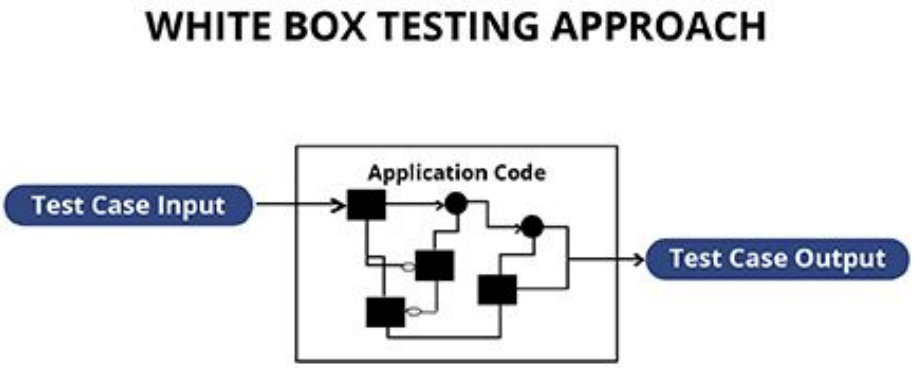
Lecture Notes
October 16, 2018

Black box testing focuses on inputs and outputs, without concern for how the software under test transforms the inputs into outputs

In contrast, white box testing focuses on "*covering*" that transformation, particularly control-flow execution paths = the sequence of instructions executed for a given input to produce the actual output



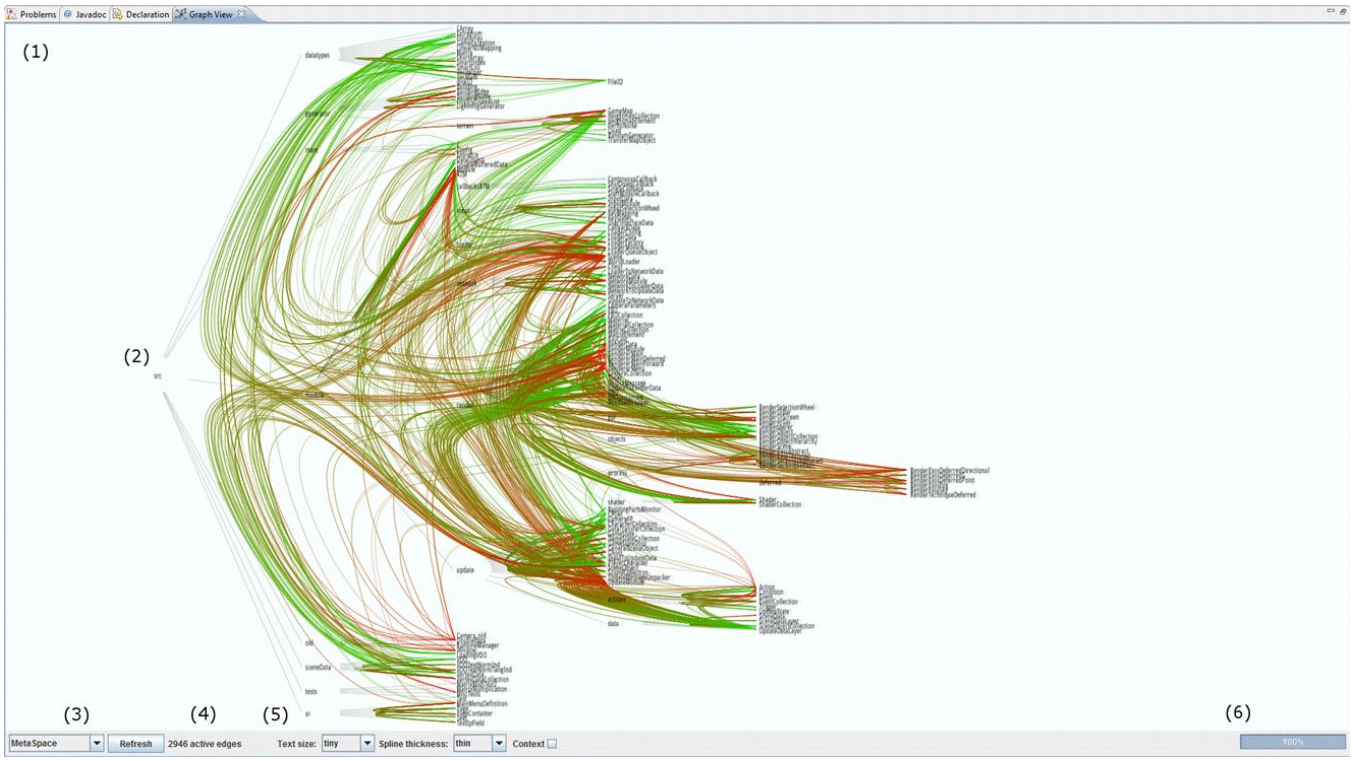
Intuition: If you never tested some segment of the code, you can't have any confidence that it works

Also answers the question "when can I stop testing?" = when all the code has been covered (or 80%, 90%, etc.)

