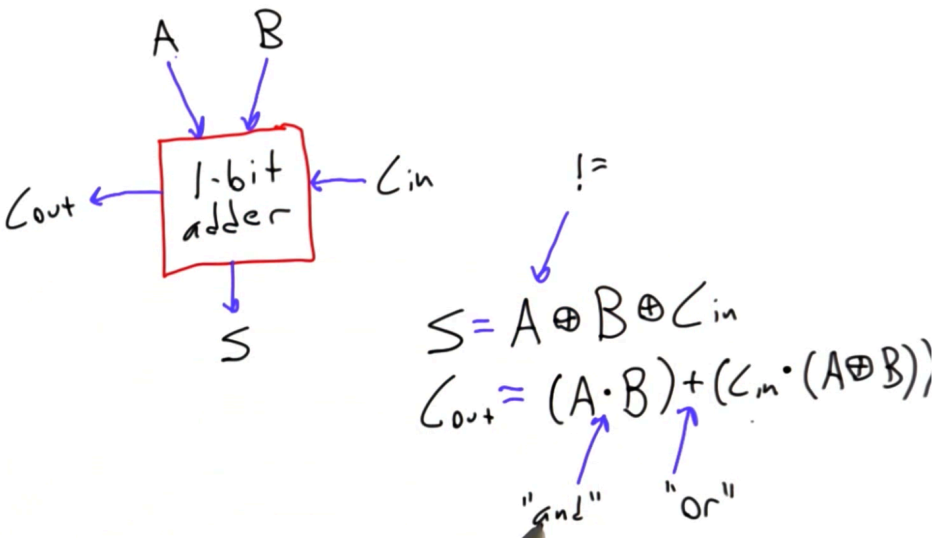


- If we call foo a second time with 0 and -2, we get:

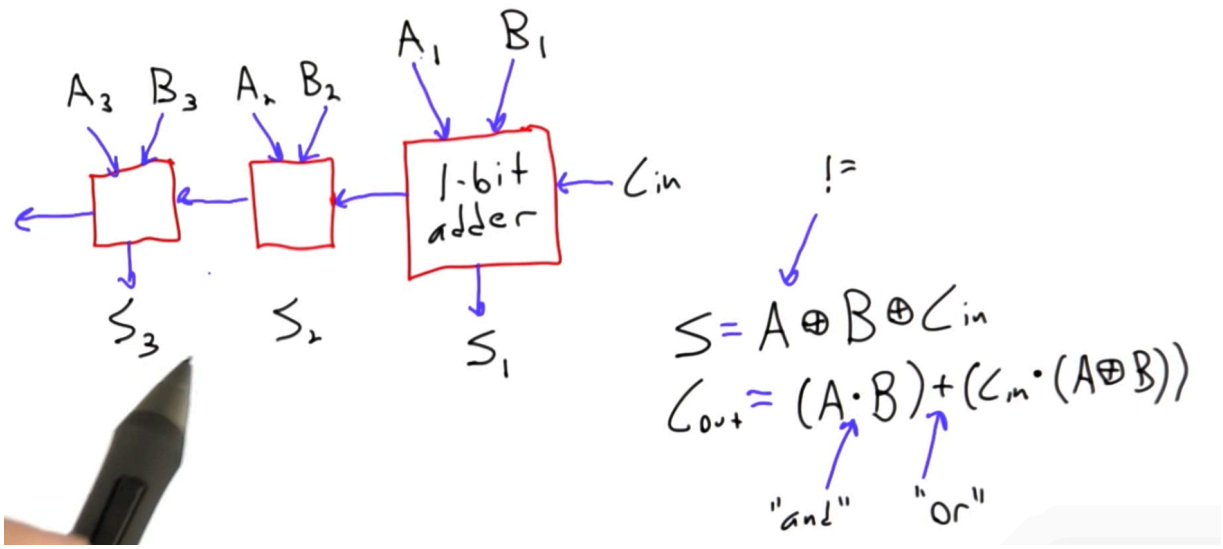


15. 8 Bit adder

- Let's consider an example of Python code that simulates some adders, i.e. simple hardware modules that perform addition.
- The way that 1-bit adder works, where A and B are 2 input bits, C in is the carry into the column, C out is the carry out of the column, S is the sum, ⊕ is XOR operator implemented in Python using the ! Operator:



- An actual adder that corresponds to the add instruction that you would find in an instruction set for a real computer looks like in the following:



- Python code implementing an 8-bit adder:

```
## SPECIFICATION:
#
# add8 emulates an 8-bit hardware adder.
# it takes 17 bits, representing two 8-bit
# numbers and a carry bit.
#
# TASK:
#
# Write test() such that it achieves 100%
# branch coverage of the add8 function.

def add8(a0,a1,a2,a3,a4,a5,a6,a7,b0,b1,b2,b3,b4,b5,b6,b7,c0):
    s1 = False
    if (a0 != b0) != c0:
        s1 = True
    c1 = False
    if (a0 and b0) != (c0 and (a0 != b0)):
        c1 = True
    s2 = False
    if (a1 != b1) != c1:
        s2 = True
    c2 = False
    if (a1 and b1) != (c1 and (a1 != b1)):
        c2 = True
    s3 = False
    if (a2 != b2) != c2:
        s3 = True
    c3 = False
    if (a2 and b2) != (c2 and (a2 != b2)):
        c3 = True
    s4 = False
    if (a3 != b3) != c3:
        s4 = True
    c4 = False
    if (a3 and b3) != (c3 and (a3 != b3)):
        c4 = True
```

```
    s5 = False
    if (a4 != b4) != c4:
        s5 = True
    c5 = False
    if (a4 and b4) != (c4 and (a4 != b4)):
        c5 = True
    s6 = False
    if (a5 != b5) != c5:
        s6 = True
    c6 = False
    if (a5 and b5) != (c5 and (a5 != b5)):
        c6 = True
    s7 = False
    if (a6 != b6) != c6:
        s7 = True
    c7 = False
    if (a6 and b6) != (c6 and (a6 != b6)):
        c7 = True
    s8 = False
    if (a7 != b7) != c7:
        s8 = True
    c8 = False
    if (a7 and b7) != (c7 and (a7 != b7)):
        c8 = True
    return (s1,s2,s3,s4,s5,s6,s7,s8,c8)
```

- Let's say this code is part of some sort of a circuit simulation, and we want to test the validity of our circuit, therefore we pass it actual numbers. We can write some little support functions that take integers and convert them into the bit format.
- In the following there is a solution using exhaustive testing:

```
def split (n):
    return (n&0x1,n&0x2,n&0x4,n&0x8,n&0x10,n&0x20,n&0x40,n&0x80)

def glue (b0,b1,b2,b3,b4,b5,b6,b7,c):
    t = 0
    if b0:
        t += 1
    if b1:
        t += 2
    if b2:
        t += 4
    if b3:
        t += 8
    if b4:
        t += 16
    if b5:
        t += 32
    if b6:
        t += 64
    if b7:
        t += 128
    if c:
        t += 256
    return t

def myadd (a,b):
    (a0,a1,a2,a3,a4,a5,a6,a7) = split(a)
    (b0,b1,b2,b3,b4,b5,b6,b7) = split(b)
    (s0,s1,s2,s3,s4,s5,s6,s7,c) =
add8(a0,a1,a2,a3,a4,a5,a6,a7,b0,b1,b2,b3,b4,b5,b6,b7,false)
    return glue (s0,s1,s2,s3,s4,s5,s6,s7,c)

def testExhaustive():
    for i in range(256):
        for j in range(256):
            res = myadd(i,j)
            print str(i) + " " + str(j) + " " + str(res)
            assert res == (i+j)

testExhaustive()
```

- When running

  `coverage erase ; coverage run --branch 8bits-adder.py ; coverage html -i`

  the coverage output is:

  Coverage for **adder** : 100%

  85 statements | 85 run | 0 missing | 0 excluded | 0 partial

- Now let's look at an alternative solution. Instead of our exhaustive test, we could have written a much smaller test that gets 100% branch coverage.

```
def smallTest():
    myadd(0,0);
    myadd(0,1);
    myadd(255,255);

smallTest()
```

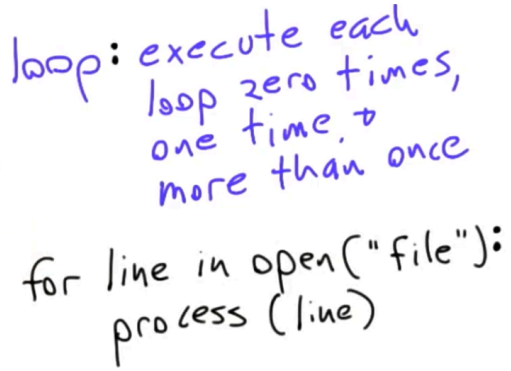
Coverage for **adder** : 100%

81 statements	81 run	0 missing	0 excluded	0 partial

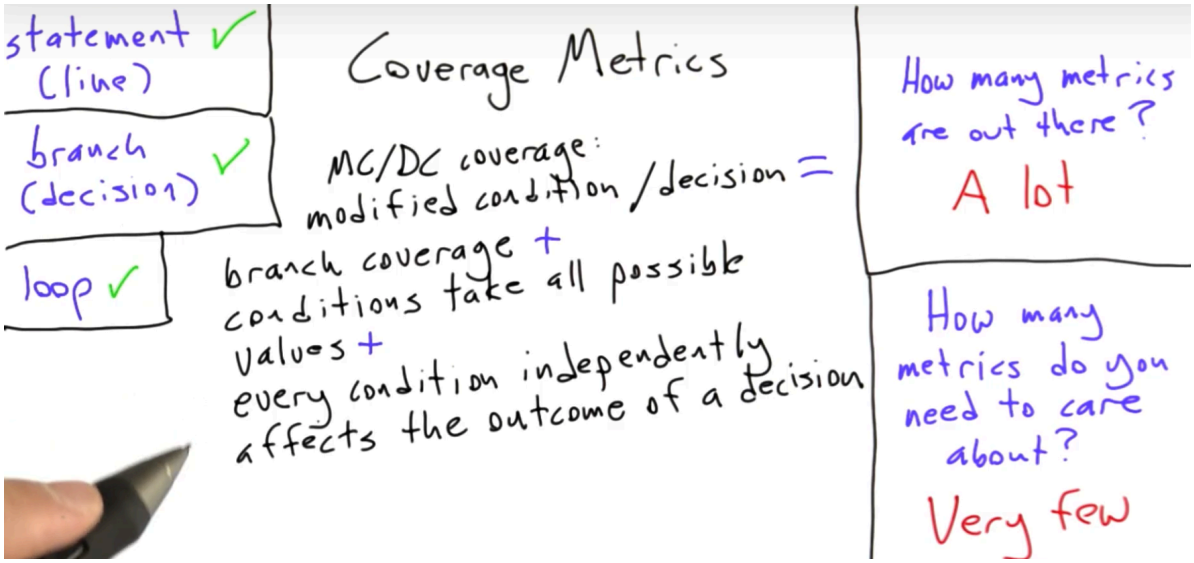
- So, as you can see, the coverage tool believes that just these 3 really simple test cases were sufficient to get 100% branch coverage of our adder.

16. Other metrics

- We looked at a couple common coverage metrics that come up in practice
  - **Statement coverage**, which is close relative to line coverage,
  - **Branch coverage**, also called decision coverage.
- These are the coverage metrics that are going to matter for everyday life.
- There are many other coverage metrics, and we're going to just look at a few of them not because we're going to go out and obsessively get 100% coverage on our code on all these metrics, but rather because they form part of the answer to the question **how shall we come up with good test inputs in order to effectively find bugs in our software**?
  - **Loop coverage** specifies that we execute each loop 0 times, once, and more than once. In the example on the right, to get full loop coverage we would need to test this code using a file that contains no lines, using a file that contains just one line, and using a file that contains multiple lines.

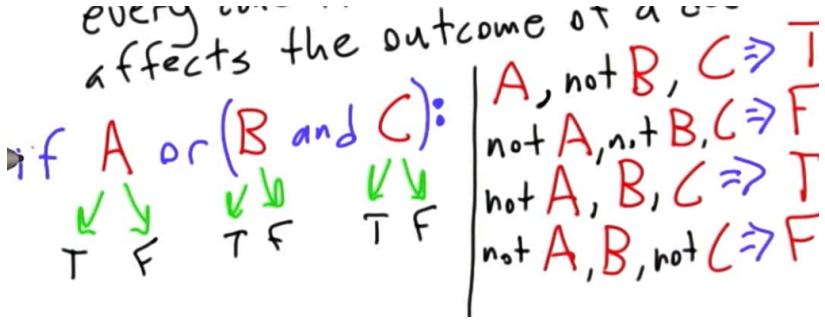


  - **Modified condition decision coverage** or MC/DC starts off with a branch coverage, it additionally states that every condition involved in a decision takes on every possible outcome:



17. MC/DC coverage

- To get full MC/DC coverage of the next Python conditional statement, we need to test each of the variables so that every condition independently affects the outcome:

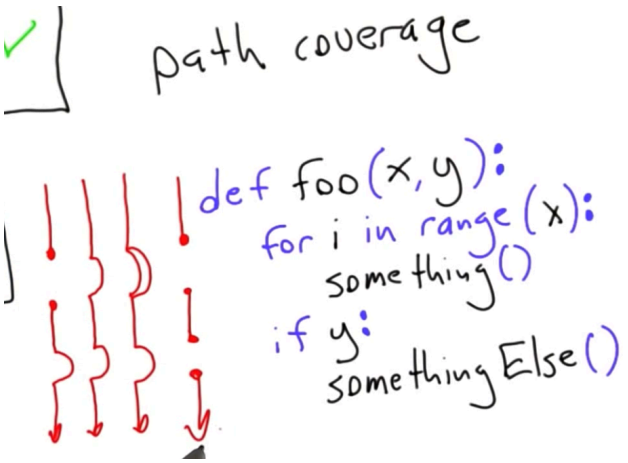


- We have above a minimal - or if not minimal, at least fairly small - set of test cases that together gets 100% MC/DC coverage.
- We could've written a conditional much more complicated, and if we had, we probably would've had a fairly hard time reasoning this stuff out by hand, and what we would've needed to do in that case is probably draw out a full truth table.
- The domain of interest for MC/DC coverage, i.e. embedded control systems that happen to be embedded in avionics systems end up having generally lots of complicated conditionals.
- We lack good tool support for MC/DC coverage in Python.

18. Path coverage

- It cares about how you got to a certain piece of code.

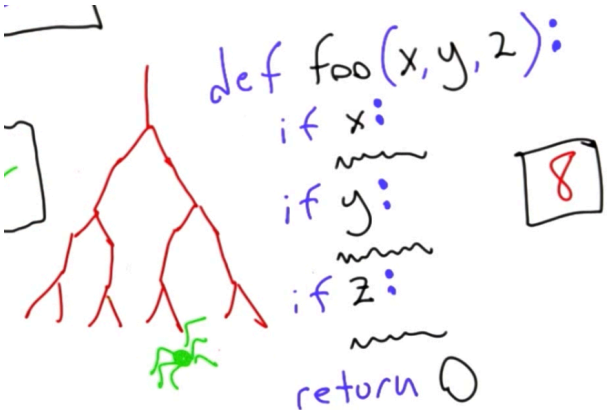
- Statement coverage and branch coverage, and even to a large extent MC/DC coverage and loop coverage, don't really care how you got somewhere as long as you executed the code in such a way that you met the conditions.



•A **path through a program** is a sequence of decisions made by operators in the program.

•Suppose we have a function foo and we will try to visualize the decisions made by the Python language. Let's consider x = 0, y = true, then (1, true). As x increases in value, we get more and more and more paths. There's going to be a similar family of paths for y is false.

- By changing x how many paths can we get through the code? The answer is that it's unlimited, so **achieving path coverage is going to be impossible for all real code**.
- Path coverage is basically an ideal that we'd like to approach if we want to do a good job testing. It's not going to be something that we can actually achieve.
- Suppose we have another function foo. How many paths through this code are?

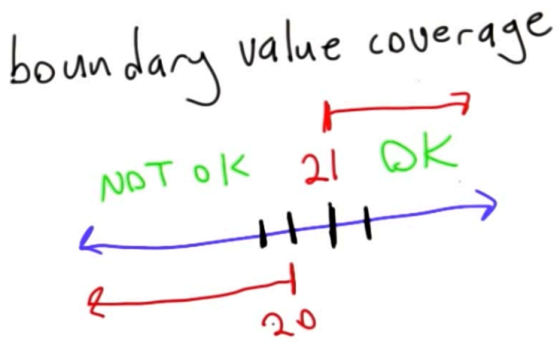


- We need path coverage to find a certain bug. This is a valuable weapon to have in our testing arsenal, even if we're not going to be achieving path coverage in practice.

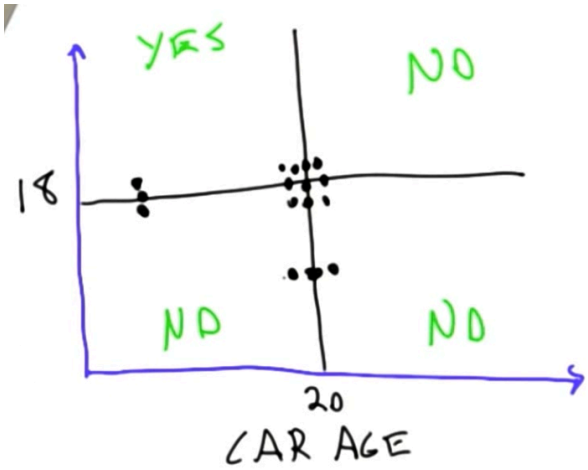
19. Boundary value coverage

- Unlike some of the other coverage measures we discussed in this lesson, doesn't have any specially take technical definition.
- What boundary value coverage basically says is when a program depends on some numerical range and when the program has different behaviors based on numbers within that range then we should test numbers close to the boundary.

- Let's take the example where we're writing a program to determine whether somebody who lives in the USA is permitted to drink alcohol.
- We want to get the boundary value coverage on this program, so we want to include the ages of 20 and 21 in our test input and possibly also of 19 and 22 close enough to the boundary values that there may be interesting behaviors looking at.



- We framed the boundary value coverage as a function of only one variable in terms of the program specification not in terms of the implementation.
- Let's consider a program with 2 inputs. Assume that these inputs are treated independently by the software that we are running. The first input is going to be the age of a car. An insurance company declines to insure cars more than 20 years old. The other parameter is the age of the driver. Drivers who are less than 18 years old will be declined to insure.
- If we have specific knowledge of our implementation we consider these variables together therefore we probably also need to test combinations of inputs



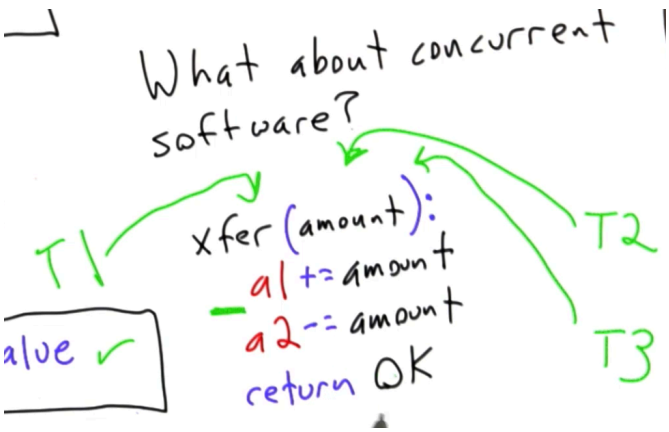
- As the number of inputs of the program goes up the number of test cases can grow very large because we have to consider the interaction between all possible

combinations of variables that are dependent. On the other hand, if our variables are independent then we can test this separately.

