

# COMS W4156 Advanced Software Engineering (ASE)

October 21, 2021

[shared google doc for discussion during class](#)

Have all Teams Scheduled to Meet with their IA Mentor?

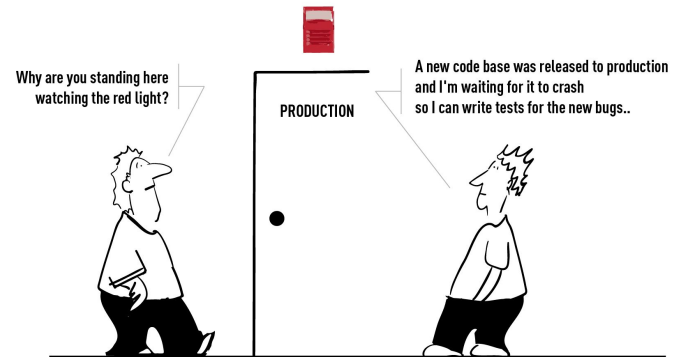
# Baseline Testing Strategy

- *Test all individual units followed by big bang system testing, blackbox-only at both levels with typical valid, atypical valid and invalid input*

It's better than no strategy

It's better than the common CS1 strategy - test program with two or three typical valid inputs, done

Think of it as just a starting point



# Better Approaches to Choosing Test Inputs

If you have an automated coverage tool, or your program is small enough to check manually, try to achieve 100% coverage - but full coverage does not test missing code (more on coverage in a later lecture)



Test with a set of inputs that you think would make a nice demo - but what happens when the demo audience asks to “drive”? If this is higher level management in your company, your venture capital investor, or the instructor/TA who decides your grade, you cannot say No



# More Approaches to Choosing Test Inputs



If you have user stories with conditions of satisfaction, add test cases to check that all the conditions are indeed satisfied

If you have use cases with individual steps for happy path, alternative and exceptional flows, add test cases to exercise each step of every flow



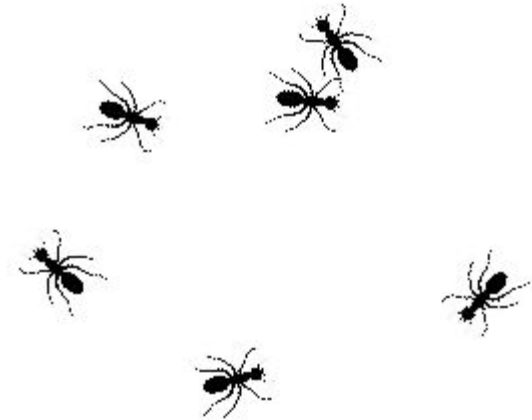
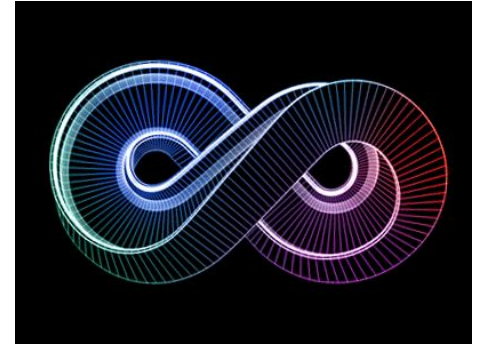
# How About Testing ALL Possible Inputs?

Usually “infinite” number of valid and invalid inputs  
(assuming infinite size input buffers available)

Even if number of valid and invalid inputs is finite,  
infeasible to try all of them

To maximize the chances of finding and fixing  
bugs, we need to choose a practically small  
number of inputs that, collectively,