Since not feasible to test *all* execution paths for non-trivial programs, the tester needs to select a practically small number of paths that, collectively, are *most likely* to reveal bugs

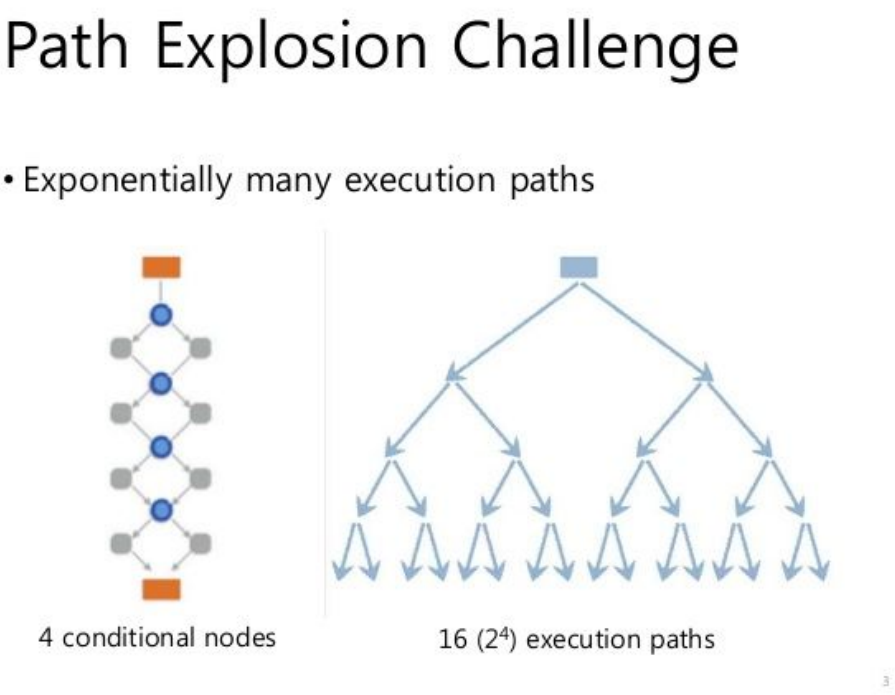
An execution path from beginning to end runs through all the methods invoked during that execution

But control flow graphs are large and complicated



So recording of execution paths often restricted to intraprocedural, rather than interprocedural (that is, only consider paths from begin of method to end)

But even the number of execution paths within a single method can be too many if there are many branches



So most coverage tools support at most if-else *branch coverage*, which checks that the test suite executes both true and false for every condition

This means there is at least one test where the condition is true and at least one test where the condition is false, not (usually) the same test

Some support only statement coverage or function coverage - lesser criteria that are easier to achieve

Most coverage tools do not support more sophisticated branches:

Loop statements are multi-way branches, perhaps ∞ many, but we need to stop somewhere so typical rule of thumb is zero times, one times, more than one time through loop

Polymorphism in object-oriented languages allows "hidden" branches, one for each class of object that can be substituted

"Hidden" branches may also be created via in-lining and other compiler optimizations

Condition coverage is not much harder to implement than branch coverage, but rarely used outside safety-critical software

Why would we want condition coverage?

```
boolean purchaseAlcohol (int buyerAge, int ageFriend)
{
        boolean allow = False;
        if ((buyerAge>=21) or (ageFriend >= 21))
        {
        allow = True
        }
        return allow;
}
```

Assert purchaseAlcohol(25,25) == True

gives 100% *statement* coverage

Assert purchaseAlcohol(25,25) == True

Assert purchaseAlcohol(20,20) == False

gives 100% *branch* coverage

But *both* the buyer *and* the friend must be 21 or older. Does branch coverage detect the bug?

For condition coverage, conditions must evaluate to T or F in all combinations (truth table)

purchaseAlcohol(25,25) == True covers T, T
purchaseAlcohol(20,20) == False covers F, F
purchaseAlcohol(25,20) == False covers T, F
purchaseAlcohol(20,25) == False covers F, T

Does condition coverage detect the bug?