- Issue: we are doing **boundary value coverage with respect to the requirements** of the specification purpose of software or whether we are doing it with respect to the **implementation**.
- Recall the stats function described before where we inserted the bug i = abs(i) which causes it to misbehave for some inputs and not for others.
- We find a collection of test cases to get good coverage for the absolute value of the inputs when we pass numbers into the function which contain negative values.
- Considering the implementation not just the specification, what will happen is a function like absolute value would change its behavior around 0 and so what we need to call it with at least one negative number.
- In the stats function we have a lot of different operators with different behaviors around certain boundaries and so to get good boundary value coverage over all would be probably extremely difficult.
- There aren't good tools automating boundary value coverage in Python. There are techniques such as **mutation testing** that can automate boundary value coverage in some forms.

20. Concurrent software

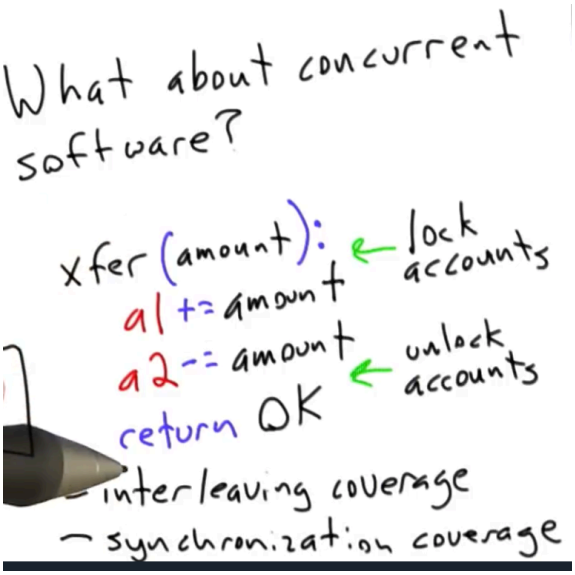
- We haven't been dealing at all with testing of concurrent code and this mainly it is a difficult and specialized skill.
- It's clear that applying sequential code coverage metrics to concurrent software is a fine idea, but these aren't going to give us any confidence with the code lacks concurrency such as race condition and the deadlocks.
- Let's take, for example, a function xfer, which transfer some amount of money between bank account one and bank account two.
- This particular function is designed to be called from different threads.
- The transfer function does not synchronise, i.e. it hasn't taken any sort of a lock while it manipulates the accounts. If these threads are operating on the same accounts concurrently, then it's going to be a problem.



- What sort of coverage what we are looking for while testing this function in order to detect this kind of bug? We want to make sure we tested the case where the transfer account is concurrently call.

21. Synchronization coverage

- The likely fix for the bug that we had in xfer function is to lock both of the bank accounts that we're processing, transfer the balance between them, and then unlock the accounts.
- We can delete all of the locks out of a code, run it through some tests, and often it passes. This is in spite of the test coverage metric called **synchronization coverage,** which ensures that during testing this lock actually does something.



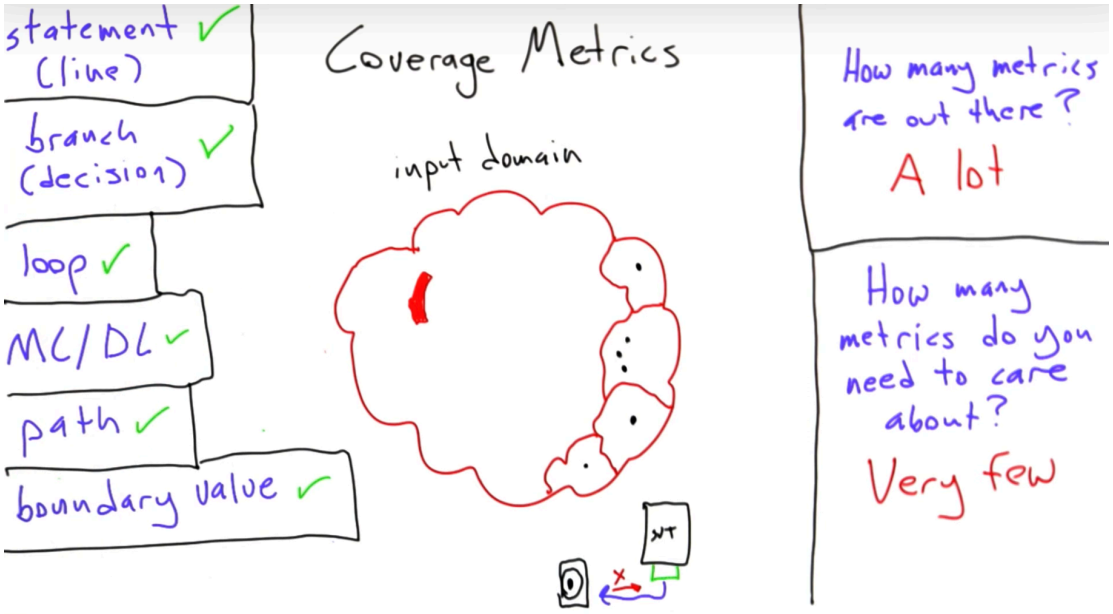
- In other words, during testing, the xfer function is going to be called to transfer money between accounts when the accounts are already locked and this ensures that we're stressing the system to a level that the synchronization code is actually firing.
- **Interleaving coverage** means if we recall functions which accessed shared data are actually called and in a truly concurrent fashion, i.e. by multiple threads at the same time.

22. When coverage doesn’t work

- What coverage is letting us do is divide up the input domain into regions in such a way, for any given region, any test input in that region, will accomplish some specific coverage task.
- Any input that we select within this particular region of the input domain will cause a particular statement or line to execute, cause the branch to execute in some specific direction, execute a loop zero times, one time, or more, etc.
- The obvious problem is, if we partition the input domain this way and we go ahead and test, it is easier to reach the region in the input domain, and get good coverage, but **we won’t be able to find errors of omission.**
- For example, the SUT is creating files on the hard disc. One extremely possible kind of bug that we would put into the SUT is failing to check error codes that could be

returned from file creation operations that happen when the disc is full or when there is a hard disc failure or something like that.

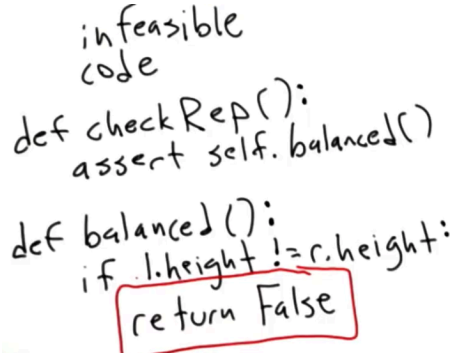
- Coverage metrics are not particularly good at discovering errors of omission like missing error checks.
- We discussed fault injection where we make the disc fail, we will make it send something bad up to the system and we see what happens, and in that case, if an error check is missing, then the system should actually do the wrong thing and we will be able to discover this by watching the system misbehave.



- Another thing we could have done is partition the input domain in a different way, i.e. not partition the input domain using an automated coverage metrics rather using the specification.
- In conclusion, there are multiple ways of partitioning the input domain for purposes of testing.

23. Infeasible code

- What does it mean when we have code that doesn't get covered, for example, if we're using statement coverage, **what happens when we have some statements that we haven't been able to cover?**
- There're basically **3 possibilities**. **The first possibility is infeasible code.**
- Let's say that we're writing the checkRep function for a balanced tree data structure. See open source code in Python for AVL tree at https://github.com/



majek/dump/blob/master/avl/avl.py-orig

- Assuming that the code is not bugged and assuming that we're testing a correct tree, we'll never going to be able to return false from our balanced function.
- A coverage tool is going to tell that we failed to achieve code coverage for this particular statement in the code. We have to ask ourselves, is that a bad thing? It's not bad because that code only can execute if we made a mistake somewhere. So, the proper response to this kind of situation is, we need to t**ell our coverage tool that we believe this line to be infeasible** and then the tool won't count this line when we're measuring code coverage.
- The code comes with its own test suite so let's run it under the coverage monitor using `coverage erase ; coverage run --branch avl_tree.py ; coverage html`

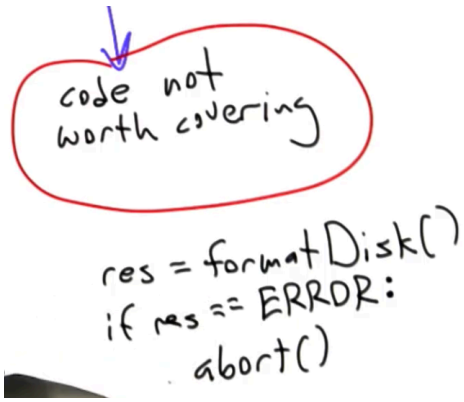
Coverage for **avltree** : 86%

389 statements	328 run	61 missing	0 excluded	20 partial

- Every statement that raises an exception has failed to have been covered. So superficially, we haven't gotten very good coverage of this function, but actually, what we hope is that this AVL tree code is correct and these are truly infeasible. Therefore, we can tell the coverage tool to ignore them using a comment that has a special form:
  `# pragma: no cover`
- When we run the coverage tool again things look a little bit better.

24. Code not worth covering

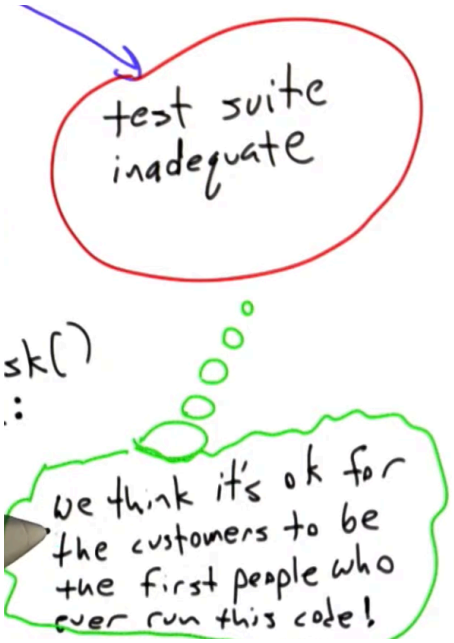
- **The second case** is the code that we believe to be feasible but which isn't worth covering because it's very hard to trigger and it's very simple.



•As an example, let's consider res variable the result of the command to format a disc:

•Might be the case that we lack an appropriate fault injection tool that will let us easily simulate the failure of a format disc call. Furthermore, the abort code, which is going to terminate the entire application, is presumably something that was tested elsewhere.

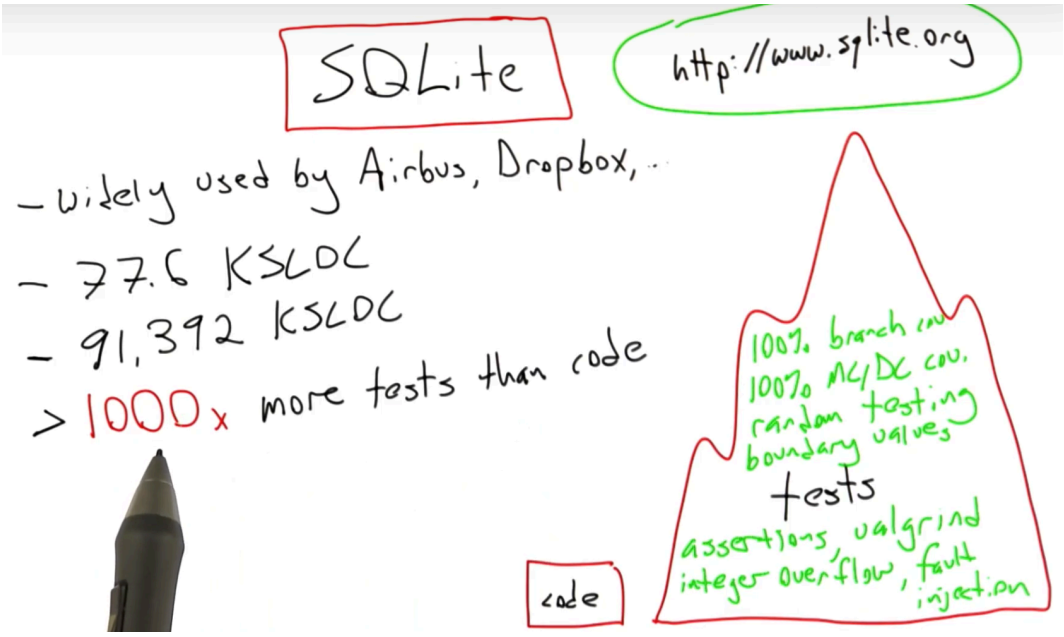
25. Inadequate test suite



- **The third reason t**he code might not have been covered is that the **test suite** might simply just be **inadequate**, and in that case we have to decide what to do about it.
- One option of course is to improve the test suites so that the code gets covered. Since it can be difficult, we might decide to ship our software without achieving a 100% code coverage, because like all forms of testing, is basically just a cost-benefit tradeoff.

26. Sqlite

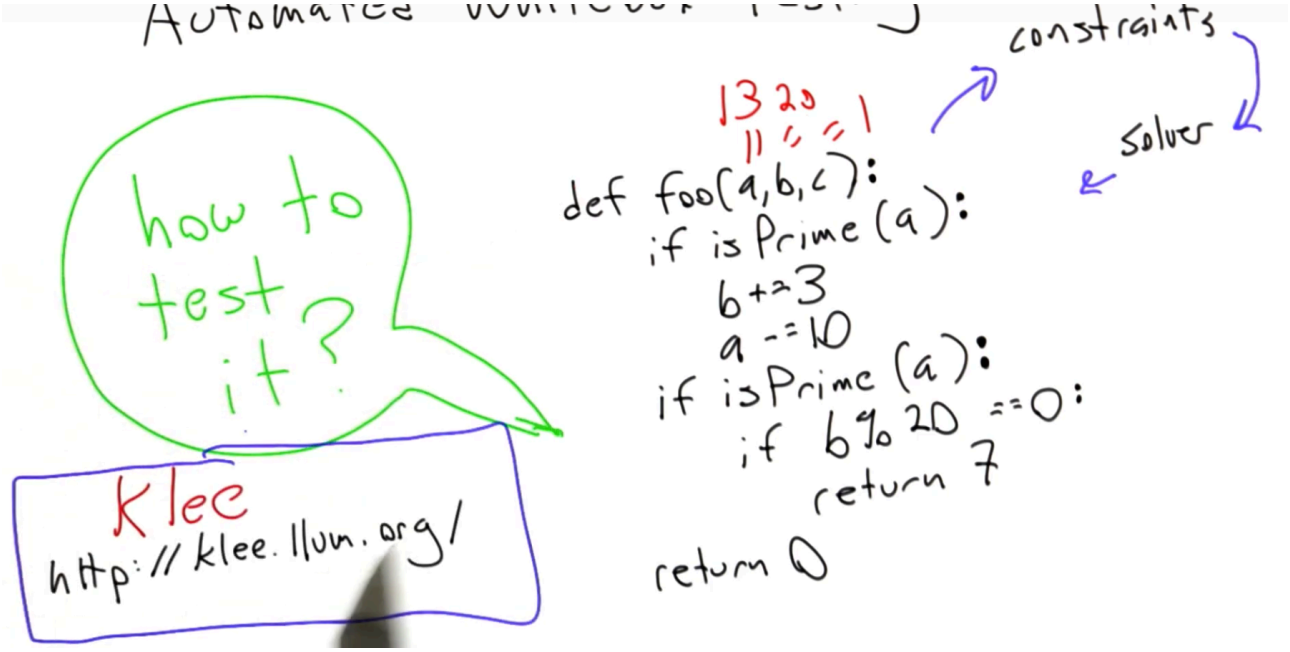
- An example of testing done right is an open source database called SQLite. This small open source database is designed to be easily embeddable in other applications.
- It's very widely used by companies like Airbus, Dropbox, in various Apple products, Android phones, etc.



- Almost every single testing technique presented in this lesson and many of the coverage metrics have been applied to SQLite, and the result is generally, a really solid piece of software.

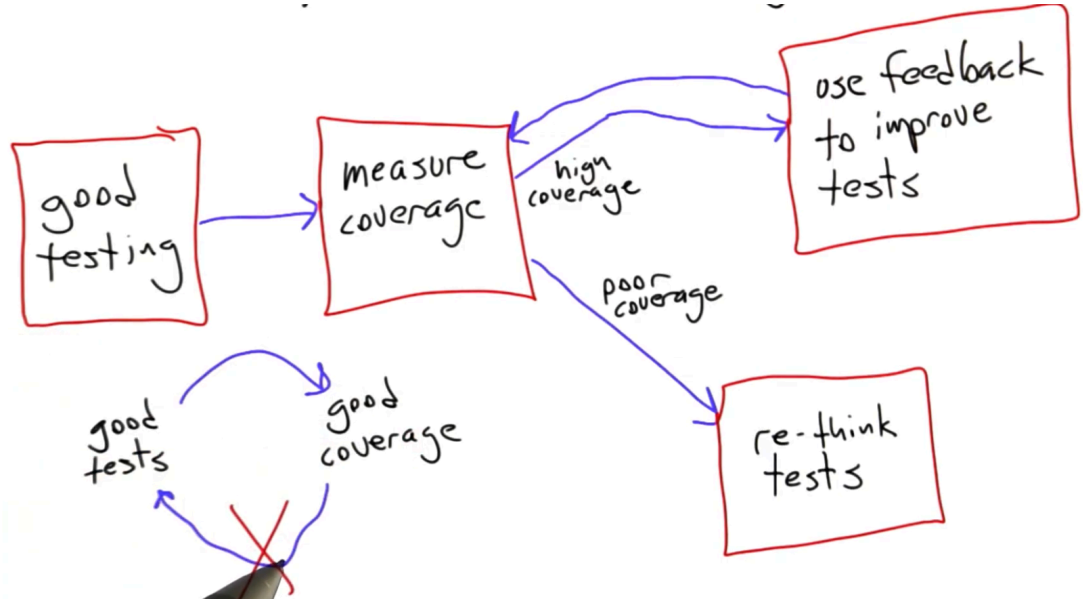
27. Automated white box testing

- Is not a form of code coverage but what rather a way to get software tools to automatically generate test for our code.



- The goal for this tool is to generate good path coverage for the code.
- Therefore, a tool will iterate the process of generating inputs that take different branches multiple times and then using what it learned about the code build up a set of constraints to explore different paths, pass it to the solver and the solver is either going to succeed in coming up with a new value or possibly it will fail.
- There are no automated whitebox testing tools for Python.
- For C programs there is a tool called Klee.
- Microsoft uses these techniques of automatically generating good test inputs for finding a very large number of bugs in real products.

