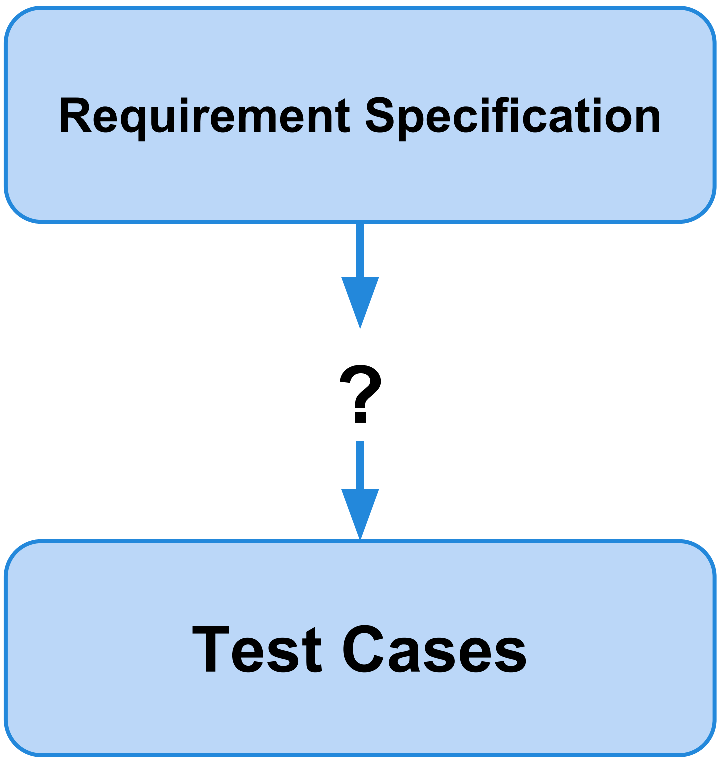
Developing Requirements-Based Tests

CSCE 740 - Lecture 8 - 09/21/2017

Partitioning



- Functional testing is based on the idea of **partitioning**.
  - You can't actually test individual requirements in isolation.
  - First, we need to partition the specification and software into features that can be tested.
  - Not all inputs have the same effect.
  - We can partition the outputs of a feature into the possible outcomes.
    - and the inputs, by what outcomes they cause (or other potential groupings).

Creating Requirements-Based Tests

Write Testable Specifications

Produce clear, detailed, and testable requirements.

Identify Independently Testable Features

Figure out what functions can be tested in (relative) isolation.

Identify Representative Input Values

What are the outcomes of the feature, and which input classes will trigger them?

Generate Test Case Specifications

Identify abstract classes of test cases.

Generate Test Cases

Instantiate concrete input/output pairs.

Today’s Goals

- How to define and select requirements-based tests
  - Choosing representative input values.
  - Creating abstract test case “specifications”
  - Filling in the concrete input values.

Calculator Requirement

- Requirement 7.63: Divide-By-Zero
    - When a 0 is provided as input, it should be intercepted. Division-by-zero indicates an unsolvable expression.

Any problems?

- Input to what? Anything?
- Intercepted?