➔Demo Coverage Tools (Ashna): Jasmine test framework for Javascript with Travis CI, with Instanbul code coverage tool and reports displayed on Codecov

The tester needs to devise a set of tests that achieve high coverage, for whatever coverage metric is chosen

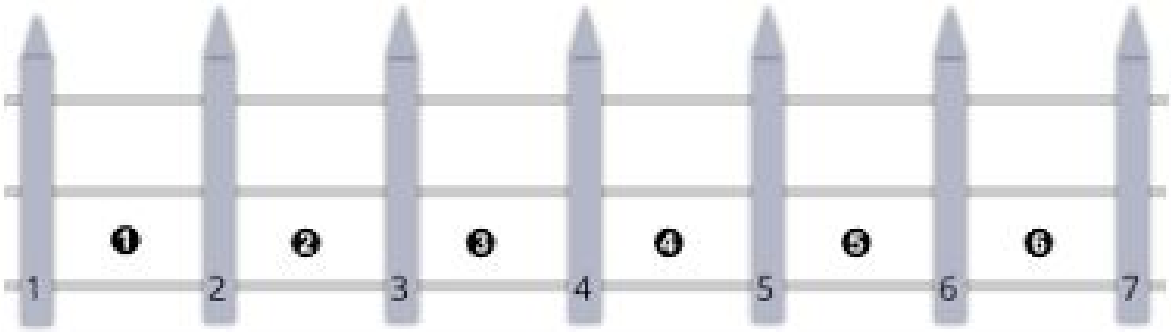
Many branches are covered "for free" by the equivalence classes tested during black box testing

For example, one test case *inside* an equivalence class and another test case *outside* that equivalence class will typically exercise the true and false branches, respectively, of at least one condition somewhere in the codebase

What would it mean if there was no such condition?

Extending equivalence classes with *boundary analysis* adds more "for free" branches covered (per applicable equivalence class)

Errors often occur at the *boundaries* of ranges or ordered sets that correspond to equivalence classes: off-by-one error, fencepost error



Seven fenceposts are required to complete a six-section fence

Boundary analysis aims to check the boundaries of each equivalence class in addition to checking a value "in the middle"

> min, min+1, max, max-1 → what about min-1 and max+1?

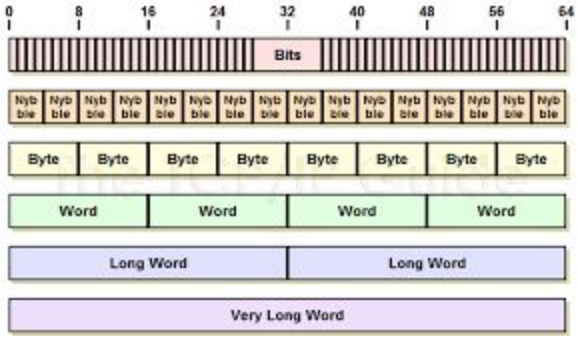
This function searches the list parameter for the searchItem. If found, it returns the location in the list; if not found, it returns -1

What are the equivalence classes for this code?

```
template <typename T>
int seqSearch(const T list[], int listLength, T searchItem)
{
   ...
}
```

What are the boundaries?

Additional boundaries consider how data is represented in memory - maybe "hidden" boundaries when going from one byte to one word, from single to double