28. How to use coverage



- We strongly believe that if we have a good test suite, and we measure it's coverage, the coverage will be good. We do not believe, on the other hand, that if we have a test suite which gets good coverage, it must be a good test suite.
- Used in the right way, coverage can be a relatively low cost way To improve the testing that we do for a piece of software.
- Used incorrectly, it can waste our time, and perhaps worst, lead to a false sense of security.

Code coverage

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

1. Queue coverage

- In the test function we call checkRep function after every single test that we do, just as a further test. Any of the test that we do that call the internals of the queue class should probably be in checkRep, because we don't necessarily want our test code to have to worry about the internals of it, we want the internal testing to be taken care of by the class itself.

```
# TASK: Achieve full statement coverage on the Queue class.

import array

class Queue:
    def __init__(self, size_max):
        assert size_max > 0
        self.max = size_max
        self.head = 0
        self.tail = 0
        self.size = 0
        self.data = array.array('i', range(size_max))

    def empty(self):
        return self.size == 0

    def full(self):
        return self.size == self.max

    def enqueue(self, x):
        if self.size == self.max:
            return False
        self.data[self.tail] = x
        self.size += 1
        self.tail += 1
        if self.tail == self.max:
            self.tail = 0
        return True

    def dequeue(self):
        if self.size == 0:
            return None
        x = self.data[self.head]
        self.size -= 1
        self.head += 1
        if self.head == self.max:
            self.head = 0
        return x
```

```
    def checkRep(self):
        assert self.tail >= 0
        assert self.tail < self.max
        assert self.head >= 0
        assert self.head < self.max
        if self.tail > self.head:
            assert (self.tail-self.head) == self.size
        if self.tail < self.head:
            assert (self.head-self.tail) == (self.max-self.size)
        if self.head == self.tail:
            assert (self.size==0) or (self.size==self.max)

def test():
    q = Queue(2)
    assert q
    q.checkRep()

    empty = q.empty()
    assert empty
    q.checkRep()

    full = q.full()
    assert not full
    q.checkRep()

    result = q.dequeue()
    assert result == None
    q.checkRep()

    result = q.enqueue(10)
    assert result == True
    q.checkRep()

    result = q.enqueue(20)
    assert result == True
    q.checkRep()

    empty = q.empty()
    assert not empty
    q.checkRep()

    full = q.full()
    assert full
    q.checkRep()

    result = q.enqueue(30)
    assert result == False
    q.checkRep()

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

    result = q.dequeue()
    assert result == 20
    q.checkRep()

test()
```

- It doesn't really take a huge amount of code to get full coverage of the queue class. If fact, what is done here is probably a bit much. It doesn't necessarily tell us a whole lot about whether the queue class is, in fact, correct in any meaningful way.

2. Splay tree coverage

- In the previous implementation there was a bug. Remove and splay functions are updated.

```
# TASK:
# We didn't achieve full statement coverage before, but we will now.

class Node:
    def __init__(self, key):
        self.key = key
        self.left = self.right = None

    def equals(self, node):
        return self.key == node.key

class SplayTree:
    def __init__(self):
        self.root = None
        self.header = Node(None) #For splay()

    def insert(self, key):
        if (self.root == None):
            self.root = Node(key)
            return

        self.splay(key)
        if self.root.key == key:
            # If the key is already there in the tree, don't do anything.
            return

        n = Node(key)
        if key < self.root.key:
            n.left = self.root.left
            n.right = self.root
            self.root.left = None
        else:
            n.right = self.root.right
            n.left = self.root
            self.root.right = None
        self.root = n
```

```
    def remove(self, key):
        self.splay(key)
        # if key != self.root.key:
        #     raise 'key not found in tree'
        if self.root is None or key != self.root.key:  # update
            return

        # Now delete the root.
        if self.root.left== None:
            self.root = self.root.right
        else:
            x = self.root.right
            self.root = self.root.left
            self.splay(key)
            self.root.right = x

    def findMin(self):
        if self.root == None:
            return None
        x = self.root
        while x.left != None:
            x = x.left
        self.splay(x.key)
        return x.key

    def findMax(self):
        if self.root == None:
            return None
        x = self.root
        while (x.right != None):
            x = x.right
        self.splay(x.key)
        return x.key

    def find(self, key):
        if self.root == None:
            return None
        self.splay(key)
        if self.root.key != key:
            return None
        return self.root.key

    def isEmpty(self):
        return self.root == None
```

```
    def splay(self, key):
        l = r = self.header
        t = self.root
        if t is None: # update
            return  # update
        self.header.left = self.header.right = None
        while True:
            if key < t.key:
                if t.left == None:
                    break
                if key < t.left.key:
                    y = t.left
                    t.left = y.right
                    y.right = t
                    t = y
                    if t.left == None:
                        break
                r.left = t
                r = t
                t = t.left
            elif key > t.key:
                if t.right == None:
                    break
                if key > t.right.key:
                    y = t.right
                    t.right = y.left
                    y.left = t
                    t = y
                    if t.right == None:
                        break
                l.right = t
                l = t
                t = t.right
            else:
                break
        l.right = t.left
        r.left = t.right
        t.left = self.header.right
        t.right = self.header.left
        self.root = t

def test():
    s = SplayTree()
    current_min = None
    current_max = None

    empty = s.isEmpty()
    assert empty == True
    _min = s.findMin()
    assert _min == None
    _max = s.findMax()
    assert _max == None

    found = s.find(10)
    assert found == None
```

```
    s.insert(100)
    current_min = 100
    current_max = 100

    for i in range(10,20):
        empty = s.isEmpty()
        assert empty == False

        s.insert(i)
        s.insert(i)

        if not current_min or i < current_min:
            current_min = i
        if not current_max or i > current_max:
            current_max = i

        found = s.find(i)
        assert found == i

        _min = s.findMin()
        assert _min == current_min

        _max = s.findMax()
        assert _max == current_max

    for i in range(10,20):
        empty = s.isEmpty()
        assert empty == False

        s.remove(i)
        s.remove(i)

        found = s.find(i)
        assert found == None

    s.insert(373)
    s.remove(373)

test()
```

Random testing

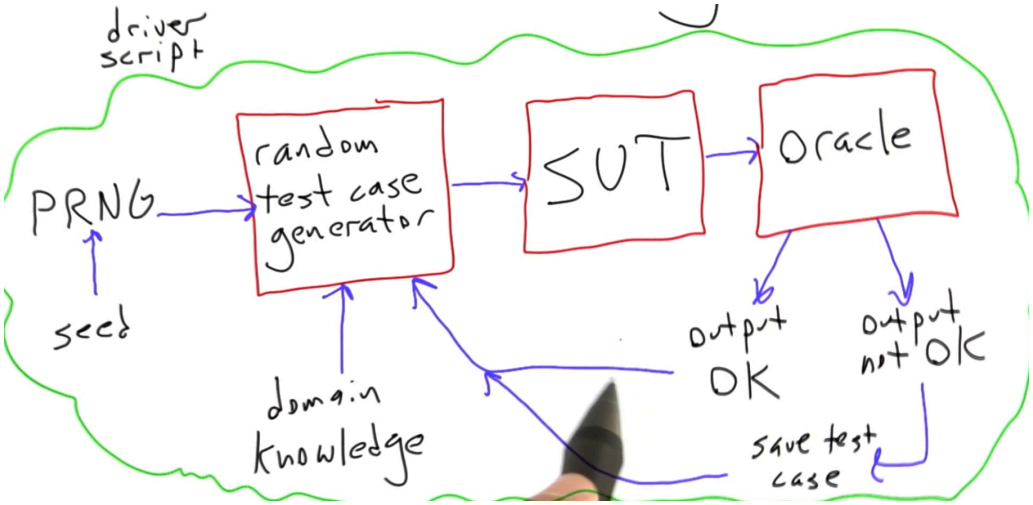
https://www.udacity.com/course/software-testing--cs258

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 it 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 and 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 give us a sufficient amount of CPU resources to 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, they 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://embed.cs.utah.edu/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

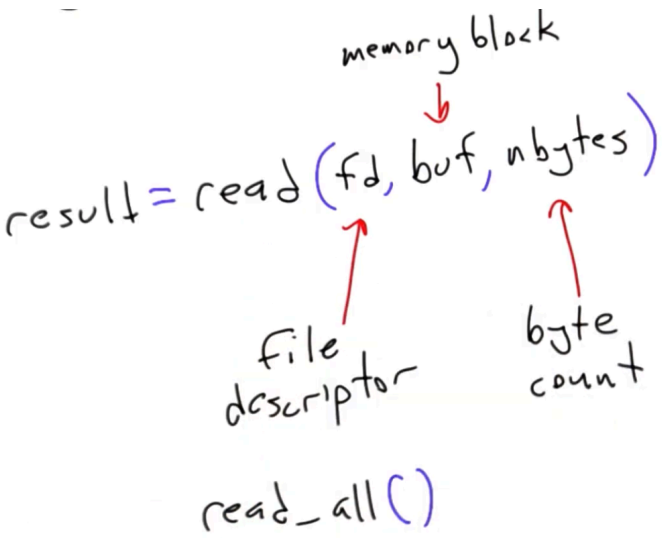
- We don't know what about that program (1600 lines or 37 kilobytes of code) was that caused Clang to crash. The stack trace is giving us a detailed version of the arguments and some other stuff. More about the analysed violated assertions and demo see the Udacity lesson 5.

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 fail, and the reason is because the **compiler developers** have conveniently **included lots of assertion violations**.
- So the driver's script are 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 the very tiny random tester.
- Remember a couple of units ago we talked about the **UNIX read system call**, so let's look at the 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 times 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.

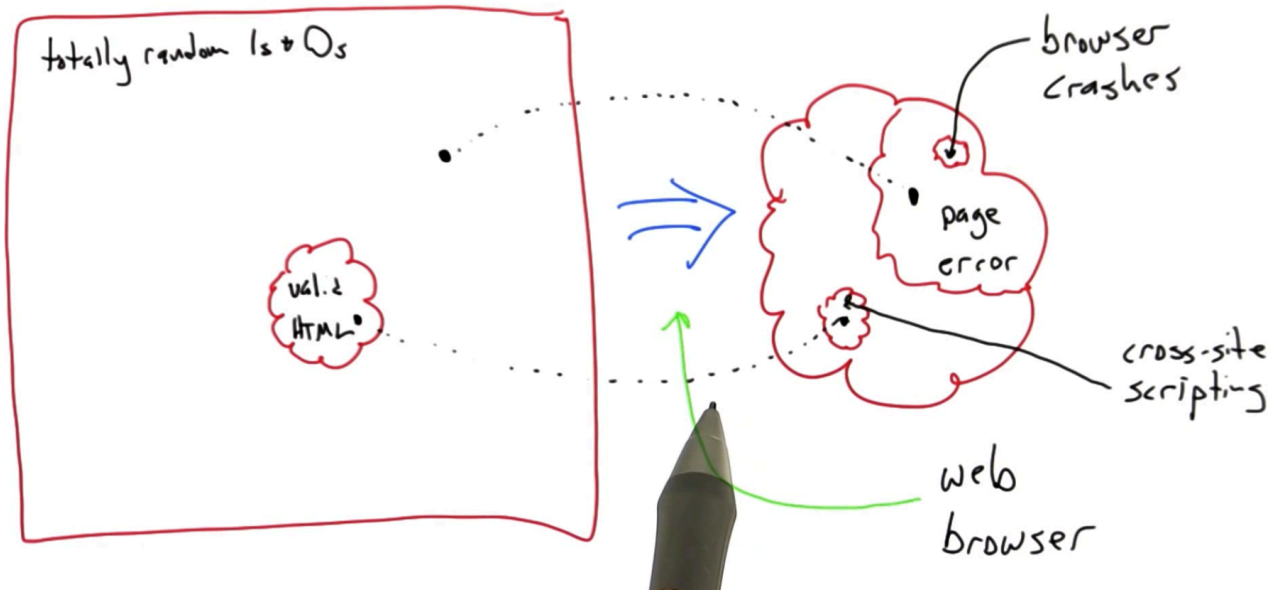
9. How it fits in the loop

- 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. d 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.

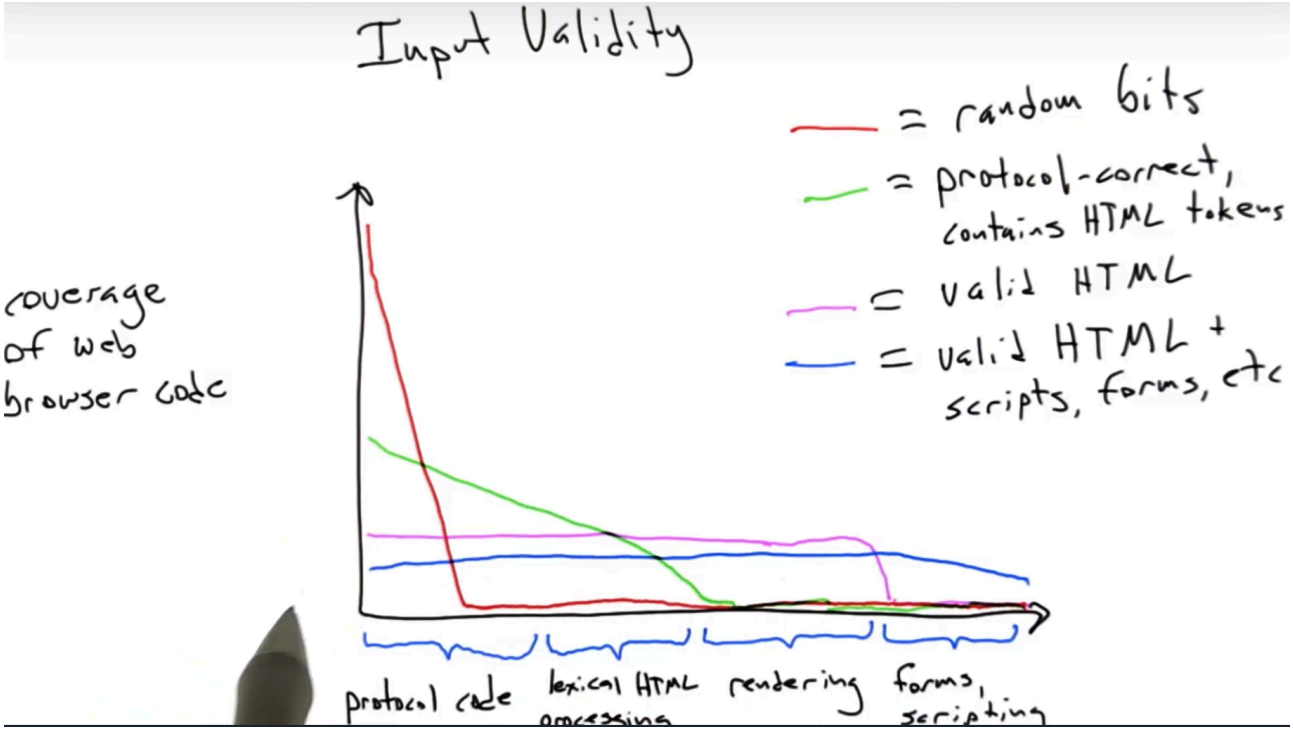
10. Input validity

- 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

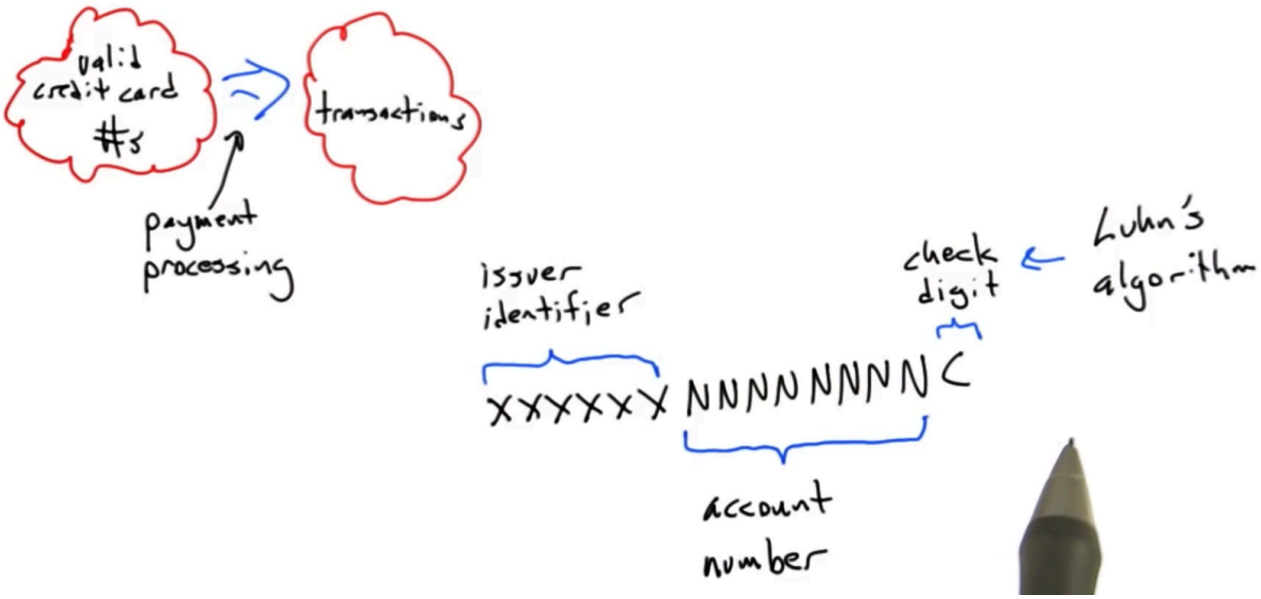
- Let's take a little bit different view of the same problem and draw a graph on the level of code coverage with random test cases induced on the SUT.



- 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

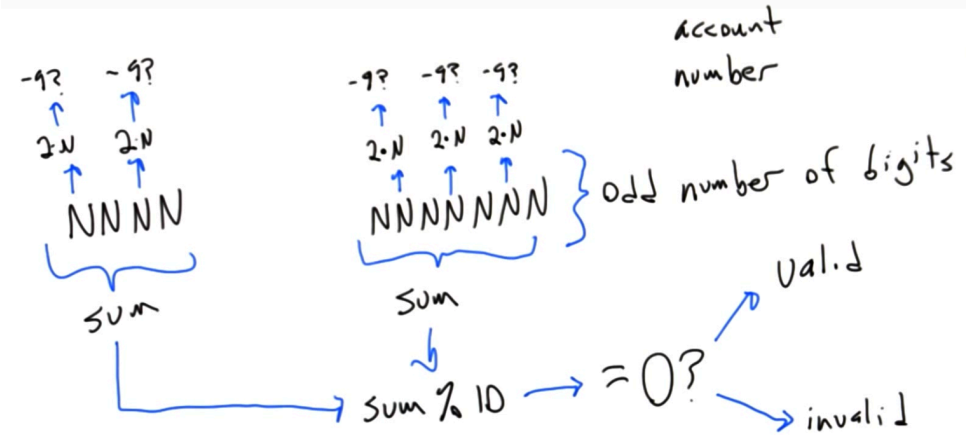
- Let’s assume that we work on some SUT whose input domain is valid credit card numbers. **SUT** is probably doing something like **processing payments** and its output is going to be something like **completed transactions**.
- We can test a variety of credit cards by running through our processing 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** so here's how those work.