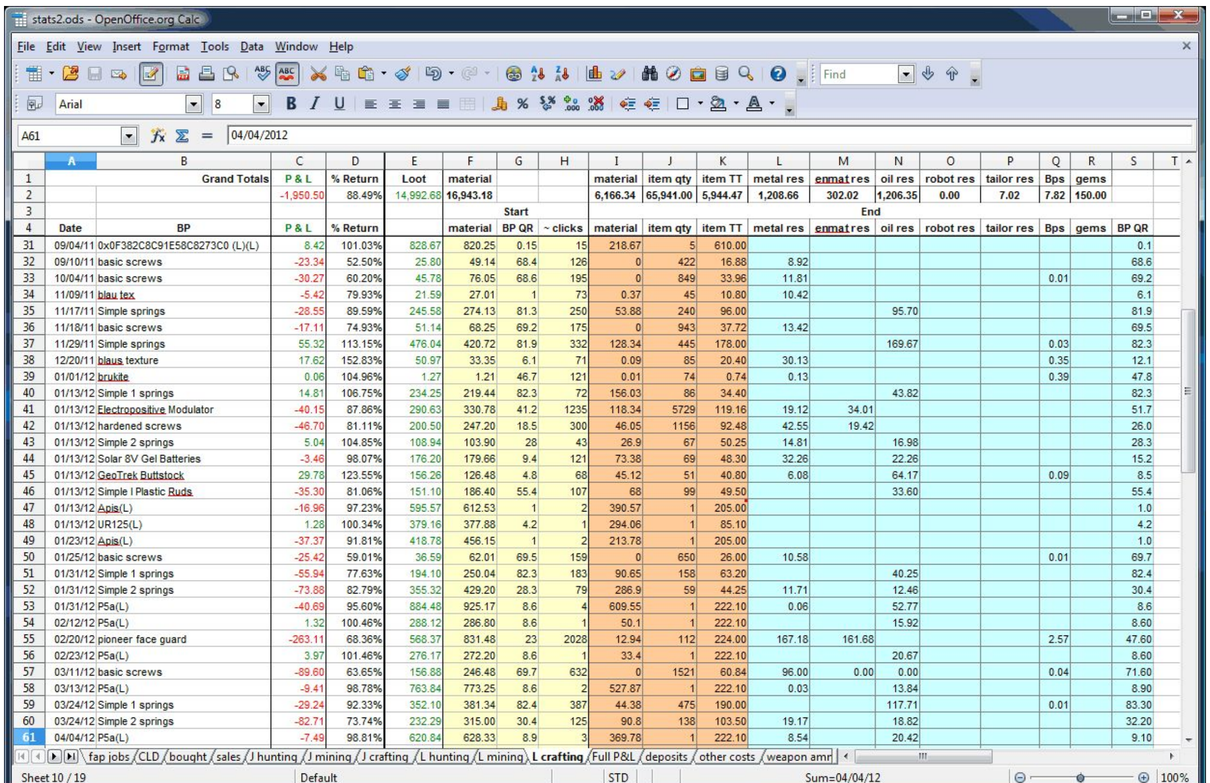
Calculator Requirement (Take 2)

- Requirement 7.63: Divide-By-Zero
  - When a 0 is provided as input as the divisor in any use of the division function, the software shall issue an error message indicating that this is an unsolvable expression.

- What are the independently testable features of a calculator?
- What are the parameters of the division feature? Their characteristics?
- How would you test that this requirement is fulfilled?

Independently Testable Features

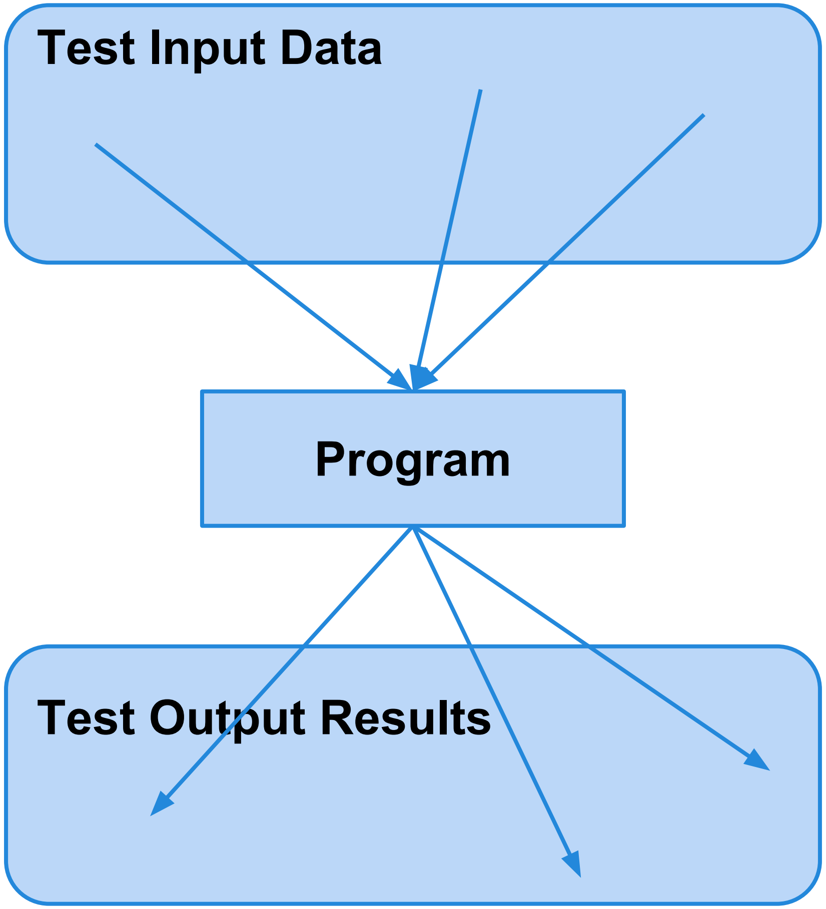
What are three independently testable features of a spreadsheet?