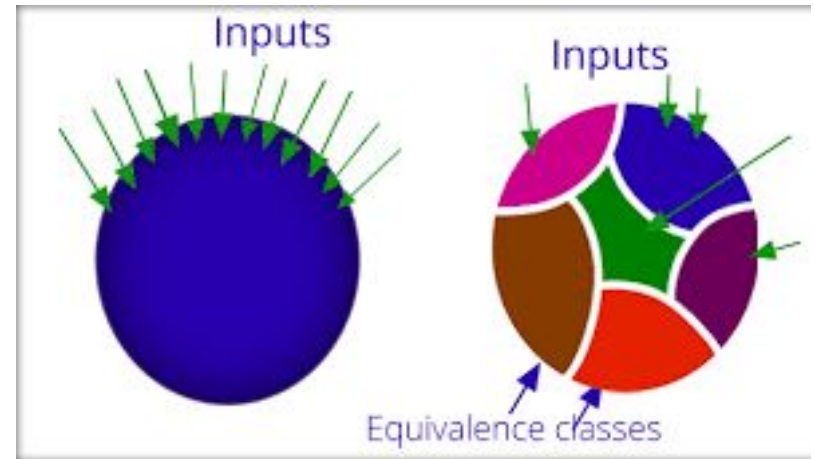
- represent all possible inputs and
- are *most likely* to reveal bugs



# Equivalence Partitions

The simplest approach for representing all possible inputs to a “box” with a practically small number of examples is to divide the input space into *equivalence partitions* (also known as equivalence classes) based on **both**

- application/domain knowledge (e.g., the user stories and/or use cases)
- software engineering knowledge

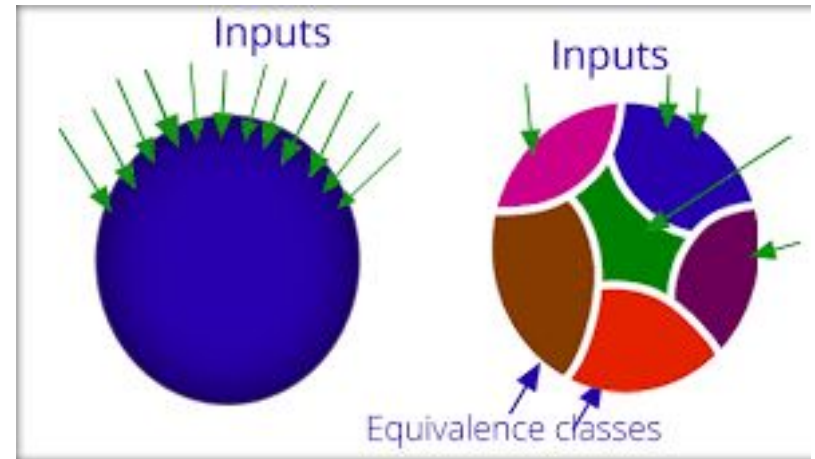


# Equivalence Partitions

Equivalence partition = set of alternative input values where the software can reasonably be expected to behave equivalently (similarly)

This does **not** mean literally the same output for every input in the class.

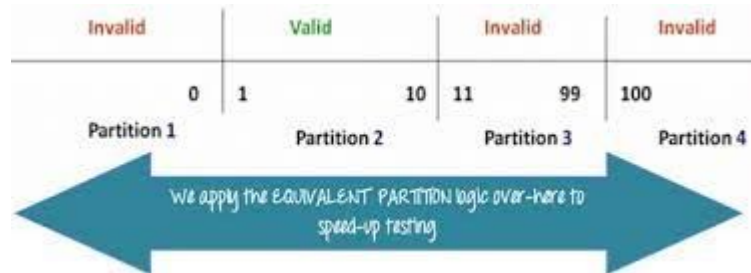
Instead it means that if the software behaves correctly on one example input chosen from that equivalence class, it is reasonable to believe it will behave correctly on all the other inputs in the same class





# Simple Example

If we are testing a “box” intended to accept numbers from 1 to 1000000, then there is no point in writing a million test cases for all 1000000 valid input numbers - and also no point in writing many more test cases for all invalid input numbers between MinInt and MaxInt