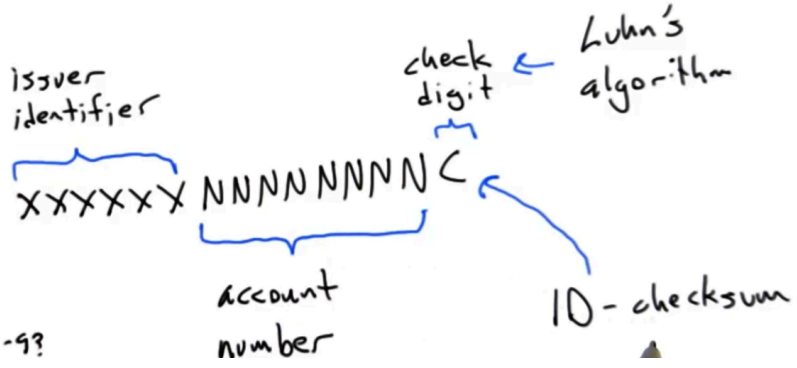
- The check digit is computed algorithmically from the preceding numbers, and **the function of the check digit is to serve as a validity check for a credit card number**.
- The check digit is computed using Luhn's algorithm.

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 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.**

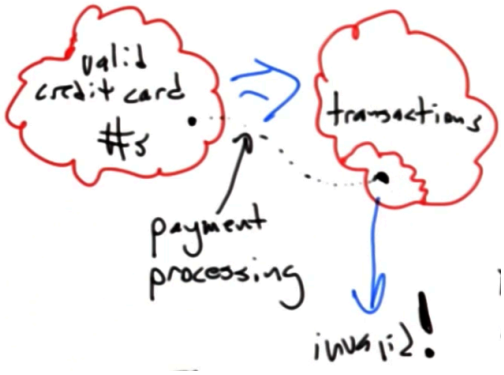
```
# concise definition of the Luhn checksum:
#
# "For a card with an even number of digits, double every odd numbered
# digit and subtract 9 if the product is greater than 9. Add up all
# the even digits as well as the doubled-odd digits, and the result
# must be a multiple of 10 or it's not a valid 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))
```

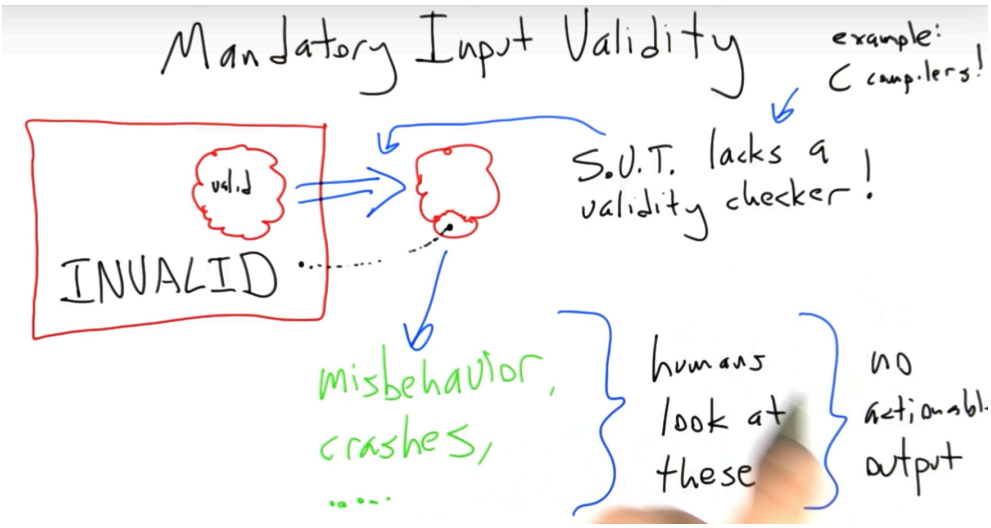
15. Problems with random tests

- Below 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.

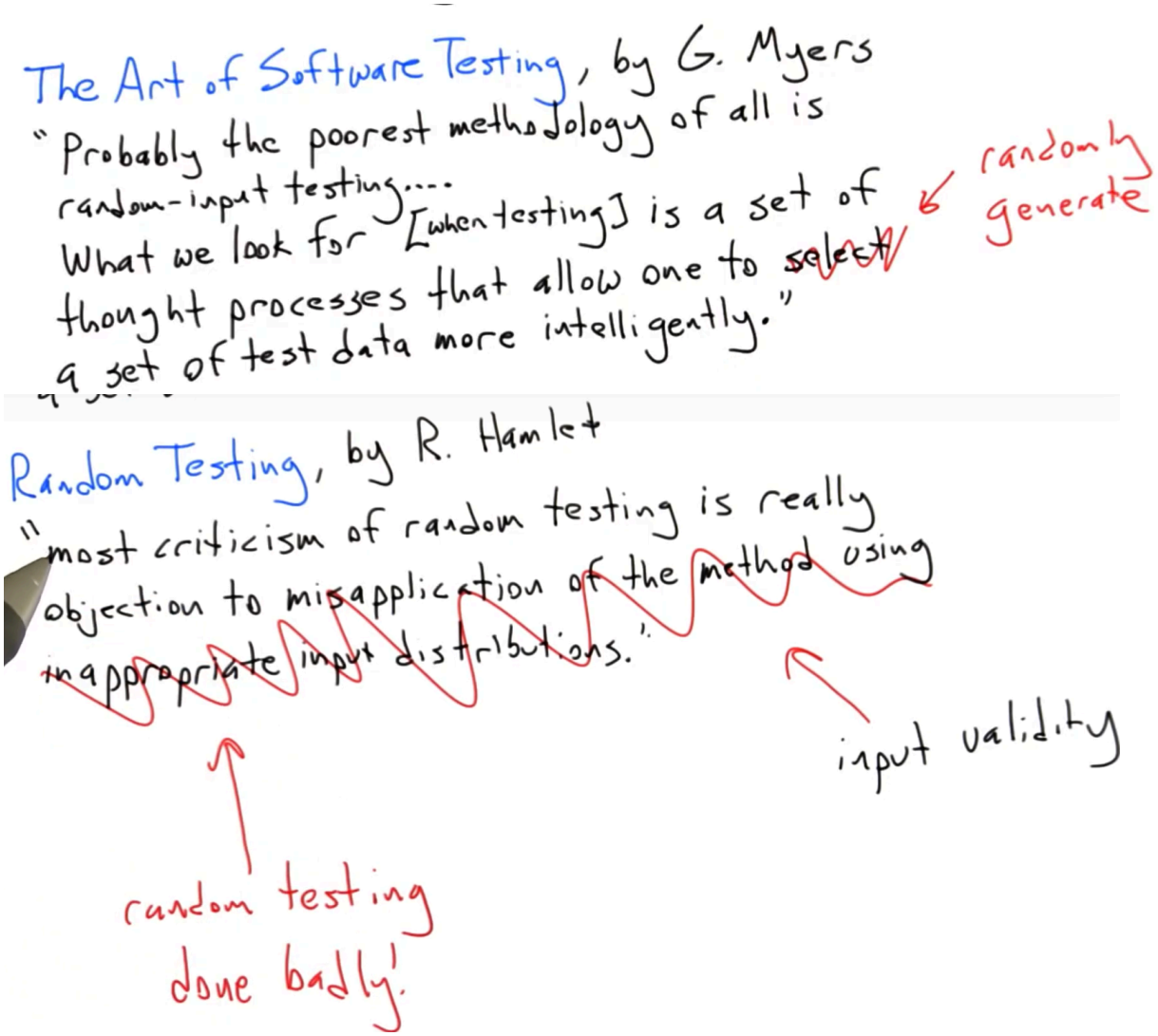


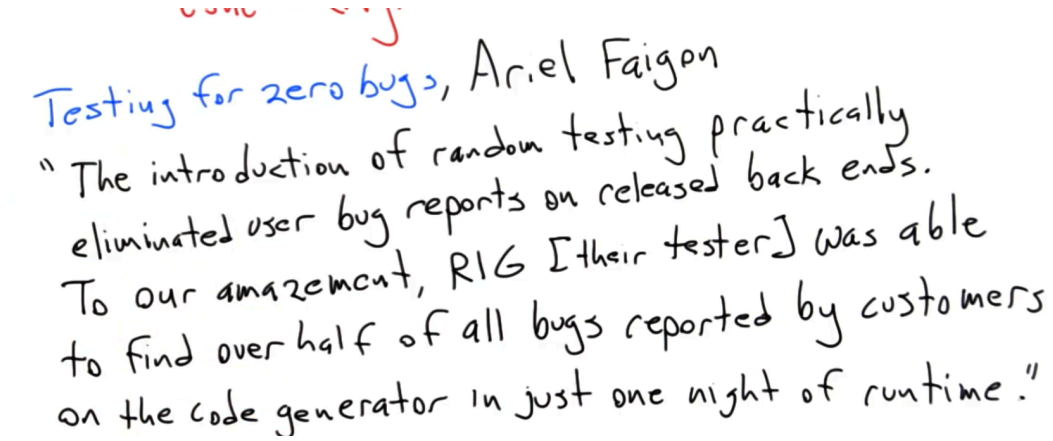
- SUT might lack a validity checker for performance reasons, for code maintainability reasons or there exists software for which it is impossible to construct a validity checker. Compilers look for syntactically invalidacy and fail to check for the dynamic validity properties that are required for certain 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, which is that random testing gets no respect, i.e. people often think that it's a really stupid way to test software.

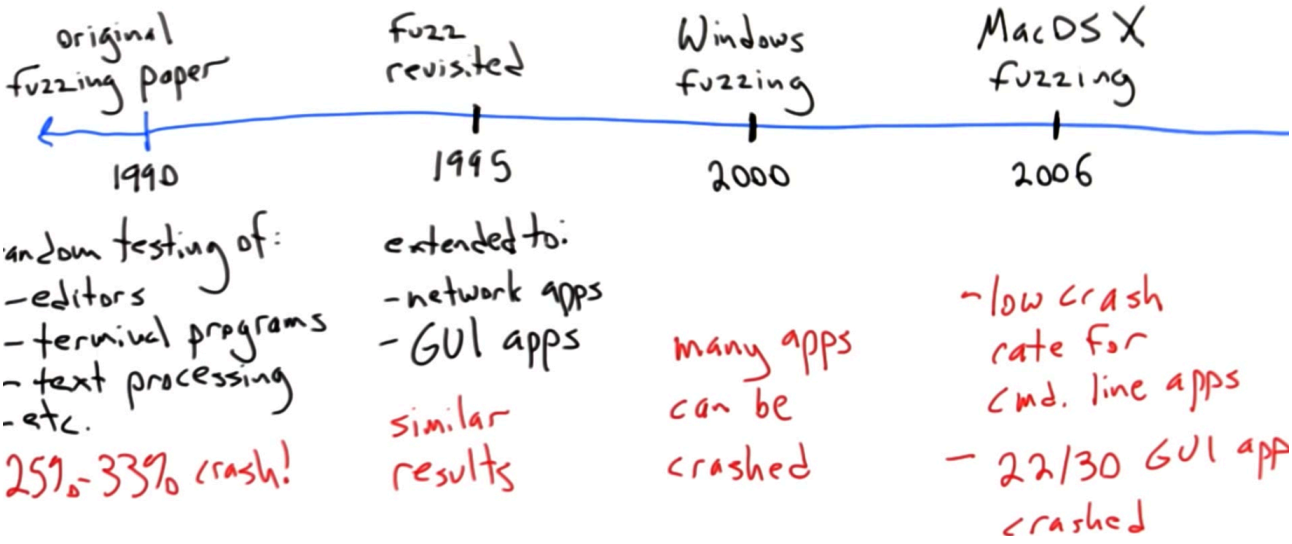
  Corrected excerpts from classic references:





18. Random testing vs fuzzing

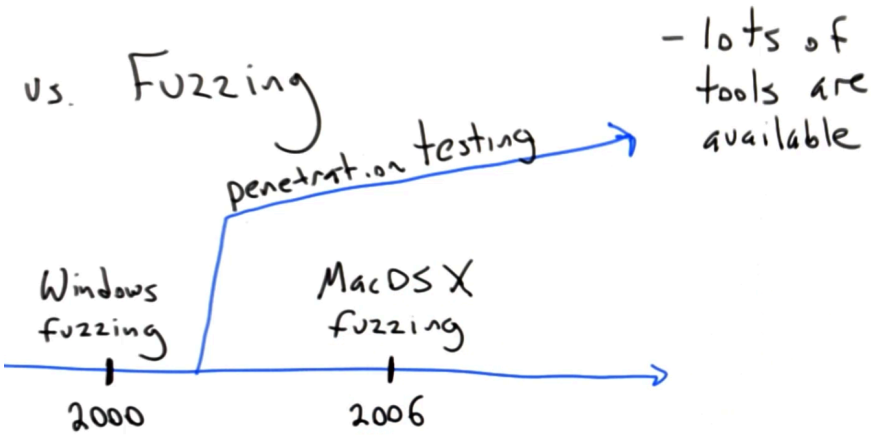
- They are the same thing. In 1990 the professor Bart Miller and his students published a paper called An Empirical Study of the Reliability of Unix Utilities.



- 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**.



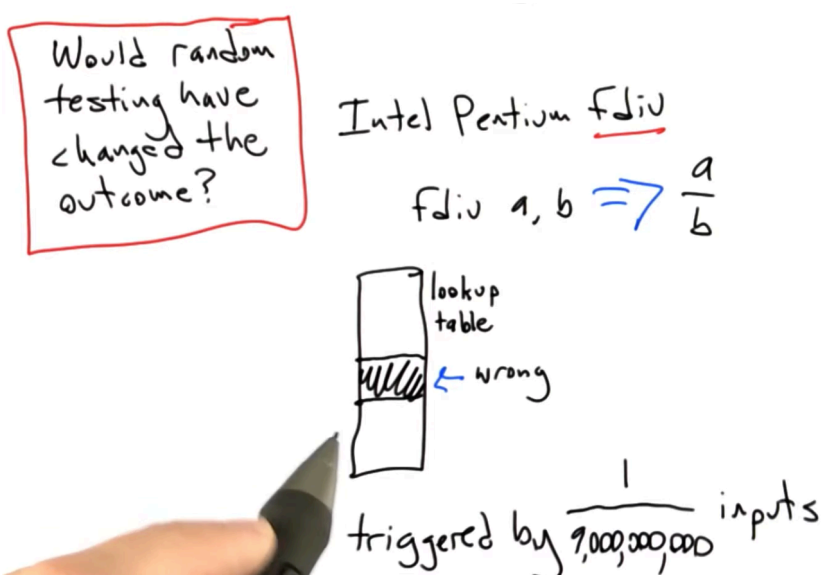
- For example we might find some sort of a machine on the Internet that offers a service and we'll do random testing over the Internet of that service with the goal of finding bugs in the service that will let us mount an attack such as a denial of service or an exploitable vulnerability for mounting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

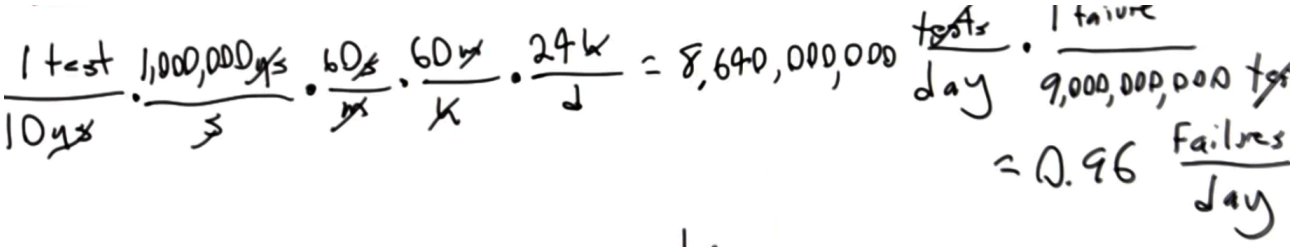
- A genre of fiction where we explore what would have happened to the world had some historical event turned out differently. For example, we might have an alternate history now exploring what would have happened if the Allies hadn't won World War II.
- What we're going to do is look at a few alternate 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 had not been loading 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.

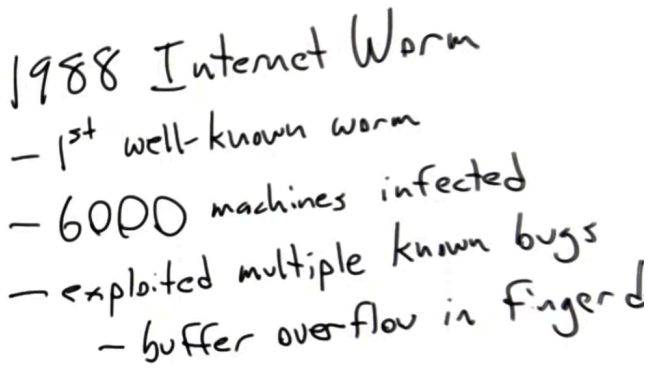


- If we perform completely random testing of the input space for fdiv, we should be able to find this bug in about a day.
- Now this kind of testing is going to need some sort of an oracle to tell if our particular output from fdiv is correct or not, i.e. IEEE floating point.

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

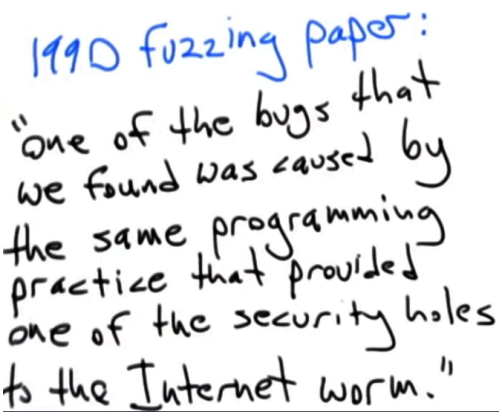
23. 1988 Internet worm

- 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.

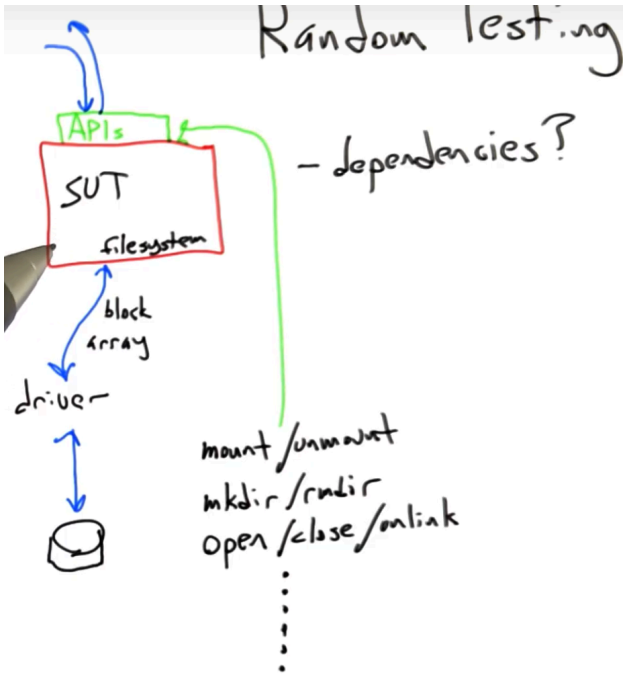
•Finger daemon was a service that would run on UNIX machines of the time, and let you query a remote machine to learn about whether a user was logged into that machine and some other stuff.