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1		Grand Totals	P & L	% Return	Loot	material			material	item qty	item TT	metal res	enmat res	oil res	robot res	tailor res	Bps	gems	
2			-1,950.50	88.49%	14,992.68	16,943.18			6,166.34	65,941.00	5,944.47	1,208.66	302.02	1,206.35	0.00	7.02	7.82	150.00	
3							Start						End						
4	Date	BP	P & L	% Return		material	BP QR	~ clicks	material	item qty	item TT	metal res	enmat res	oil res	robot res	tailor res	Bps	gems	BP QR
31	09/04/11	0x0F382C8C91E58C8273C0 (L)(L)	8.42	101.03%	828.67	820.25	0.15	15	218.67	5	610.00								0.1
32	09/10/11	basic screws	-23.34	52.50%	25.80	49.14	68.4	126	0	422	16.88	8.92							68.6
33	10/04/11	basic screws	-30.27	60.20%	45.78	76.05	68.6	195	0	849	33.96	11.81					0.01		69.2
34	11/09/11	blau tex	-5.42	79.93%	21.59	27.01	1	73	0.37	45	10.80	10.42							6.1
35	11/17/11	Simple springs	-28.55	89.59%	245.58	274.13	81.3	250	53.88	240	96.00			95.70					81.9
36	11/18/11	basic screws	-17.11	74.93%	51.14	68.25	69.2	175	0	943	37.72	13.42							69.5
37	11/29/11	Simple springs	55.32	113.15%	476.04	420.72	81.9	332	128.34	445	178.00			169.67			0.03		82.3
38	12/20/11	blaus texture	17.62	152.83%	50.97	33.35	6.1	71	0.09	85	20.40	30.13					0.35		12.1
39	01/01/12	brukite	0.06	104.96%	1.27	1.21	46.7	121	0.01	74	0.74	0.13					0.39		47.8
40	01/13/12	Simple 1 springs	14.81	106.75%	234.25	219.44	82.3	72	156.03	86	34.40			43.82					82.3
41	01/13/12	Electropositive Modulator	-40.15	87.86%	290.63	330.78	41.2	1235	118.34	5729	119.16	19.12	34.01						51.7
42	01/13/12	hardened screws	-46.70	81.11%	200.50	247.20	18.5	300	46.05	1156	92.48	42.55	19.42						26.0
43	01/13/12	Simple 2 springs	5.04	104.85%	108.94	103.90	28	43	26.9	67	50.25	14.81		16.98					28.3
44	01/13/12	Solar 8V Gel Batteries	-3.46	98.07%	176.20	179.66	9.4	121	73.38	69	48.30	32.26		22.26					15.2
45	01/13/12	GeoTrek Buttstock	29.78	123.55%	156.26	126.48	4.8	68	45.12	51	40.80	6.08		64.17			0.09		8.5
46	01/13/12	Simple I Plastic Ruds	-35.30	81.06%	151.10	186.40	55.4	107	68	99	49.50			33.60					55.4
47	01/13/12	Apis(L)	-16.96	97.23%	595.57	612.53	1	2	390.57	1	205.00								1.0
48	01/13/12	UR125(L)	1.28	100.34%	379.16	377.88	4.2	1	294.06	1	85.10								4.2
49	01/23/12	Apis(L)	-37.37	91.81%	418.78	456.15	1	2	213.78	1	205.00								1.0
50	01/25/12	basic screws	-25.42	59.01%	36.59	62.01	69.5	159	0	650	26.00	10.58					0.01		69.7
51	01/31/12	Simple 1 springs	-55.94	77.63%	194.10	250.04	82.3	183	90.65	158	63.20			40.25					82.4
52	01/31/12	Simple 2 springs	-73.88	82.79%	355.32	429.20	28.3	79	286.9	59	44.25	11.71		12.46					30.4
53	01/31/12	P5a(L)	-40.69	95.60%	884.48	925.17	8.6	4	609.55	1	222.10	0.06		52.77					8.6
54	02/12/12	P5a(L)	1.32	100.46%	288.12	286.80	8.6	1	50.1	1	222.10			15.92					8.60
55	02/20/12	pioneer face guard	-263.11	68.36%	568.37	831.48	23	2028	12.94	112	224.00	167.18	161.68				2.57		47.60
56	02/23/12	P5a(L)	3.97	101.46%	276.17	272.20	8.6	1	33.4	1	222.10			20.67					8.60
57	03/11/12	basic screws	-89.60	63.65%	156.88	246.48	69.7	632	0	1521	60.84	96.00	0.00	0.00			0.04		71.60
58	03/13/12	P5a(L)	-9.41	98.78%	763.84	773.25	8.6	2	527.87	1	222.10	0.03		13.84					8.90
59	03/24/12	Simple 1 springs	-29.24	92.33%	352.10	381.34	82.4	387	44.38	475	190.00			117.71			0.01		83.30
60	03/24/12	Simple 2 springs	-82.71	73.74%	232.29	315.00	30.4	125	90.8	138	103.50	19.17		18.82					32.20
61	04/04/12	P5a(L)	-7.49	98.81%	620.84	628.33	8.9	3	369.78	1	222.10	8.54		20.42					9.10