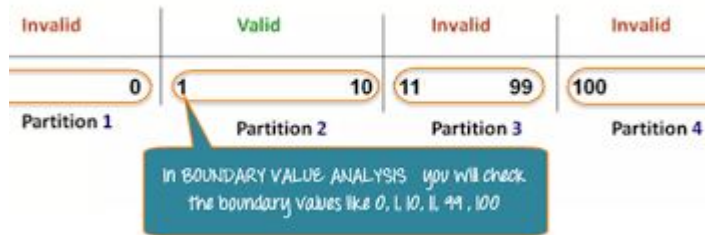


Instead, divide the test cases into three sets:

- 1) Input partition with all valid inputs. Pick a single value from range 1 to 1000000 as a valid test case. If we select other values between 1 and 1000000 the result “should” also be correct (or also be incorrect)
- 2) Pick a single value for the input partition with all values *below* the lower limit of the range, i.e., any value below 1
- 3) Pick a single value for the input partition with all values *above* the upper limit, i.e., any value above 1000000

# Boundary Analysis

Equivalence partitions are typically extended with *boundary analysis* because input values at the extreme ends of the valid and invalid partitions are more likely to trigger bugs, e.g., “off by one”, “corner case”



- 1) Test cases with input exactly at the boundaries of the valid partition, i.e., values 1 and 1000000
- 2) Test data with values just below the extreme edges, i.e., values 0 and 999999
- 3) Test data with values just above the extreme edges, i.e., values 2 and 1000001
- 4) For numerical domains, make sure to include zero, positive and negative examples, as well as the underlying primitive extremes (integers). In this case, we already have 0 and some positive values, so we need to add some negative value, e.g., -1, plus MinInt and MaxInt
- 5) For numerical domains, also try some non-numerical input, e.g., “abcde” (if it is possible to provide such inputs)

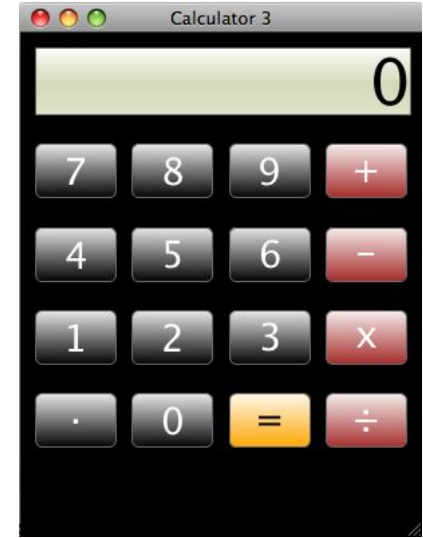
# Another Example

Consider a calculator implemented in software, with a GUI as shown. The user clicks the buttons with a mouse, but cannot enter text into the display

In mathematics, the “+” operation takes two numerical operands

Say this software is tested by an external testing team separate from the developers

Based on *domain knowledge*, a tester might reasonably expect the “+” operation to behave similarly on all pairs of numbers  
< first operand, second operand >

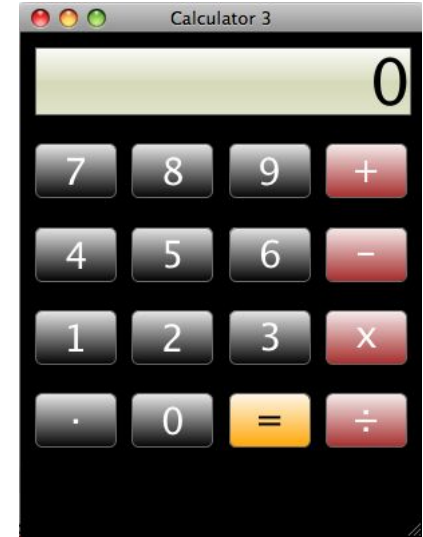


## Another Example

So the tester tries an arbitrary pair of numbers:

< 3.141592653, 42 >

The calculator flashes 00000000. What does this mean?