- **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.

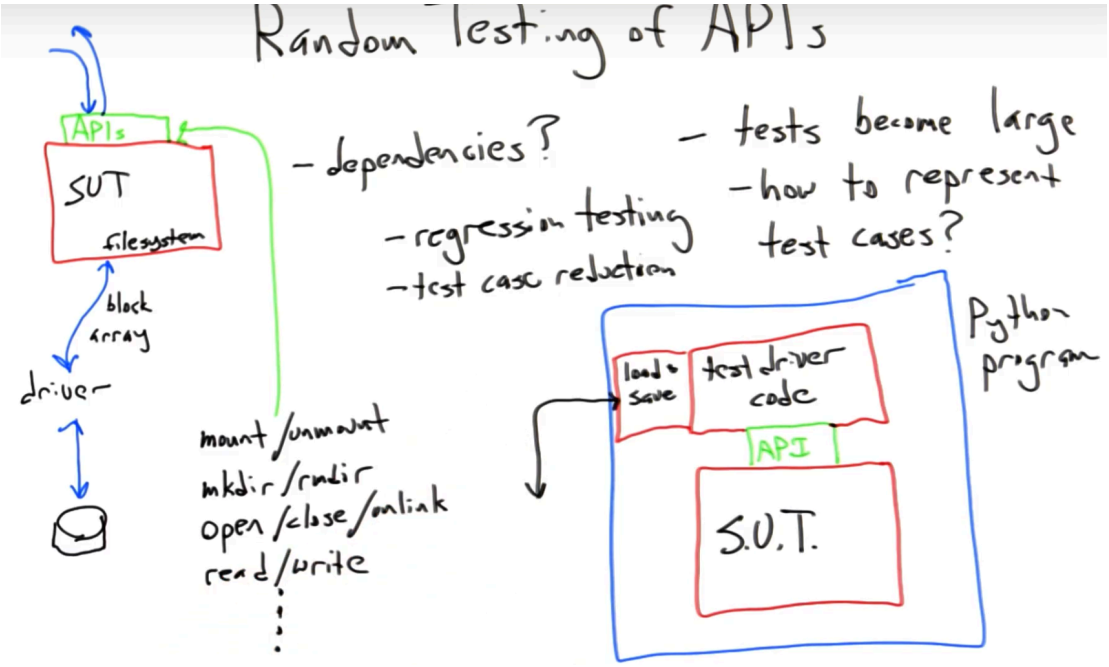
•Let's take the example of randomly testing a file system. It has to manage all of the structure just to manage the free space to efficiently respond to all sorts of

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**.

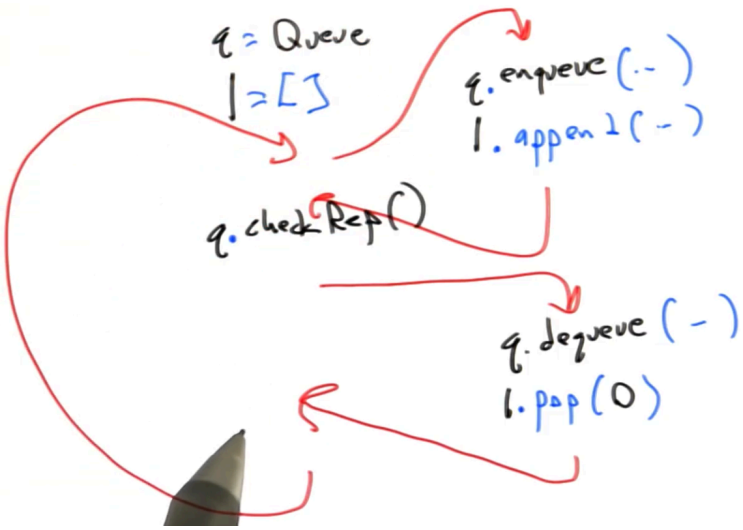
- It's often the case that when random tests become extremely large, we need to turn them into first class objects, i.e. objects living on disk using a save and load routine in order to perform **test case reduction,** a piece of technology that's very often combined with random testing, that takes a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

TO BE CONTINUED...

Random testing

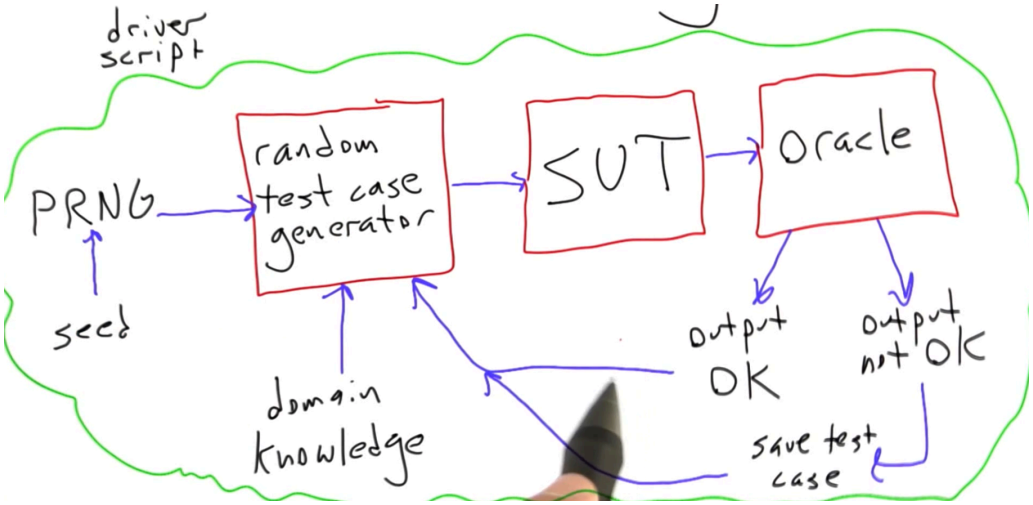
https://www.udacity.com/course/software-testing--cs258

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 it 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 and 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 give us a sufficient amount of CPU resources to 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, they 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://embed.cs.utah.edu/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

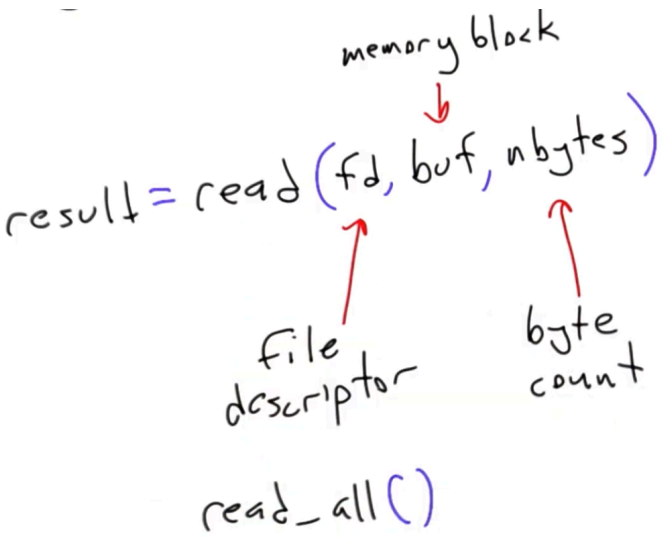
- We don't know what about that program (1600 lines or 37 kilobytes of code) was that caused Clang to crash. The stack trace is giving us a detailed version of the arguments and some other stuff. More about the analysed violated assertions and demo see the Udacity lesson 5.

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 fail, and the reason is because the **compiler developers** have conveniently **included lots of assertion violations**.
- So the driver's script are 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 the very tiny random tester.
- Remember a couple of units ago we talked about the **UNIX read system call**, so let's look at the 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 times 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.

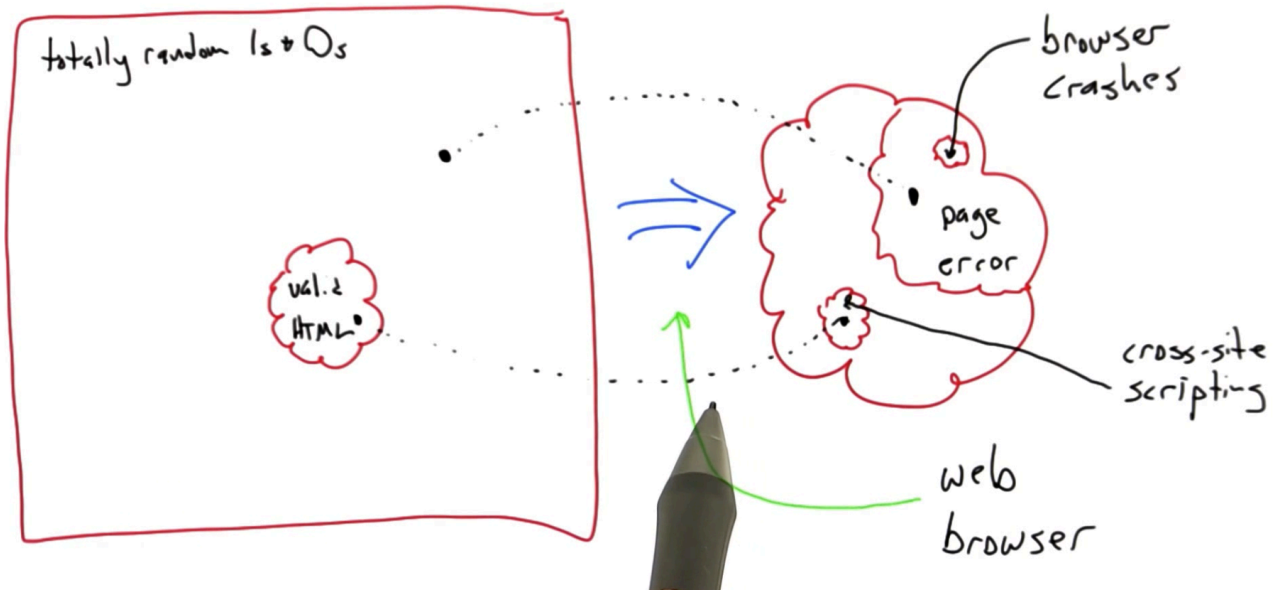
9. How it fits in the loop

- 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. d 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.

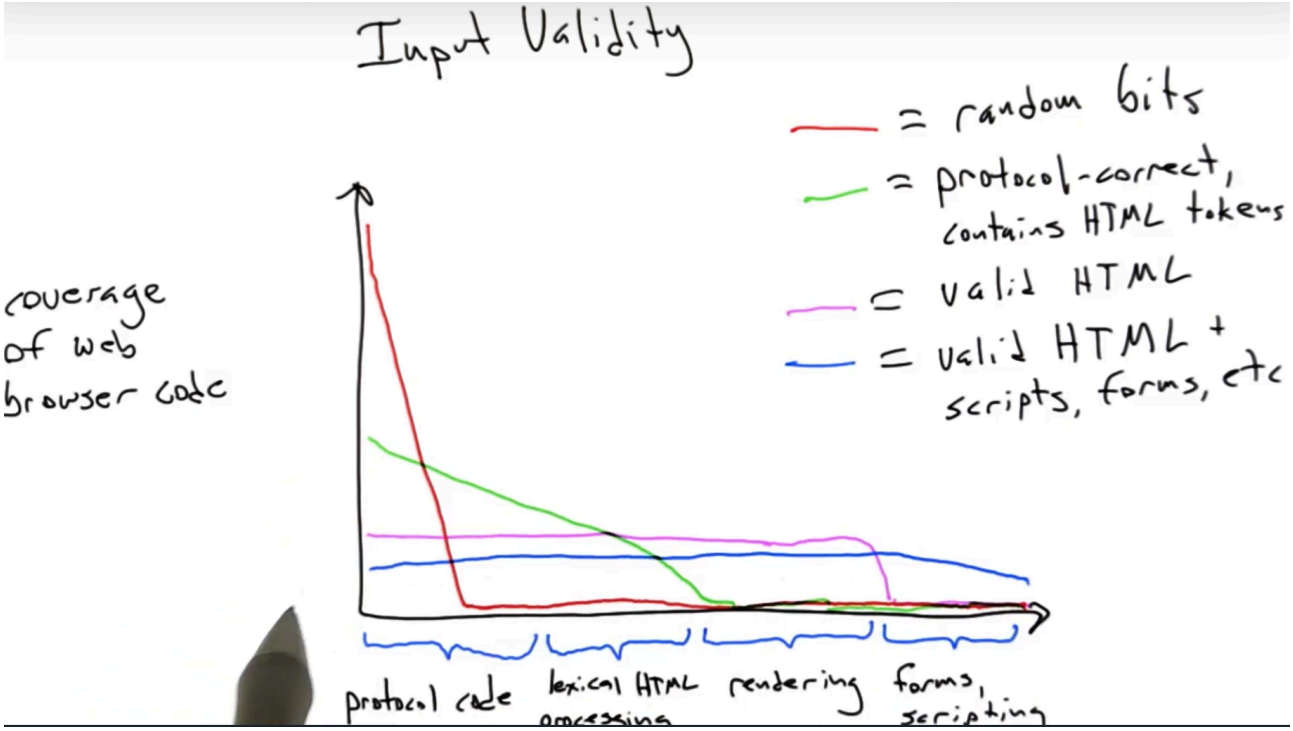
10. Input validity

- 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

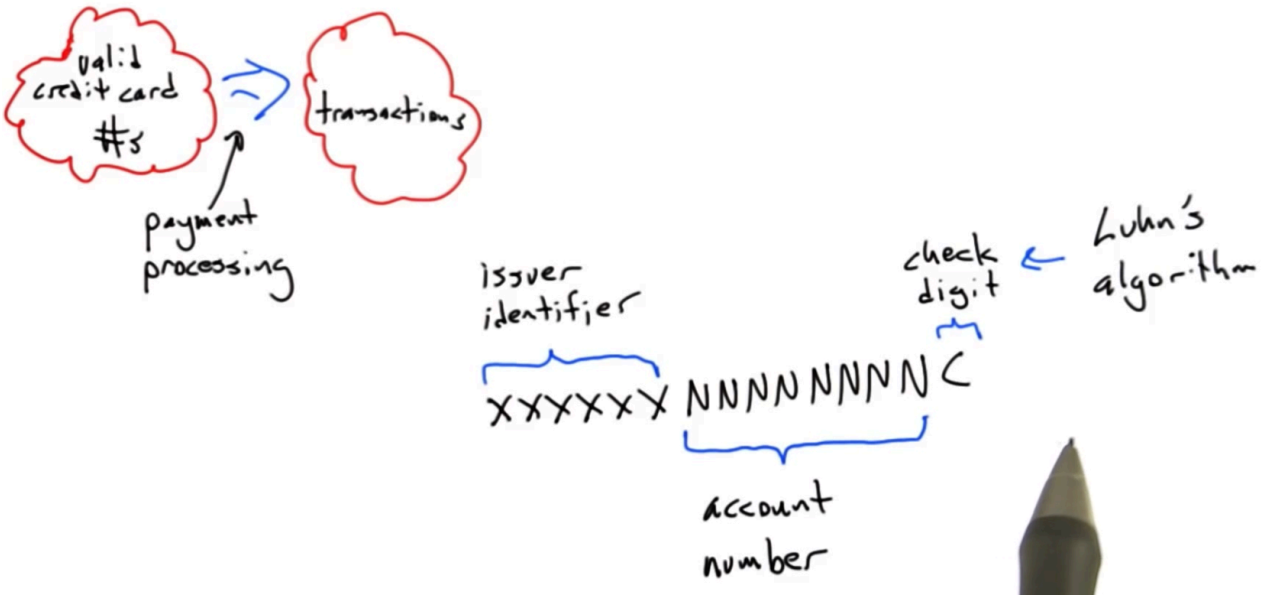
- Let's take a little bit different view of the same problem and draw a graph on the level of code coverage with random test cases induced on the SUT.



- 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

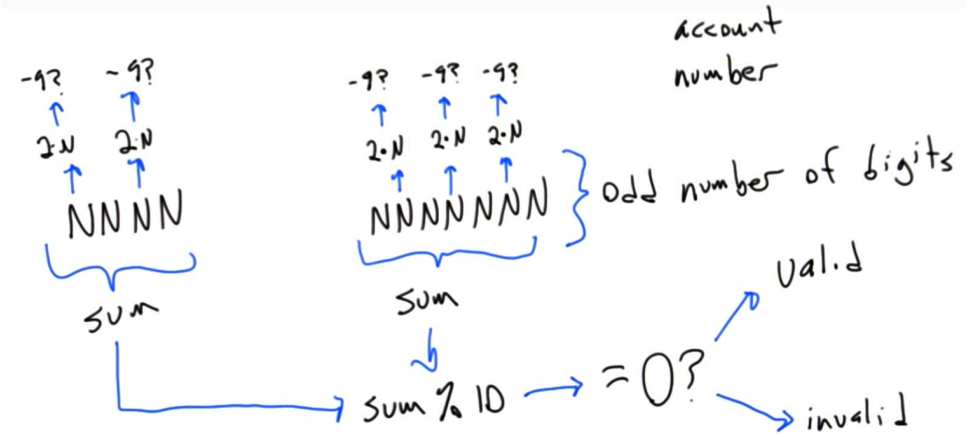
- Let’s assume that we work on some SUT whose input domain is valid credit card numbers. **SUT** is probably doing something like **processing payments** and its output is going to be something like **completed transactions**.
- We can test a variety of credit cards by running through our processing 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** so here's how those work.



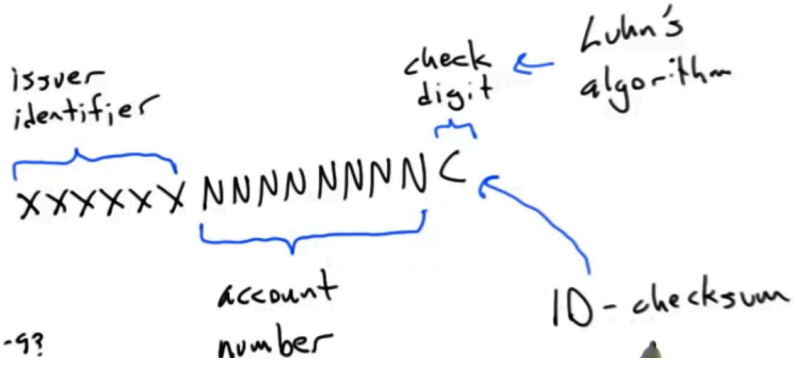
- The check digit is computed algorithmically from the preceding numbers, and **the function of the check digit is to serve as a validity check for a credit card number**.
- The check digit is computed using Luhn's algorithm.