Identifying Representative Values

- We know the features. We know their parameters.
- What input values should we pick?
- **What about exhaustively trying all inputs?**



Exhaustive Testing

Take the arithmetic function for the calculator:

`add(int a, int b)`

- How long would it take to exhaustively test this function?

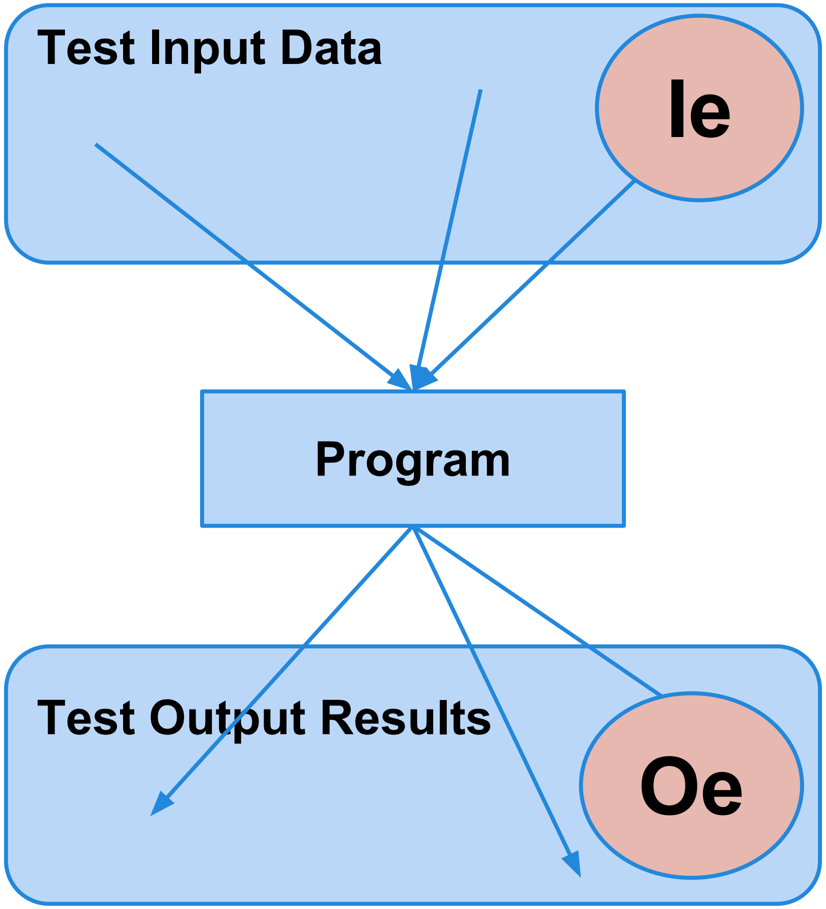
2^{32} possible integer values for each parameter.
$= 2^{32} \times 2^{32} = 2^{64}$ combinations $= 10^{13}$ tests.

1 test per nanosecond
$= 10^{5}$ tests per second
$= 10^{10}$ seconds

or… about 600 years!