# Interpreting Test Outputs

Based on *software engineering knowledge*, the tester might realize the flashing 00000000 could indicate:

- An error message because the software cannot handle numbers with more than N digits (e.g., 8)
- An error message because the software cannot handle mixed operands, i.e., one operand is floating point and the other is integer
- A bug
- Something else (e.g., the user has reached the limit of their “free trial” and now has to pay to continue using the calculator software)



# Continuing Calculator Example

Let's ignore the “.” key (planned for version 2.0) and assume the “+” operator is currently intended only for integer operands

A tester with software engineering knowledge, not just domain knowledge, might realize that a calculator implementation could behave differently for positive integers, zero, and negative integers

Thus they test with five different equivalence classes:

- < pos int, pos int >
- < pos int, neg int >
- < pos int, 0 >
- < neg int, neg int >
- < 0, 0 >

Is this sufficient?

Although the “+” operator is mathematically commutative, that does not mean that the implementation (which could be buggy) is commutative

Or, more generally, define equivalence partitions separately per parameter, and then extend to the cross-product

But notice the scaling problem with large numbers of parameters!

- < pos int, pos int >
- < pos int, neg int >
- < pos int, 0 >
- < neg int, pos int >
- < neg int, neg int >
- < neg int, 0 >
- < 0, pos int >
- < 0, neg int >
- < 0, 0 >

Let's say the calculator is intended to handle operands only up to 8 digits (or to any other fixed number of digits)

This means there is a MaxInt and a MinInt (negative), probably specific to what the calculator GUI was designed to display rather than MaxInt/MinInt of the programming language, compiler, operating system, etc.

What additional tests are needed?

- < xxx, MaxInt >
- < MaxInt, xxx >
- < xxx, MinInt >
- < MinInt, xxx >
- < MaxInt, MinInt >
- < MinInt, MaxInt >

Any more?



# Invalid Inputs to Calculator Example



What if the user could enter text, using a conventional keyboard, into the display window in addition to mousing the keys

Does that change the set of equivalence partitions that need to be tested?

## Equivalence Partitioning Examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: $-99 \leq N \leq 99$	$[-99, -10]$ $[-9, -1]$ 0 $[1, 9]$ $[10, 99]$	$< -99$ $> 99$ Malformed numbers {12-, 1-2-3, ...} Non-numeric strings {junk, 1E2, \$13} Empty value
Phone Number  Area code: [200, 999]	555-5555 (555)555-5555 555-555-5555 $200 \leq \text{Area code} \leq 999$	Invalid format 5555555, (555)(555)5555, etc.  Area code $< 200$ or $> 999$ Area code with non-numeric characters

## Example of Equivalence Partitions

- Windows filename can contain any characters except \ : \* ? " < > |.
- Filenames can have from 1 to 255 characters.
- Creating test cases for filenames:
  - Will have equivalence partitions for valid characters, invalid characters, valid length names, names that are too short and names that are too long.



# More Examples

What are the  
equivalence partitions?

How would we test this  
program?

## Design-your-burger

Please make your selection

Burger:

Cooked: ☐ rare ☐ medium ☐ well

Cheese: ☐

Lettuce: ☐

Tomato: ☐

Onion: ☐

Ketchup: ☐

Mustard: ☐

Mayo: ☐

Secret sauce: ☐



[Nutrition facts](#)   [About the chef](#)

# Required vs Optional Inputs

Get

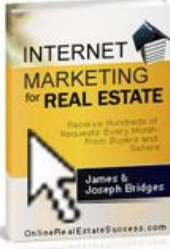
**"Internet Marketing for Real Estate EBook"**

via e-mail

Name\*:

Email\*:

Phone\*:



\* Indicates a required field

\* Indicates an optional field

Equivalence classes need to address required vs. optional inputs. with or without default values

What are the equivalence partitions?

How would we test this program?