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 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.**

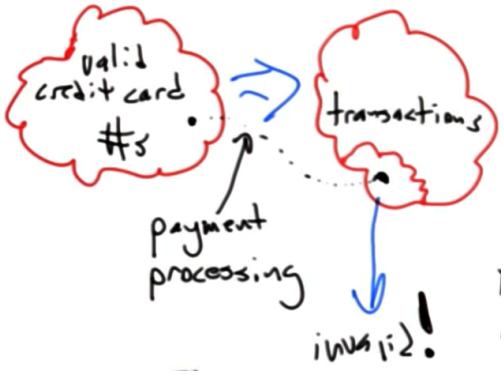
```
# concise definition of the Luhn checksum:
#
# "For a card with an even number of digits, double every odd numbered
# digit and subtract 9 if the product is greater than 9. Add up all
# the even digits as well as the doubled-odd digits, and the result
# must be a multiple of 10 or it's not a valid 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))
```

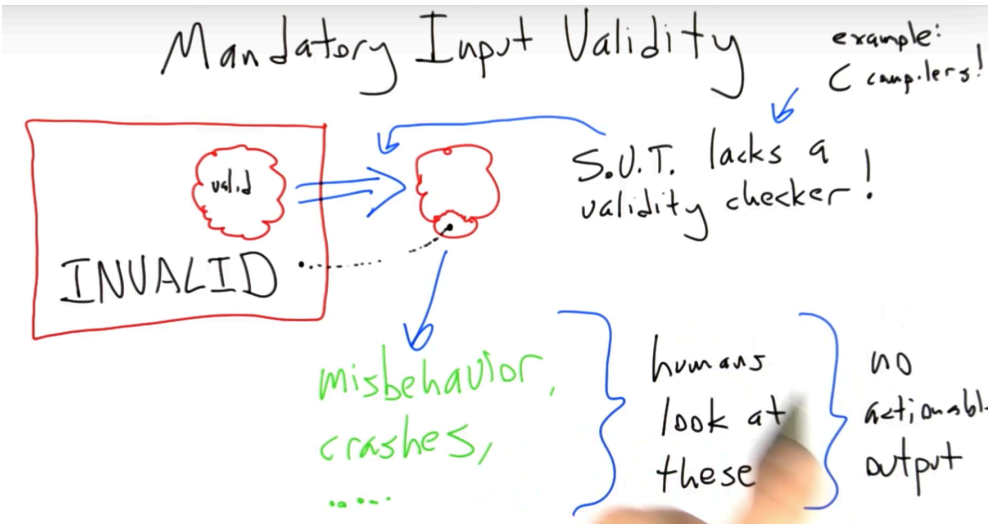
15. Problems with random tests

- Below 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.

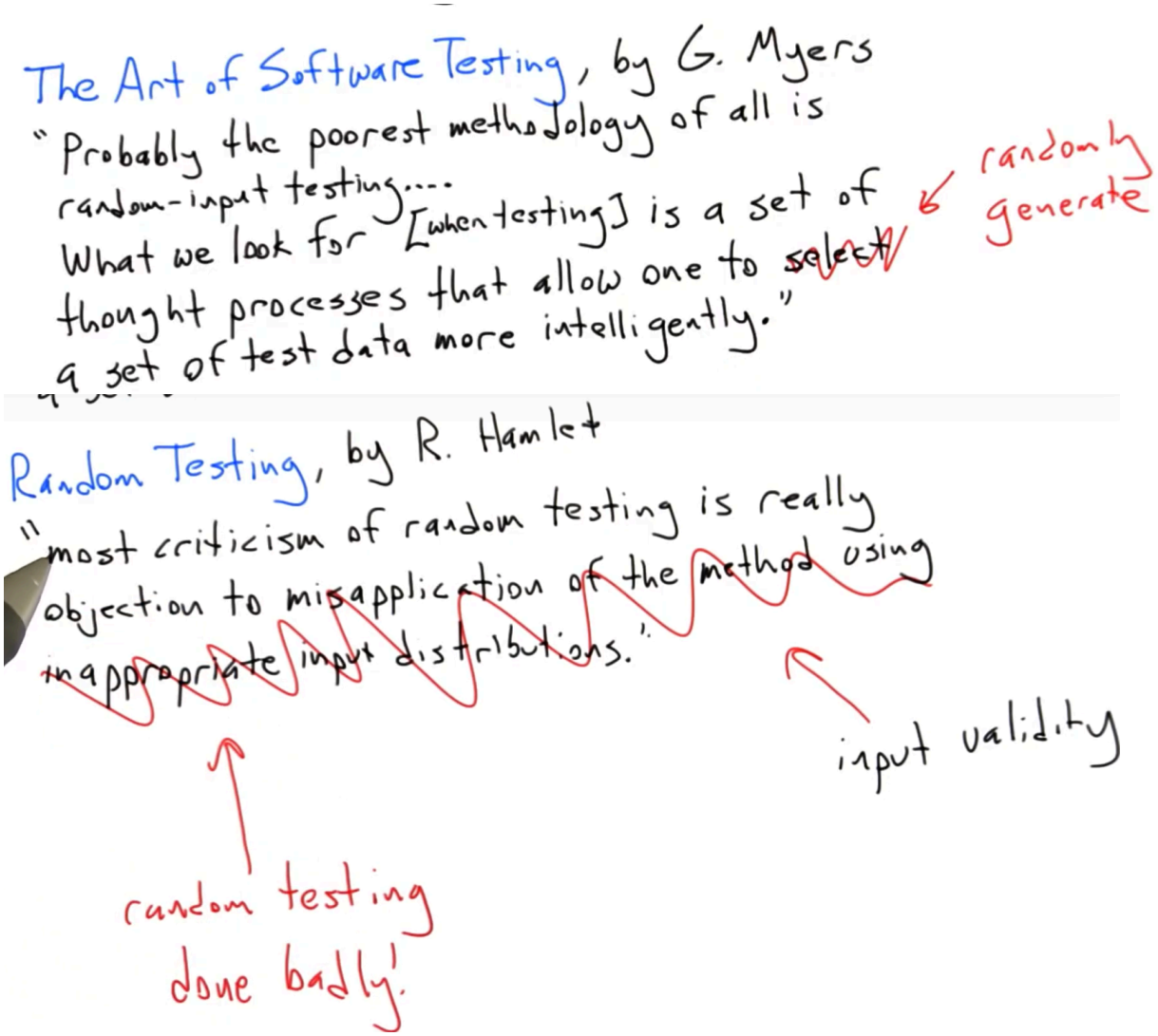


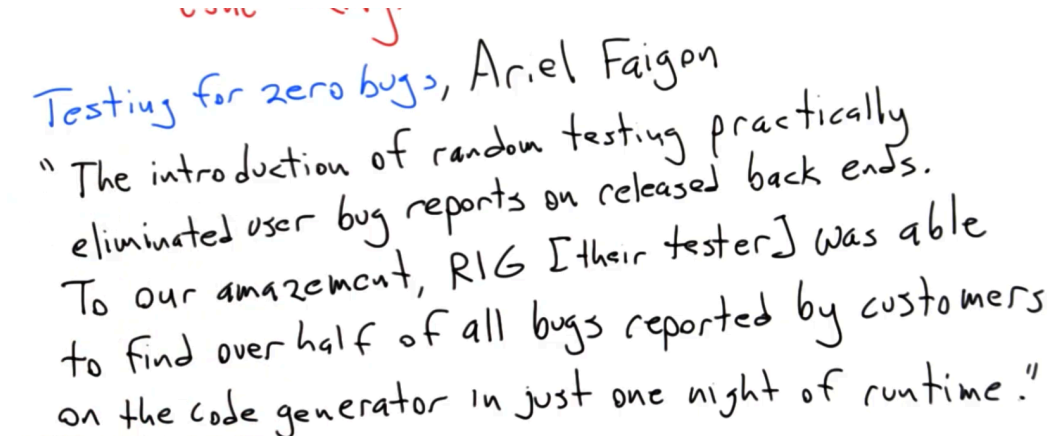
- SUT might lack a validity checker for performance reasons, for code maintainability reasons or there exists software for which it is impossible to construct a validity checker. Compilers look for syntactically invalidacy and fail to check for the dynamic validity properties that are required for certain 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, which is that random testing gets no respect, i.e. people often think that it's a really stupid way to test software.

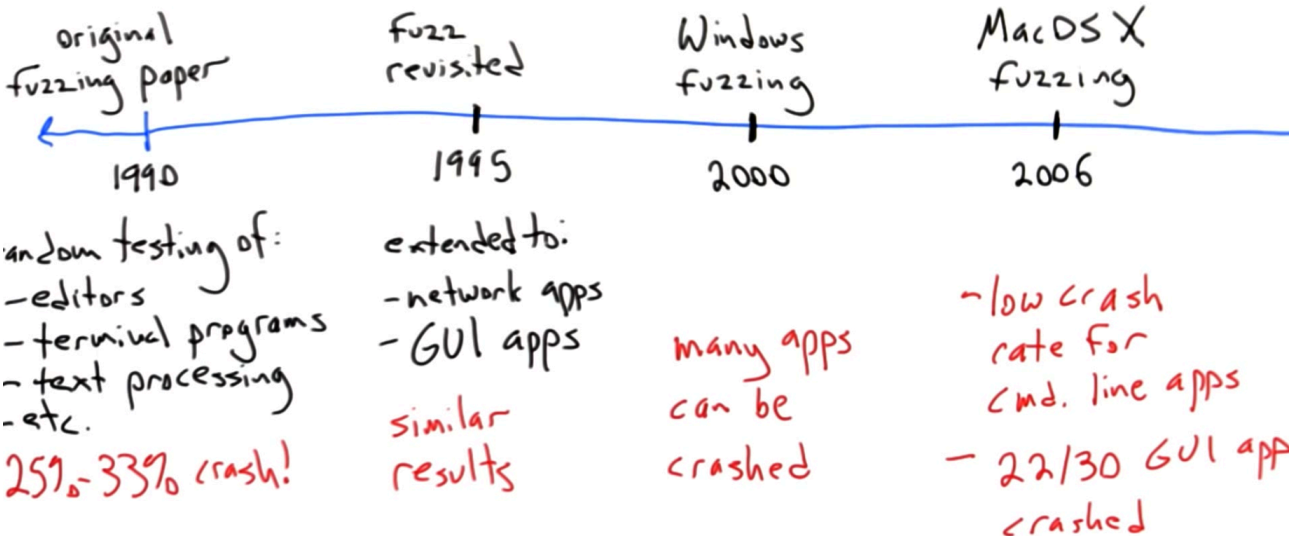
  Corrected excerpts from classic references:





18. Random testing vs fuzzing

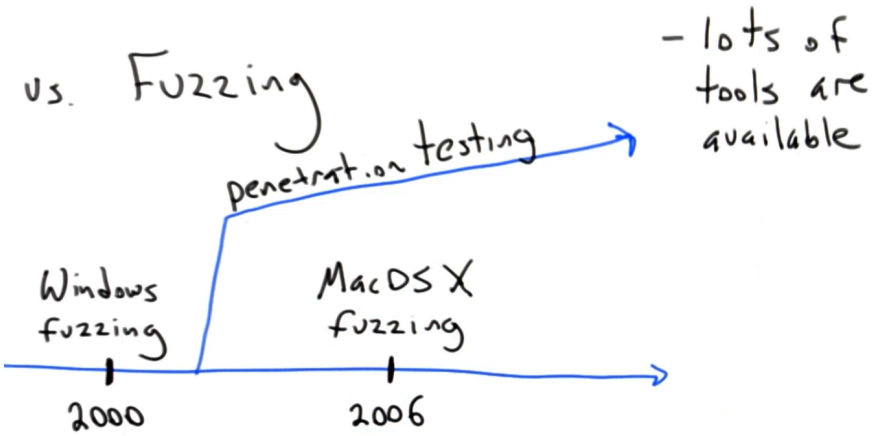
- They are the same thing. In 1990 the professor Bart Miller and his students published a paper called An Empirical Study of the Reliability of Unix Utilities.



- 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**.



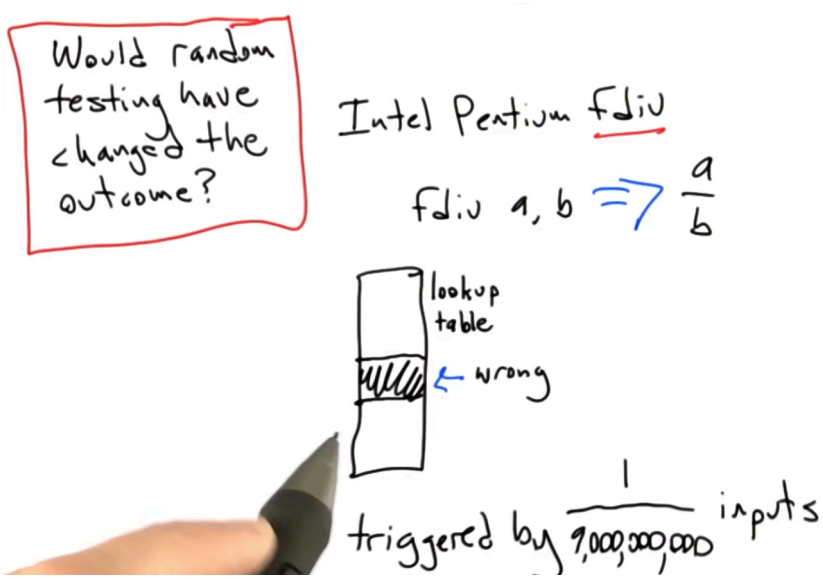
- For example we might find some sort of a machine on the Internet that offers a service and we'll do random testing over the Internet of that service with the goal of finding bugs in the service that will let us mount an attack such as a denial of service or an exploitable vulnerability for mounting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

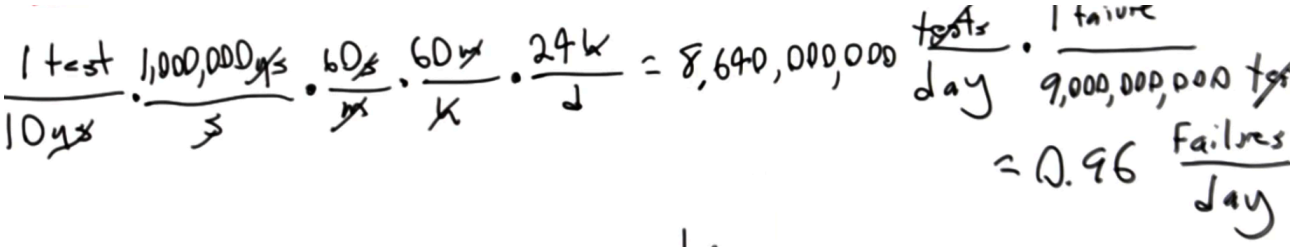
- A genre of fiction where we explore what would have happened to the world had some historical event turned out differently. For example, we might have an alternate history now exploring what would have happened if the Allies hadn't won World War II.
- What we're going to do is look at a few alternate 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 had not been loading 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.

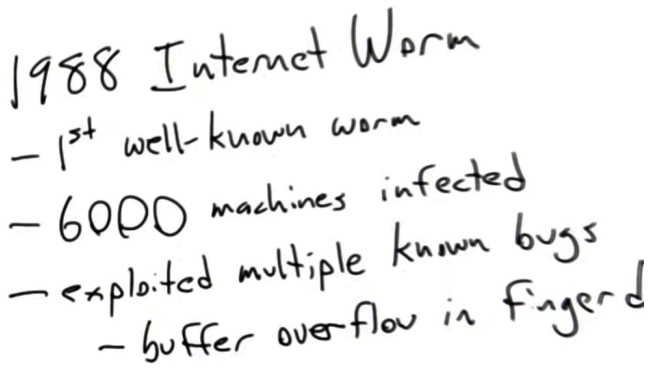


- If we perform completely random testing of the input space for fdiv, we should be able to find this bug in about a day.
- Now this kind of testing is going to need some sort of an oracle to tell if our particular output from fdiv is correct or not, i.e. IEEE floating point.

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

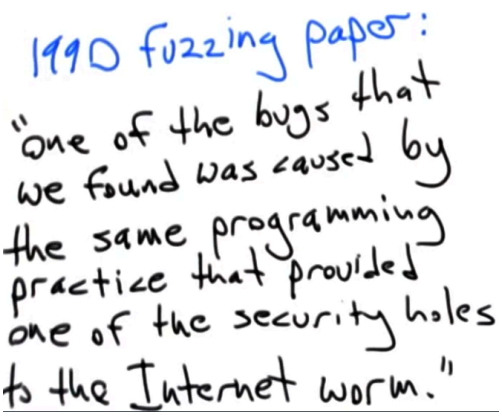
23. 1988 Internet worm

- 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.

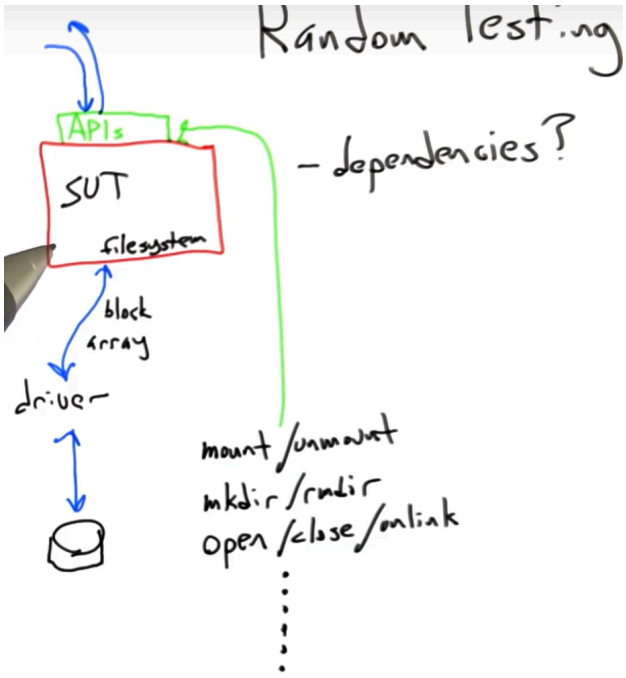
•Finger daemon was a service that would run on UNIX machines of the time, and let you query a remote machine to learn about whether a user was logged into that machine and some other stuff.



- **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.