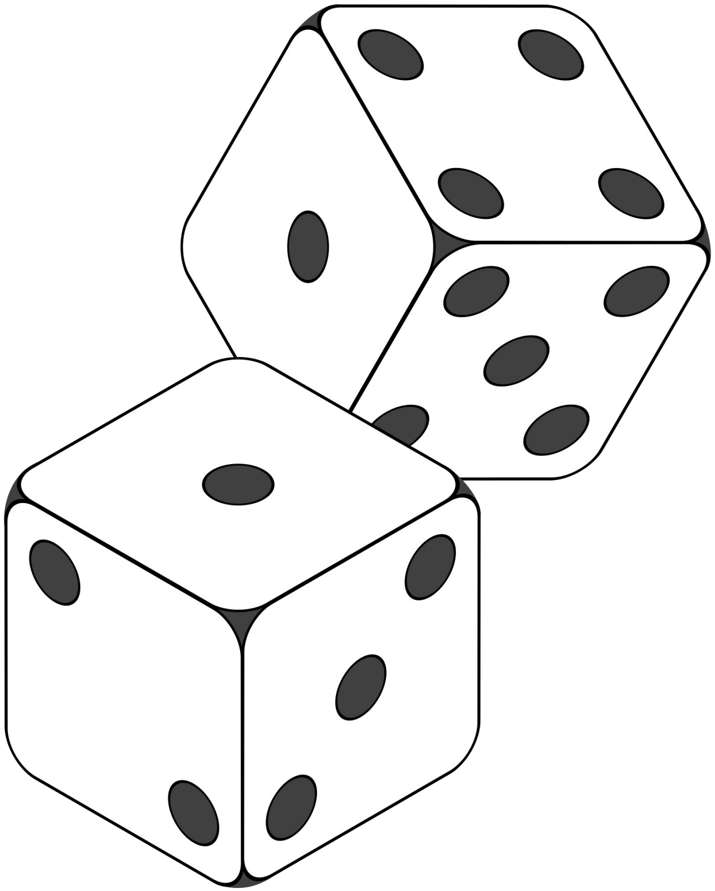
Not all Inputs are Created Equal

- We can't exhaustively test any real program.
  - **We don't need to!**
- Some inputs are better than others at revealing faults, but we can't know which in advance.
- Tests with different input than others are better than tests with similar input.