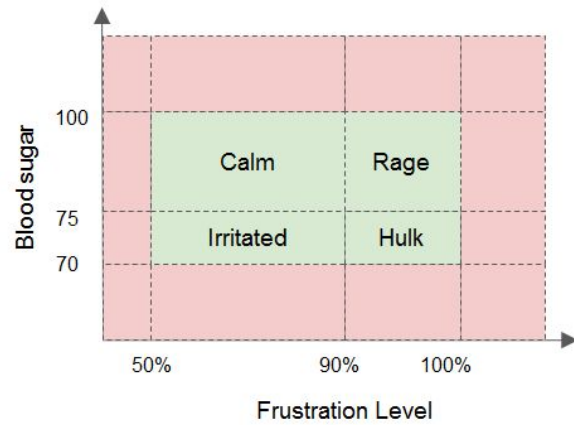
# Dependencies between Inputs

Distinct parameters may be related to each other in forming equivalence classes

Let's say we had a function that took two variables (blood sugar level and level of frustration) to calculate "mood"

Assume valid inputs are  
Blood sugar 70 - 100  
Frustration is 50% - 100%

For  $50 \leq F \leq 90$   
     $75 \leq BS \leq 100$ : Calm  
     $BS < 75$ : Irritated  
For  $F > 90$   
     $75 \leq BS \leq 100$ : Rage  
     $BS < 75$ : Hulk Smash



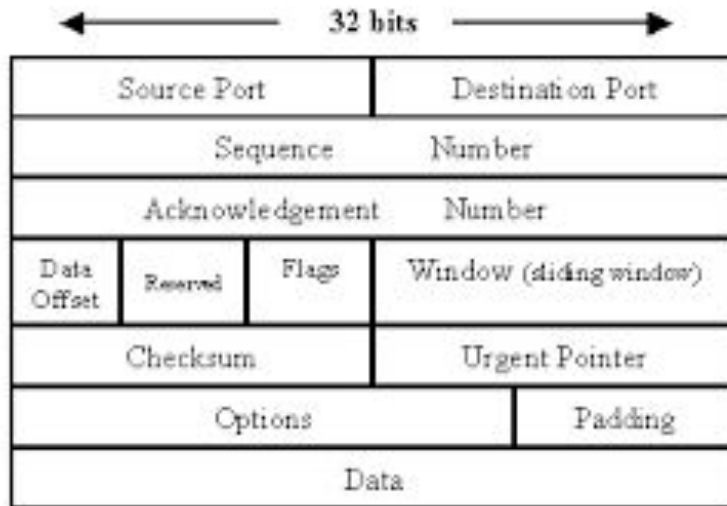
# More Dependencies between Inputs

Equivalence classes may cut across multiple inputs:

- Sometimes the value of one input restricts appropriate values for other inputs
- For example, date of birth and age should be consistent (valid) or not (invalid)
- Another example, the checksum in the network packet



# System to System Inputs (includes API inputs)



# System and Component Inputs

At the full system and component levels, inputs are generally going to be numbers, strings (text), tuples (packets), and files

File inputs can:

- Exist or not
- Be below min or above max size
- Be readable, writeable, executable by current user (permissions)
- Correct, corrupted, garbled, etc. data and metadata formats - images, audio, video, ...

# Unit Inputs

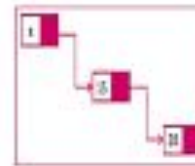
At the unit testing level, inputs/outputs can be arbitrary data structures, so equivalence classes can be more complex - and inputs more complicated to construct

## Container data structures with content elements:

- In or not in container
- Empty or full container (or min/max number of elements)
- Duplicate elements may or may not be allowed
- Ordered or organized correctly, or not