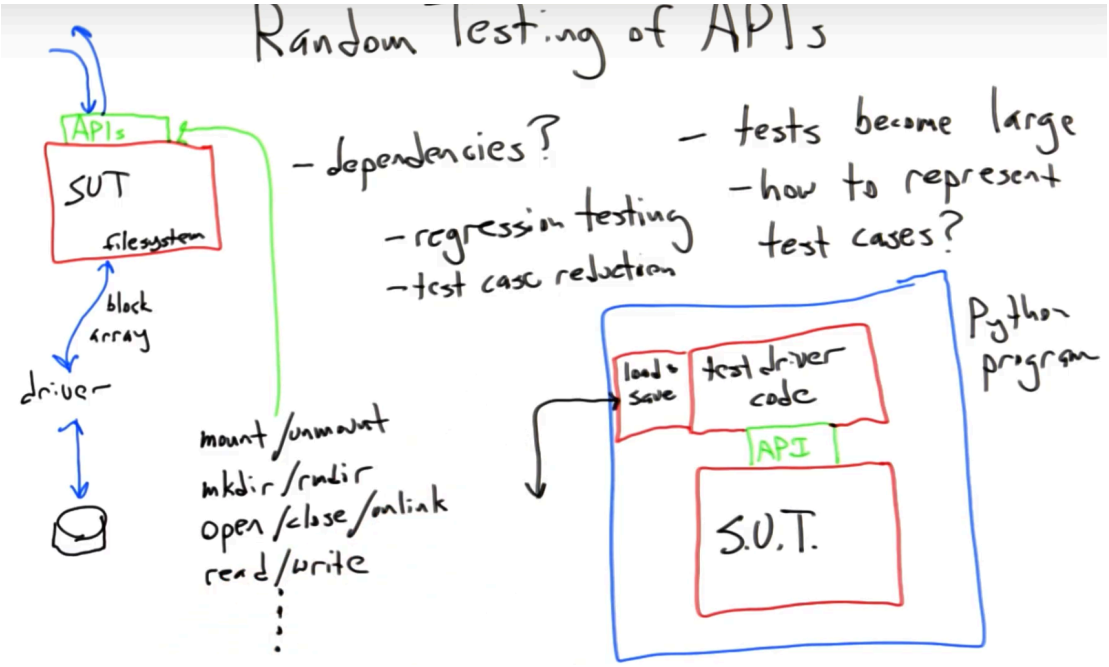
•Let's take the example of randomly testing a file system. It has to manage all of the structure just to manage the free space to efficiently respond to all sorts of

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**.

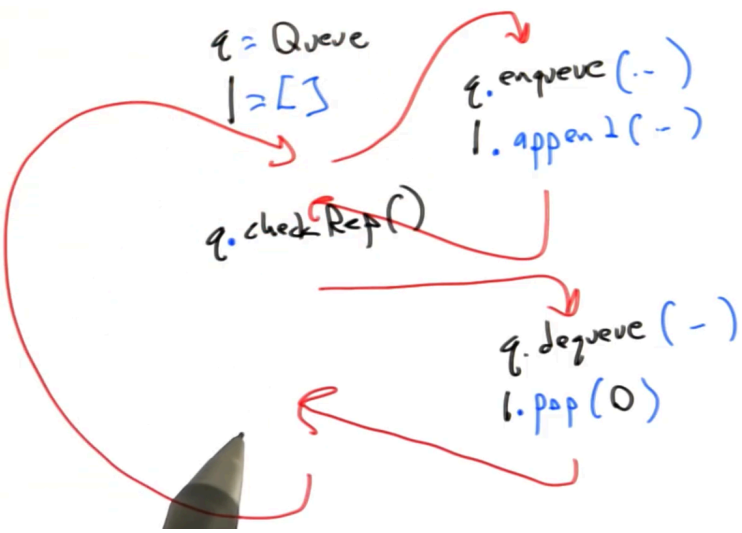
- It's often the case that when random tests become extremely large, we need to turn them into first class objects, i.e. objects living on disk using a save and load routine in order to perform **test case reduction,** a piece of technology that's very often combined with random testing, that takes a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