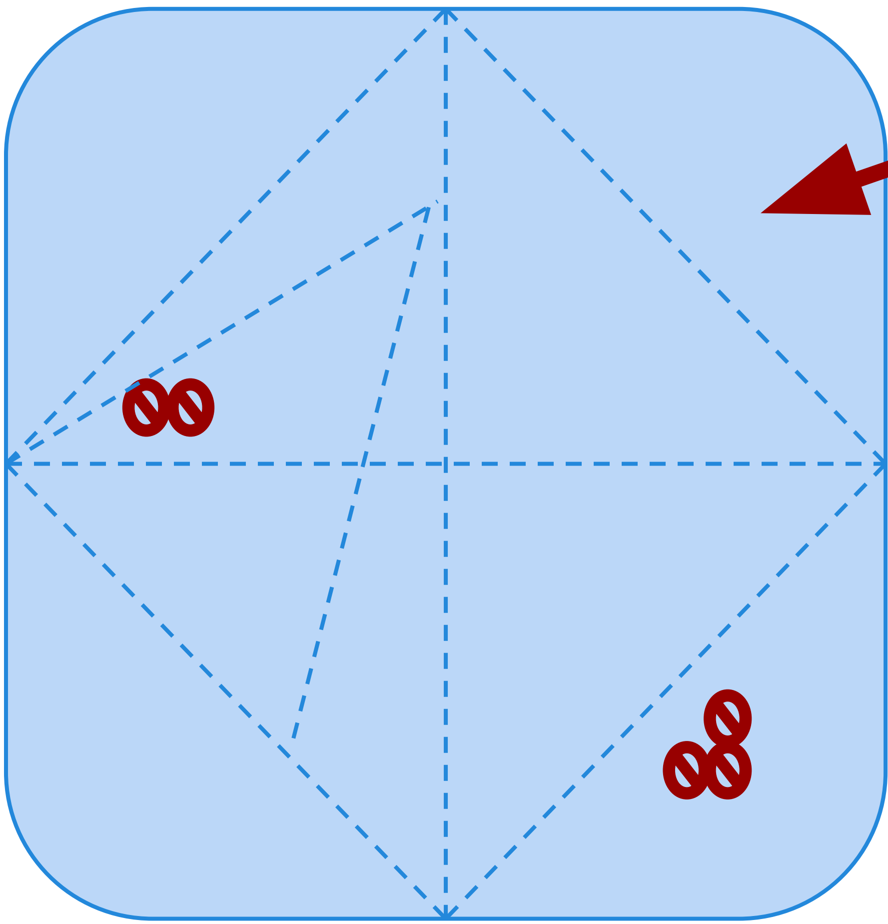
Random Testing

- Pick inputs uniformly from the distribution of all inputs.
- All inputs considered equal.
- Keep trying until you run out of time.
- No designer bias.
- Removes manual tedium.



Why Not Random?

Input Partitioning



Faults are sparse in the space of all inputs, but dense in some parts of the space where they appear.

Program

By systematically trying input from each partition, we will hit the dense fault space.

Equivalence Class

- We want to divide the input domain into *equivalence classes*.
  - Inputs from a group can be treated as the same thing (trigger the same outcome, result in the same behavior, etc.).
  - If one test reveals a fault, others in this class (probably) will too. In one test does not reveal a fault, the other ones (probably) will not either.
- Perfect partitioning is difficult, so grouping based largely on intuition, experience, and common sense.

Example

`substr(string str, int index)`

What are some possible partitions?

- index < 0
- index = 0
- index > 0
- str with length < index
- str with length = index
- str with length > index
- ...

Choosing Input Partitions

- Look for equivalent output events.
- Look for ranges of numbers or values.
- Look for membership in a logical group.
- Look for time-dependent equivalence classes.
- Look for equivalent operating environments.
- Look at the data structures involved.
- Remember invalid inputs and boundary conditions.

Look for Equivalent Outcomes

- It is often easier to find good tests by looking at the outputs and working backwards.
  - Look at the outcomes of a feature and group input by the outcomes they trigger.
- Example: getEmployeeStatus(employee ID)
  - Manager
  - Developer
  - Marketer
  - Lawyer
  - Employee Does Not Exist
  - Malformed Employee ID

Look for Ranges of Values

- If an input is intended to be a 5-digit integer between 10000 and 99999, you want partitions:

  <10000, 10000-99999, >100000
- Other options: < 0, max int, real-valued numbers
- You may want to consider non-numeric values as a special partition.

Look for Membership in a Group

Consider the following inputs to a program:

- The name of a valid Java data type.
- A letter of the alphabet.
- A country name.

- All make up input partitions.
- All groups can be subdivided further.
- Look for context that an input is used in.

Timing Partitions

The timing and duration of an input may be as important as the value of the input.

- Very hard and very crucial to get right.

- Trigger an electrical pulse 5ms before a deadline, 1ms before the deadline, exactly at the deadline, and 1ms after the deadline.
- Push the “Esc” key before, during, and after the program is writing to (or reading from) a disc.

Equivalent Operating Environments

- The environment may affect the behavior of the program. Thus, environmental factors can be partitioned and varied when testing.

- Available memory may affect the program.
- Processor speed and architecture.
  - Try with different machine specs.
- Client-Server Environment
  - No clients, some clients, many clients
  - Network latency
  - Protocols (SSH vs FTP, HTTP vs HTTPS)

Data Structure Can Suggest Partitions

Certain data structures are prone to certain types of errors. Use those to suggest equivalence classes.

For sequences, arrays, or lists:

- Sequences that have only a single value.
- Different sequences of different sizes.
- Derive tests so the first, middle, and last elements of the sequence are accessed.

Do Not Forget Invalid Inputs!

- Likely to cause problems. Do not forget to incorporate them as input partitions.
  - Exception handling is a well-known problem area.
  - People tend to think about what the program should do, not what it should protect itself against.

- Take these into account with all of the other selection criteria already discussed.