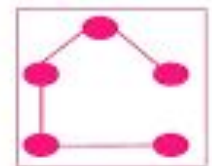
## Sorting



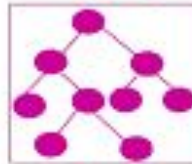
## Link list



```
list
```



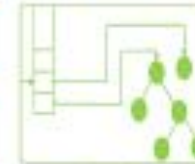
spanning tree



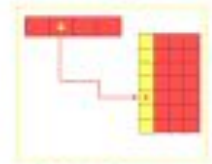
## Tree



### Graph

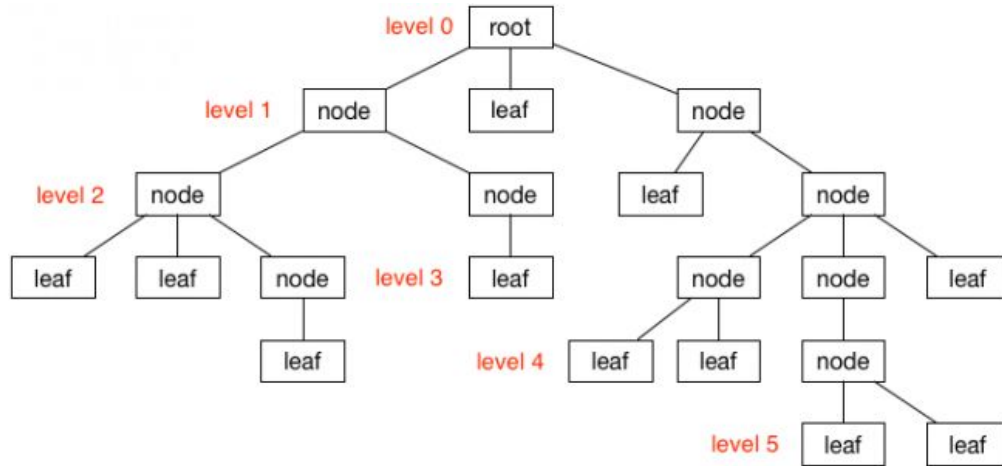


Stack



## Hashing

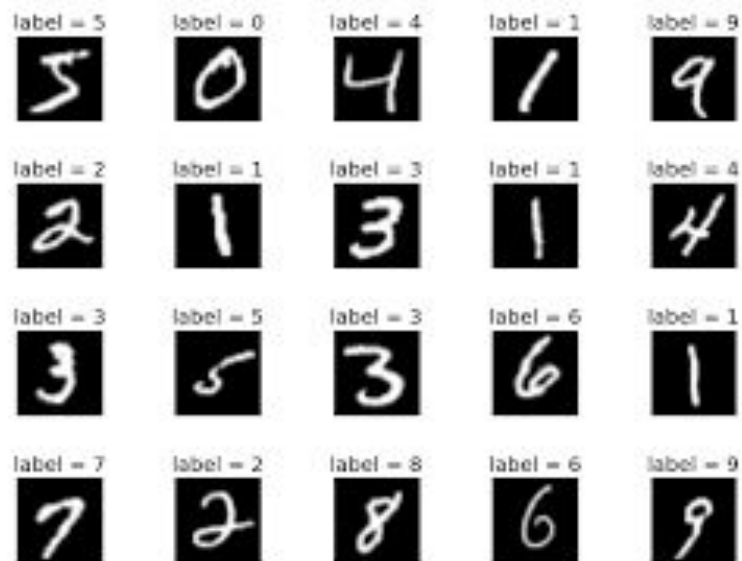
# More Unit Inputs



Specific kinds of containers, e.g., tree - root node, interior node, leaf node, null tree, tree with exactly one node, “full” tree, balanced vs. unbalanced, various specialty trees - and graphs

How would test cases construct root, node and leaf objects to use as inputs?