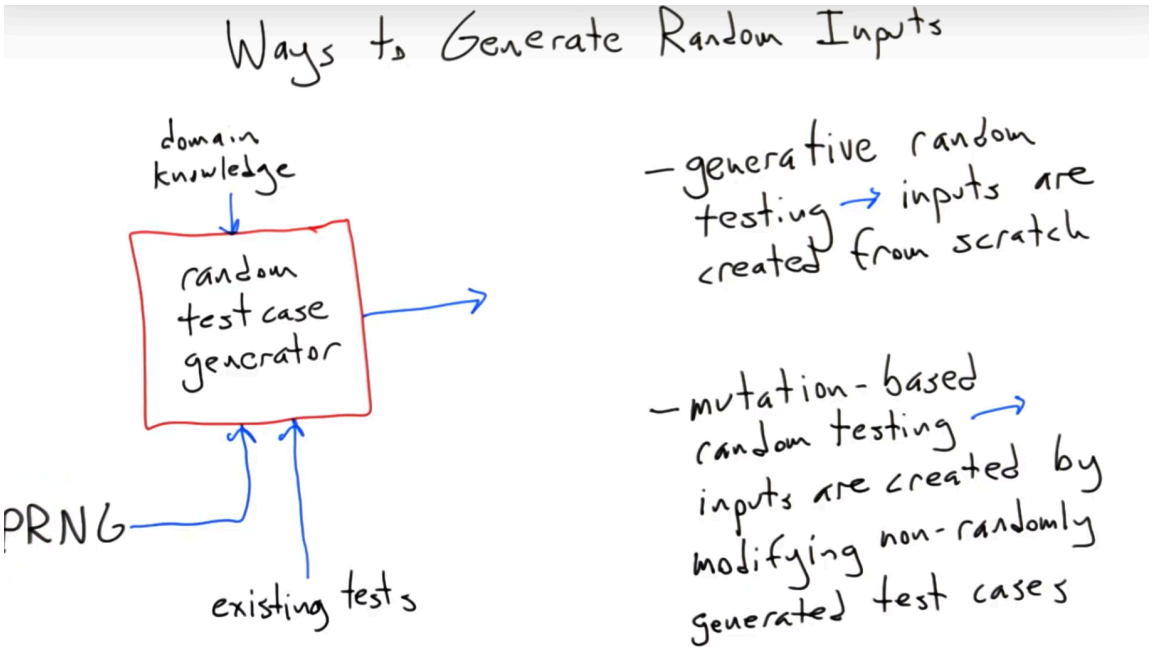



```
# TASK:
# Write a random tester for the Queue class.
# The random tester should repeatedly call the Queue methods
# on random input in a semi-random fashion.
# For instance, if you wanted to randomly decide between
# calling enqueue and dequeue, you would write something like this:
#
# q = Queue(500)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

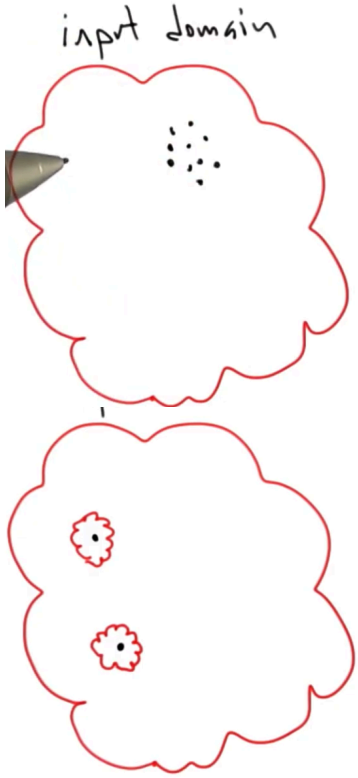
27. Generating random inputs

- The test case generator has 2 inputs: pseudo-random numbers from a pseudo-random number generator, and it's also predicated 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 input 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 is 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 file into buffer

# 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

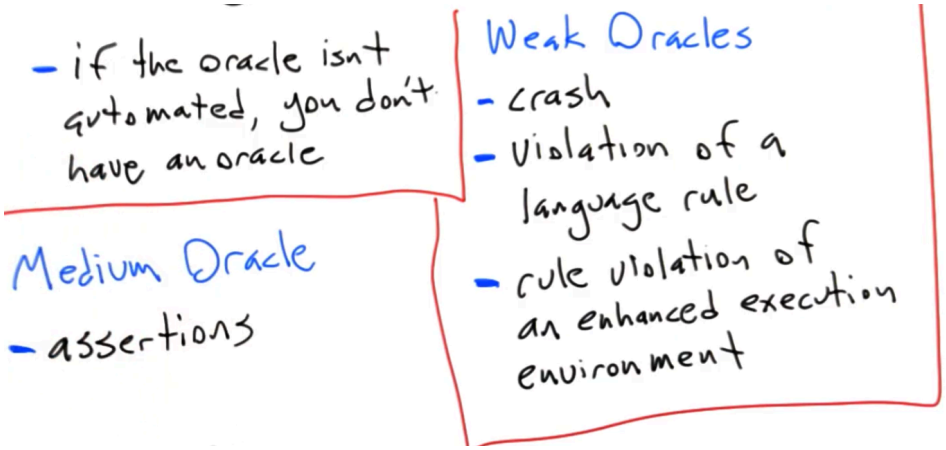
Hamlet '94:
"random testing has only a specialized niche in practice because an effective oracle is seldom available."

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.

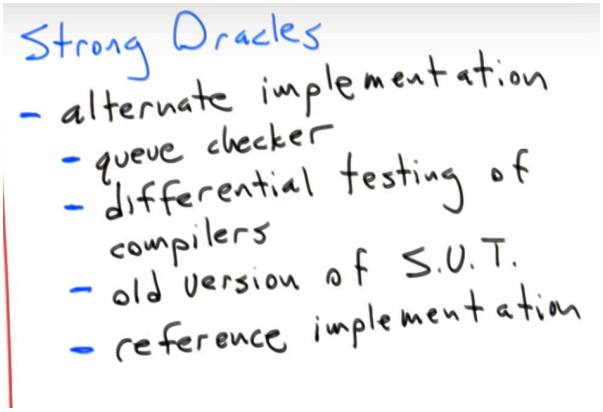
31. Medium oracles

- 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.

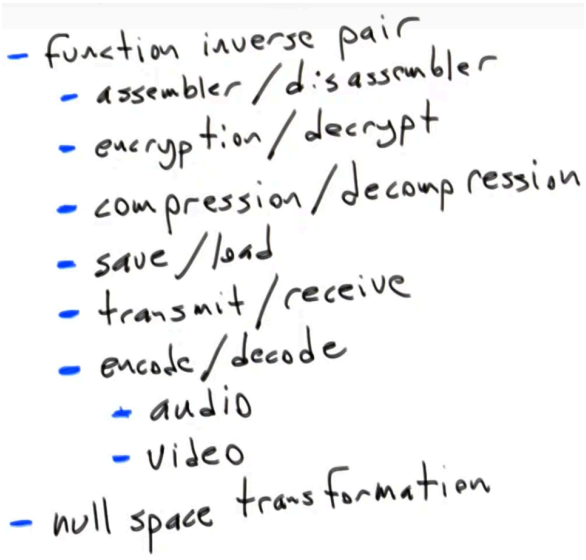
32. Strong oracles



- 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.

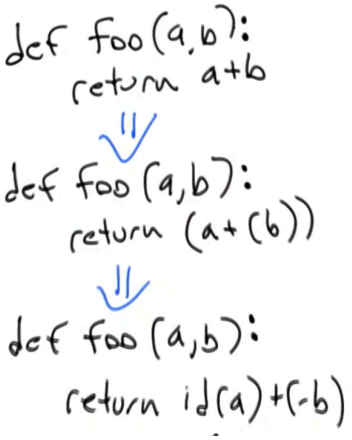
33. Function inverse pairs



•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

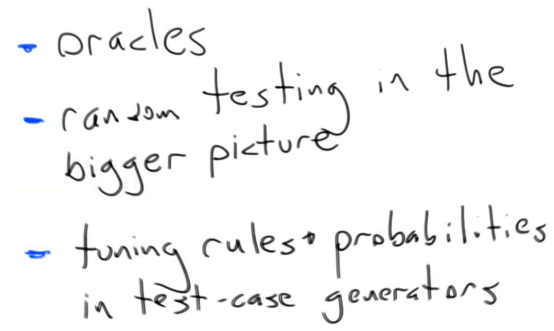


- We take a random test case or any test case and we make some change to it that shouldn't affect how it's treated by the SUT.
- Let's consider a simple Python function, namely foo. One possible null space transformation would be to add a level of parentheses or maybe several.
- We can do something very 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.

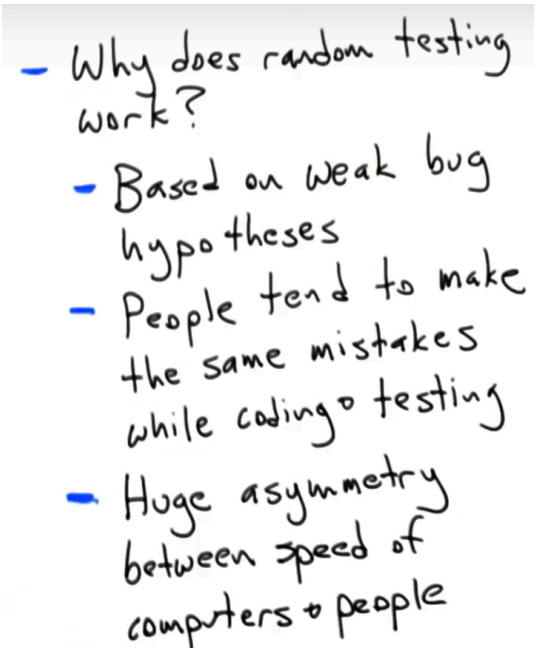
Random testing in practice

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

1. Introduction



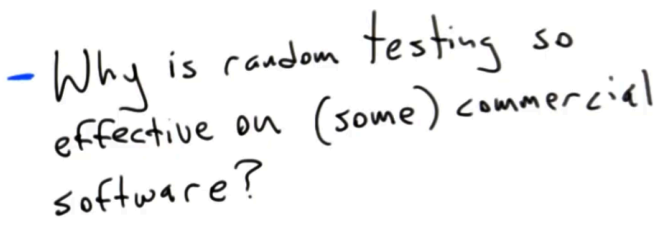
2. Random testing in the bigger picture



•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.

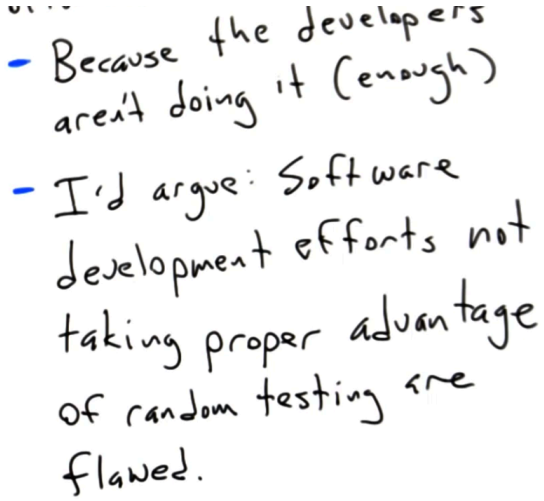
•If we forgot to implement some feature or if we miss-implemented on feature, we won't test the things done wrong. Random testing to some extent get us out of this problem because it can construct test cases to test things that we don't actually understand or know very well.

- 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.



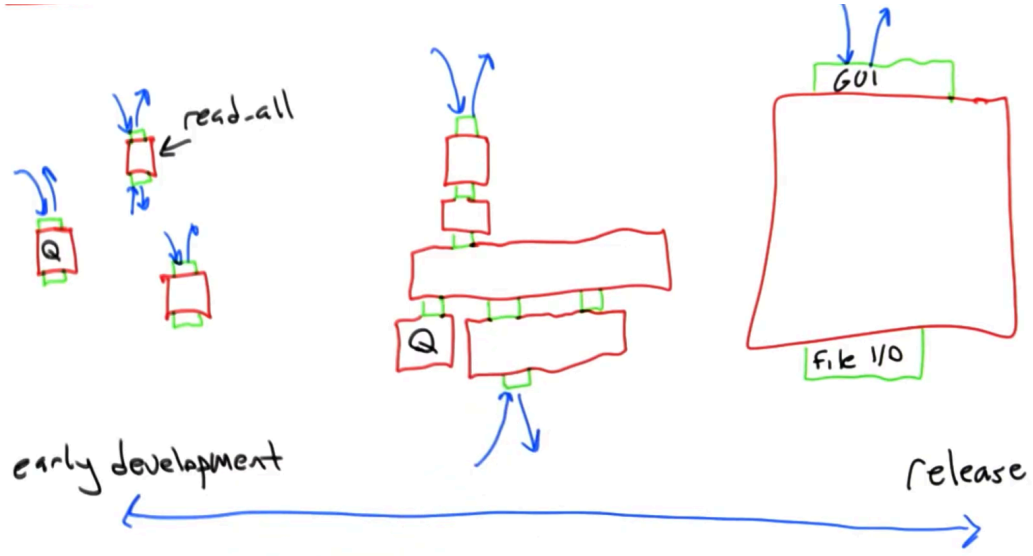
•There's pretty ample evidence, for example the fuzzing papers that we discussed or the Charlie Miller's talk that you can watch online, Babysitting an army of monkeys.

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:

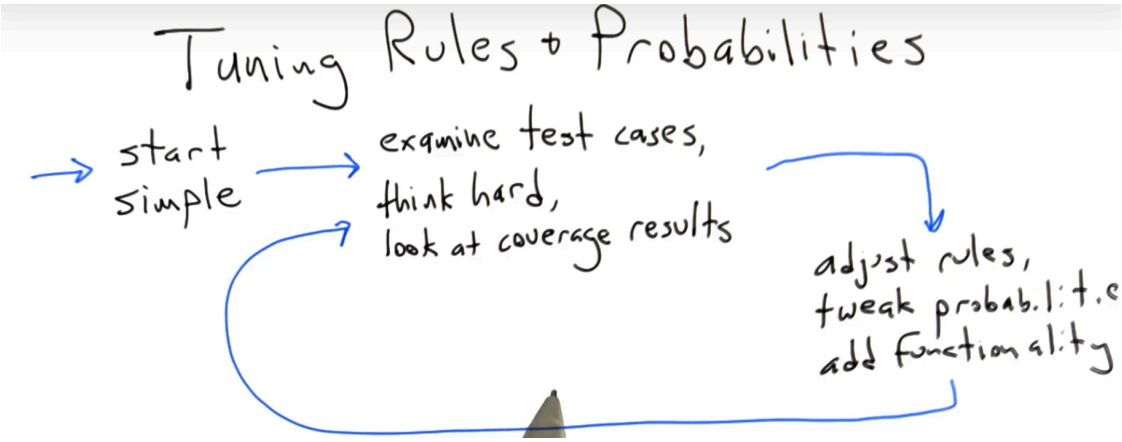


- We want to ensure as we're developing the modules that we're creating robust pieces of software whose interfaces we understand and that will be solid foundation for future work.
- It's going to be the case that some of our random testers become useless (e.g. the bounded queue) but others (e.g. those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

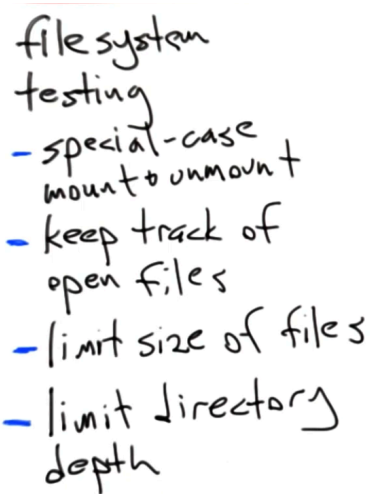
4. Random system testers

- 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



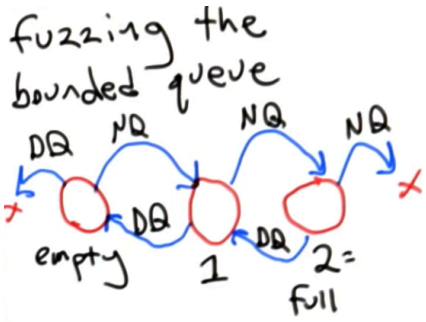
•If we start with a simple file system tester, we'll make a list of all the API calls that we'd like to test then call them randomly with random arguments.

•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.

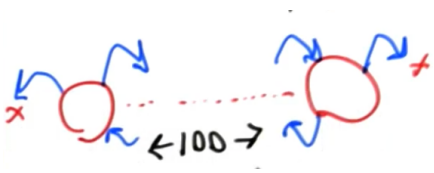
- Let's look at the queue as a FSM (FSM stands for finite state machine). It contains 3 nodes.



- 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

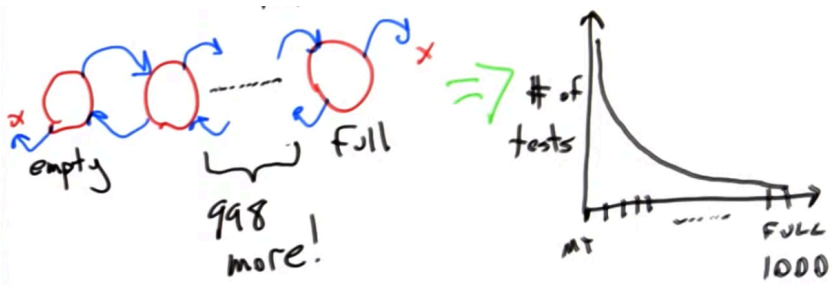
- **We go back to the random tester presented in the Random testing lecture at section 26.** We modify the code to make a 2 element queue and then run the random tester. We get a crude coverage metric. Similar information we get using a branch coverage tool.
- Let's say that we had a queue containing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

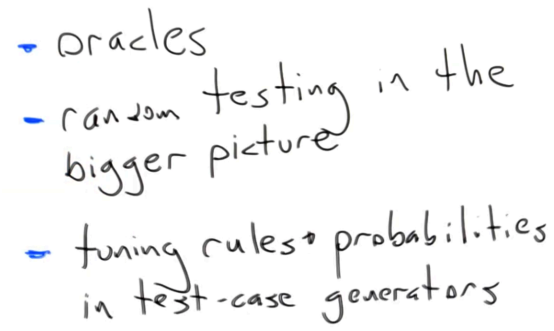
10. Stressing the whole system

-

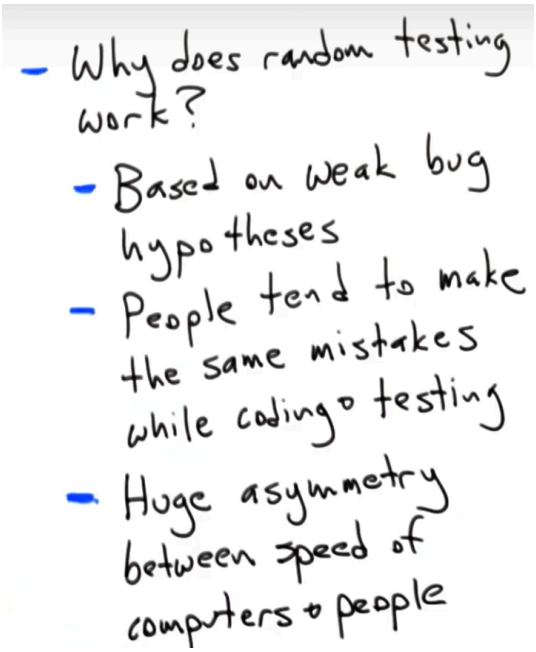
Random testing in practice

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

1. Introduction



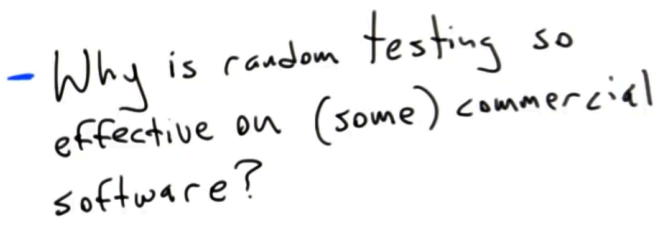
2. Random testing in the bigger picture



•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.

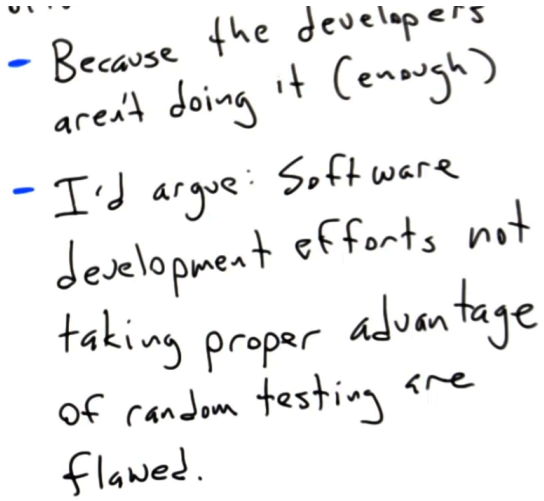
•If we forgot to implement some feature or if we miss-implemented on feature, we won't test the things done wrong. Random testing to some extent get us out of this problem because it can construct test cases to test things that we don't actually understand or know very well.

- 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.



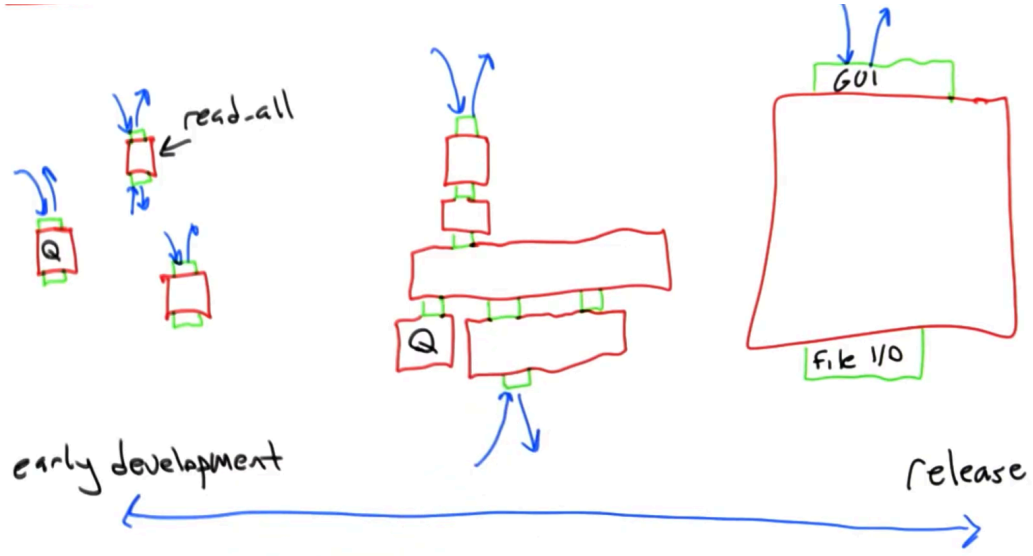
•There's pretty ample evidence, for example the fuzzing papers that we discussed or the Charlie Miller's talk that you can watch online, Babysitting an army of monkeys.

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:

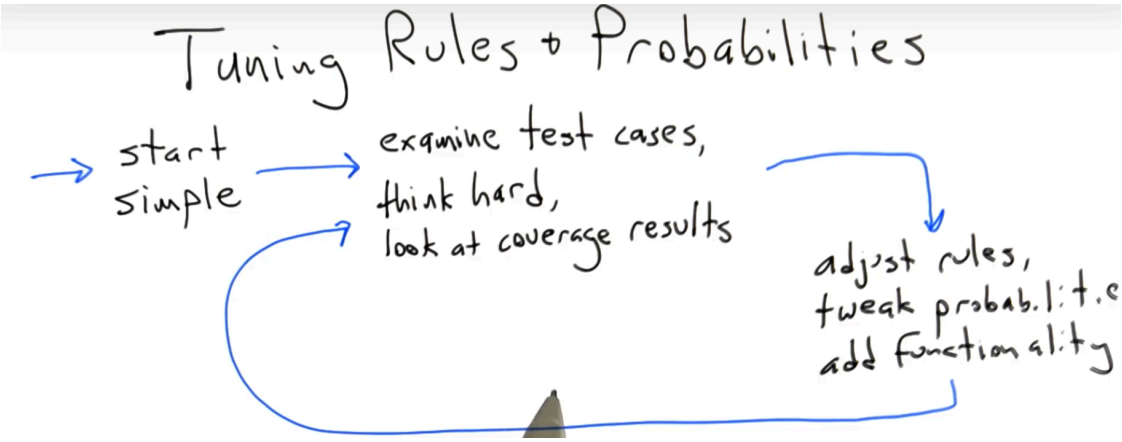


- We want to ensure as we're developing the modules that we're creating robust pieces of software whose interfaces we understand and that will be solid foundation for future work.
- It's going to be the case that some of our random testers become useless (e.g. the bounded queue) but others (e.g. those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

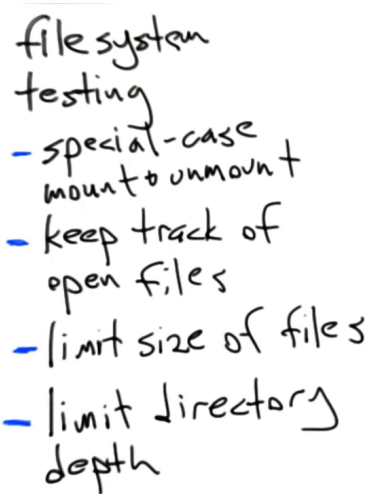
4. Random system testers

- 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



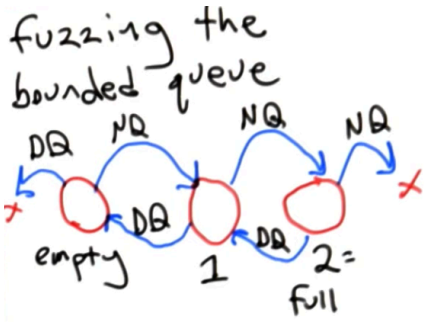
•If we start with a simple file system tester, we'll make a list of all the API calls that we'd like to test then call them randomly with random arguments.

•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.

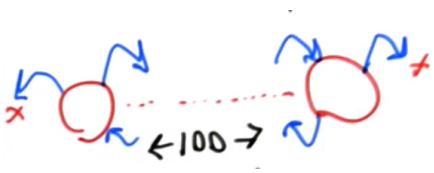
- Let's look at the queue as a FSM (FSM stands for finite state machine). It contains 3 nodes.



- 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

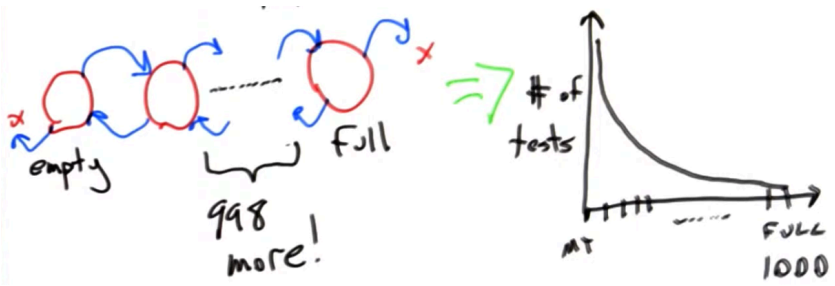
- **We go back to the random tester presented in the Random testing lecture at section 26.** We modify the code to make a 2 element queue and then run the random tester. We get a crude coverage metric. Similar information we get using a branch coverage tool.
- Let's say that we had a queue containing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

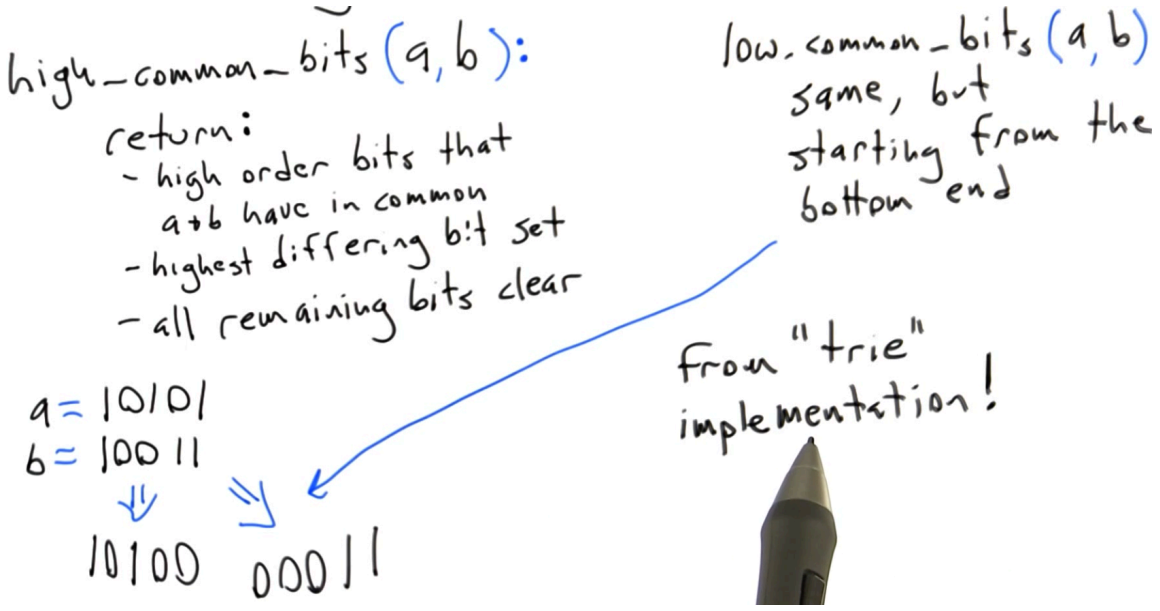
- We took a random testing loop and we **enclose it in a larger random testing loop** and in that larger loop, we made qualitative change to one of the probabilities, i.e. we bias execution towards one of our API calls in favor of the other and on even-numbered calls, we biased it the other way.

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 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.

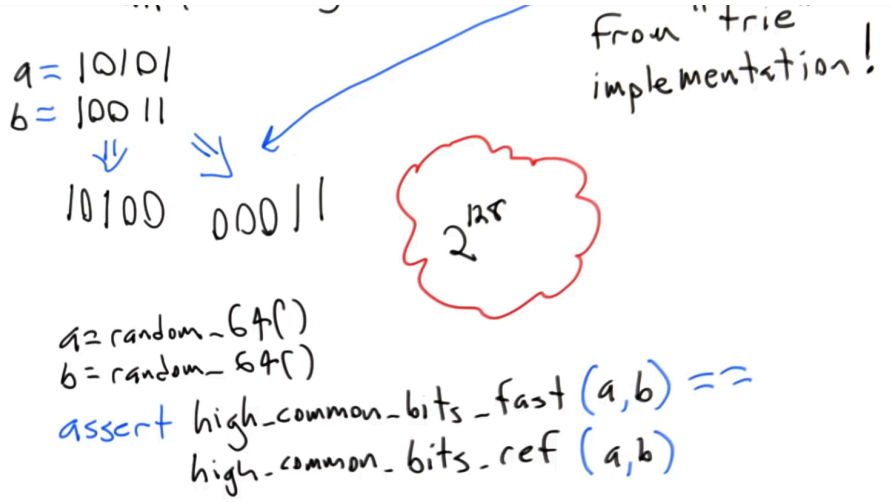
12. Bitwise implementation

- 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 |= 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

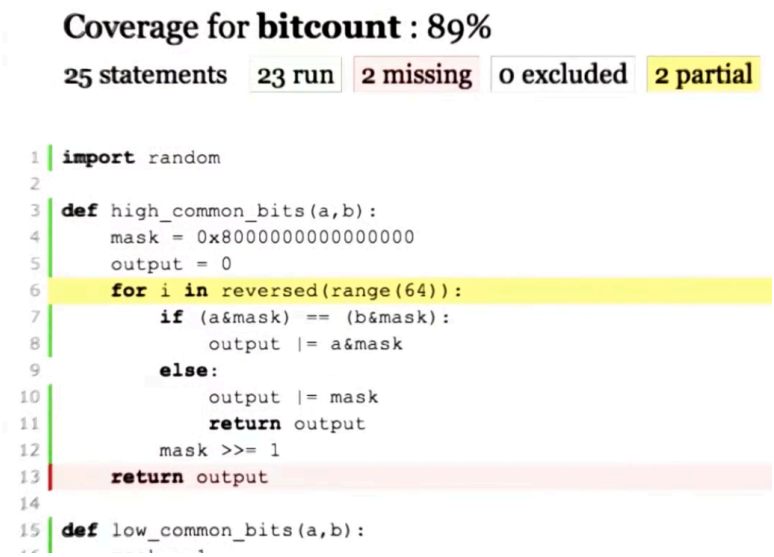
- Since all w have is the reference implementation, we will not test 2 implementations against each other, rather we will make up 2 random 64-bit integers and just run the high_common_bits and low_common_bits function on them.

```
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 a completely random and then flipping a random number of bits, i.e. b is a mutated function of a.
- Number of total random test 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%

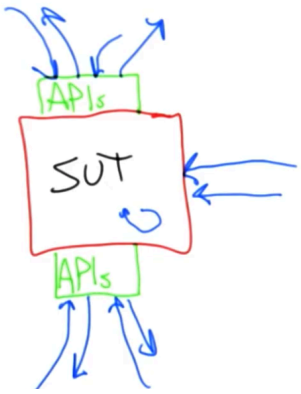
27 statements	27 run	0 missing	0 excluded	0 partial

15. Domain specific knowledge

- 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 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 i**t, and this is a fundamental limitation of random testing.

16. Fuzzing implicit inputs

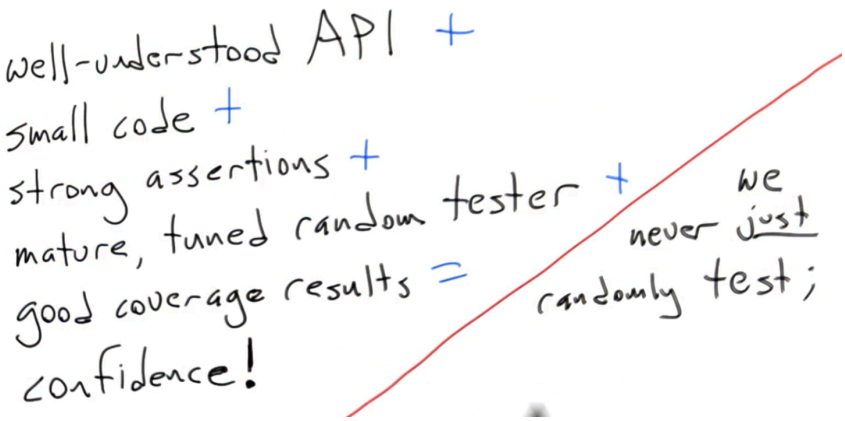
- Recall that the SUT provides APIs and most of the time that's we're fuzzing. On the other hand, the SUT uses APIs. In the following we talk about fuzzing of the implicit inputs where we have non-API inputs that affect SUT's behavior. 3 important techniques:



- 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