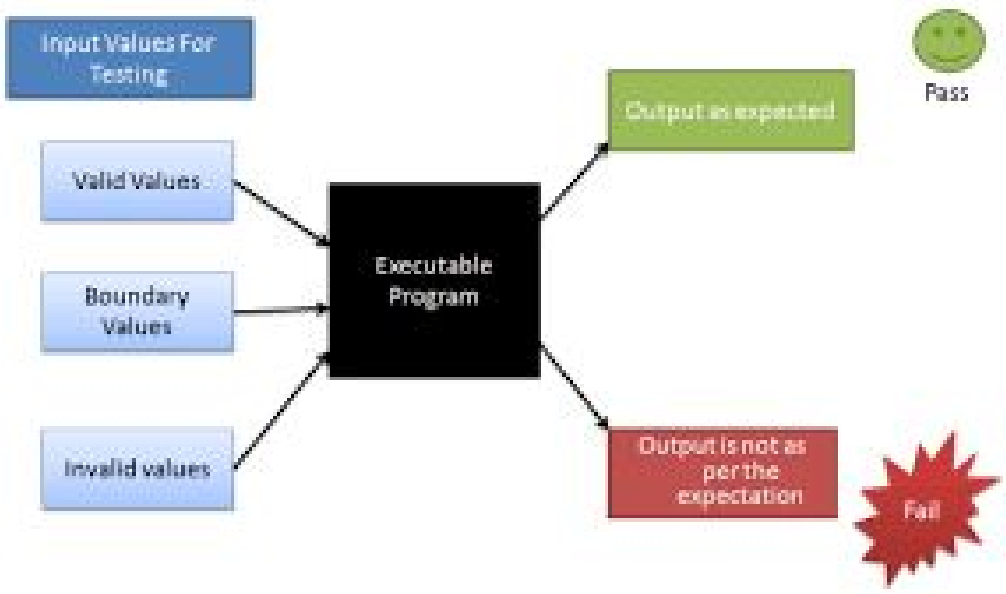


How/why would the size of the data matter?

What are the boundaries for ascii?

ASCII value	Character	ASCII value	Character	ASCII value	Character
000	^@	043	+	086	V
001	^A	044	,	087	W
002	^B	045	-	088	X
003	^C	046	.	089	Y
004	^D	047	/	090	Z
005	^E	048	0	091	[
006	^F	049	1	092	\
007	^G	050	2	093	]
008	^H	051	3	094	^
009	^I	052	4	095	_
010	^J	053	5	096	`
011	^K	054	6	097	a
012	^L	055	7	098	b
013	^M	056	8	099	c
014	^N	057	9	100	d
015	^O	158	:	101	e
016	^P	059	;	102	f
017	^Q	060	<	103	g
018	^R	061	=	104	h
019	^S	062	>	105	i
020	^T	063	?	106	j
021	^U	064	@	107	k
022	^V	065	A	108	l
023	^W	066	B	109	m
024	^X	067	C	110	n
025	^Y	068	D	111	o
026	^Z	069	E	112	p
027	^[	070	F	113	q
028	^\	071	G	114	r
029	^]	072	H	115	s
030	^^	073	I	116	t
031	^-	074	J	117	u
032	[space]	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

Table 4.2: ASCII Table



Are equivalence classes plus boundary analysis, including invalid classes and at/outside the boundaries, sufficient to cover all branches? Maybe, maybe not

If we track coverage during execution of black box testing, then we can find out what has **not** been covered yet and invent additional tests that *force* the not yet covered branches

What tests would be needed to cover every statement?
What tests would be needed to cover every branch?

```
template <typename T>
int seqSearch(const T list[], int listLength, T searchItem)
{
   int loc;

   for (loc = 0; loc < listLength; loc++)
      if (list[loc] == searchItem)
         return loc;
   return -1;
}
```

How would we cover every statement and/or every branch if we only used full-system testing, not unit testing?

How could we check coverage?

```
#define COVER cerr << "Block " << __FILE__ << ":" << __LINE__ << endl
template <typename T>
int seqSearch(const T list[], int listLength, T searchItem)
{
   <[+]>COVER;<[-]>
   int loc;

   for (loc = 0; loc < listLength; loc++)
      if (list[loc] == searchItem)
         <[+]>{
          COVER;<[-]>
          return loc;
         <[+]>}<[-]>

   <[+]>COVER;<[-]>
   return -1;
}
```

Not too hard for statements, but tedious. Much harder for branches. Better to use coverage tools (or compiler options) that automatically insert coverage-checking code, rather than manual

Coverage tools *instrument* the code to track what has already been covered versus not yet covered as the test suite executes - and at the end inform the tester what percent has been covered (0 <= covered <= 100%) and which specific branches are yet covered. Usually also count how many times covered elements executed

What tests would be needed to cover every statement?
What tests would be needed to cover every branch?

```
int BidderCollection::add (const Bidder& value)
{
  if (size < MaxSize)
    {
     addToEnd (elements, size, value);
     return size - 1;
    }
  else
    {
     <[+]>cerr << "BidderCollection::add - collection is full" <<
endl;<[-]>
     exit(1);
    }
}
```

How would we cover every statement and/or every branch if we only used full-system testing, not unit testing?

There are research tools that generate test suites that (ideally) include at least one test case that covers each branch (e.g., Evosuite)

Let's say we had such tools, why not use only white box testing, not black box?

White box testing cannot catch errors of *omission*, where the code to implement required functionality does not exist

Midterm Individual Assessment will be posted later today, due next week on Tuesday, October 23, 11:59pm

★hard deadline, do not be late!

I will answer questions in class on this Thursday. Come to class if you have any questions. CVN students should post questions on piazza before class time on Thursday. I will not answer questions after Thursday.

Next team assignment First Iteration due Thursday, October 30, 11:59pm. Follow-up team assignment First Iteration Demo due Thursday, November 8, 11:59pm