# Equivalence Partitions for Data Sets



Files containing “data sets” often correspond to a table (rows and columns)

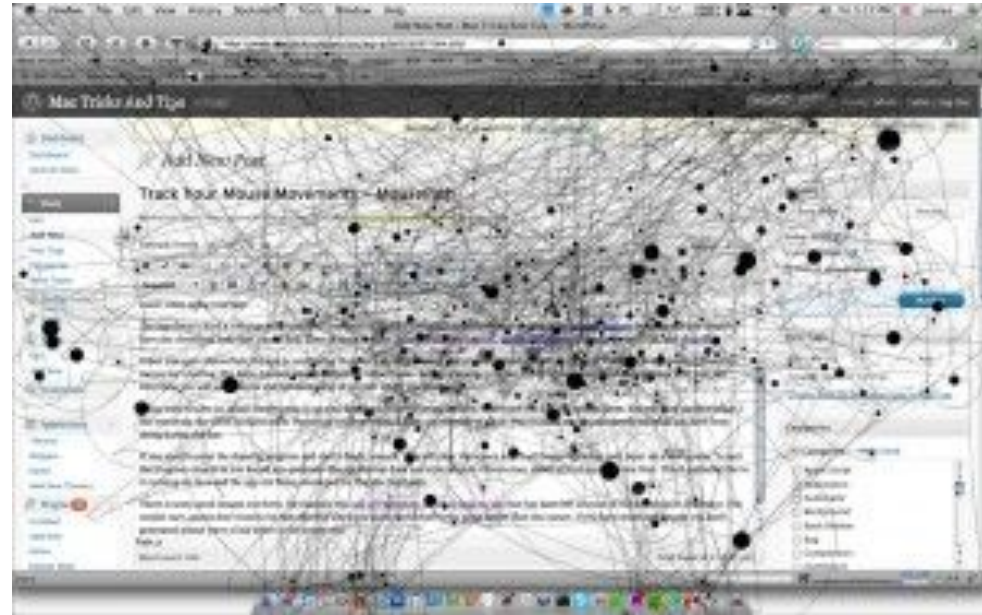
This shows part of a well-known dataset for deep learning algorithms ([MNIST](#))

What are the equivalence classes?

# Equivalence Partitions with Advanced GUIs

For some GUI applications, inputs may include mouse movements, locations or clicks, manual (touch-sensitive) gestures, camera images, gps coordinates, etc.

What might be some valid and invalid equivalence classes for mouse activity?



# In-Class Exercise if time permits

Recall the CONTAIN app we have been working on, **CON**tact Tracing for instruction **AssIstaN**ts, which does “contact tracing” based on IA (instruction assistant) connections. An IA for xxx course is a contact of all the other IAs for xxx as well as for every student who takes xxx. Since an IA is also a student, they are a contact for all other students in every course they take as well as for all the IAs of those courses.

CONTAIN leverages APIs provided by SSOL, Canvas (courseworks), and a drone-based attendance-recording service, and tries to avoid false positives and security breaches.

Pick one of these three APIs and figure out what are its input equivalence classes and boundary conditions. You can “make up” plausible API entry points.

# In-Class Exercise

timer...



# Team Project

[Assignment T2: Revised Project Proposal](#) due Monday

Meet with your IA mentor *before* submitting the revised proposal

You can and should start coding as soon as possible after the meeting

# Midterm Assessment

Will appear in courseworks at 12:01am on day 1, will close on courseworks at 11:59pm on day 4 (this may in practice be 12:02 and 11:58)

NOT “timed”, you can spend as much or as little time as you want ( $\leq 4$  days)

There will be several graded questions and two ungraded questions

None of the questions involve writing or running code (other than your text editor)

One ungraded question asks for feedback on the course, this is *optional*

The other ungraded question asks for feedback on your team, this is *required* - if you do not provide a meaningful response, we will not grade the graded questions and you will receive zero as your score on the assessment

# Required Ungraded Question

Allocate 100 points among your team members, including yourself, based on contributions thus far to the team project

And *explain* your point allocations

An answer like

“Me: 25, team member A : 25, team member B: 25, team member C: 25.  
We shared the work equally.”

is not sufficient. Describe who did what (so far, please do not predict the future).