Input Partition Example

What are the input partitions for:

`max(int a, int b) returns (int c)`

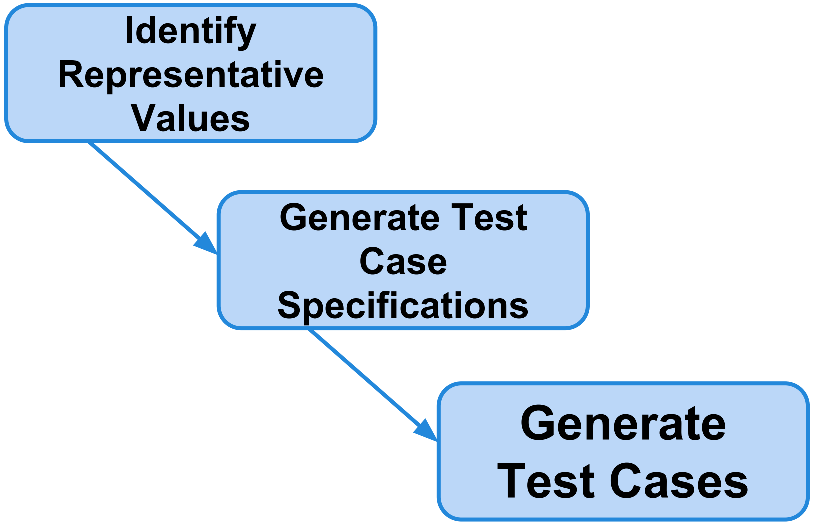
We could consider `a` or `b` in isolation:

`a < 0, a = 0, a > 0`

We should also consider the combinations of `a` and `b` that influence the outcome of `c`:

`a > b, a < b, a = b`

Creating Requirements-Based Tests



For each independently testable feature, we want to:

1. Identify the representative value partitions for each input or output.
2. Use the partitions to form abstract test specifications for the combination of inputs.
3. Then, create concrete test cases by assigning concrete values from the set of input partitions chosen for each possible test specification.

Equivalence Partitioning

Feature `insert(int N, list A)`.

Partition inputs into equivalence classes.

1. `int N` is a 5-digit integer between 10000 and 99999. Possible partitions:

   <10000, 10000-99999, >100000

2. `list A` is a list of length 1-10. Possible partitions:

   Empty List, List of Length 1, List of Length 2-10, List of Length > 10

From Partition to Test Case

Choose concrete values for each combination of input partitions: `insert(int N, list A)`

`int N`

< 10000
10000 - 99999
> 99999

`list A`

Empty List
List[1]
List[2-10]
List[>10]

Test Specifications:

```
insert(< 10000, Empty List)
insert(10000 - 99999, list[1])
insert(> 99999, list[2-10])
```

etc

Test Cases:

```
insert(5000, {})
insert(96521, {11123})
insert(150000, {11123, 98765})
```

etc

Identify Constraints Among Choices

- Test specifications are formed by combining partitions for all inputs of a feature.
- Number of possible combinations may be impractically large, so:
  - Eliminate impossible pairings.
  - Identify constraints that can remove unnecessary options.
  - From the remainder, choose a practical subset.
  - (called “category partition testing”)

Identify Constraints Among Choices

Three types of constraint:

- IF
    - This partition only needs to be considered if another property is true.
- ERROR
    - This partition should cause a problem no matter what value the other input variables have.
- SINGLE
    - Only a single test with this partition is needed.

Constraint Example - Split

`substr(string str, int index)`

Str length

length 0 [PROPERTY zeroLen]

length 1

length >= 2

Str contents

contains special characters [if !zeroLen]

contains lower case only [if !zeroLen]

contains mixed case [if !zeroLen]

Input index

value <0 [ERROR]

value = 0

value = 1

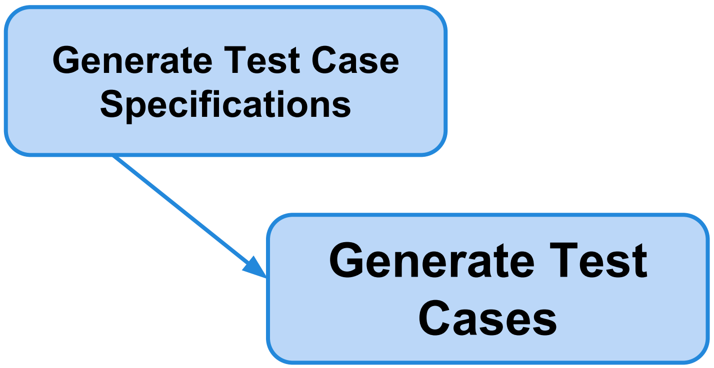
value > 1

value = MAXINT [SINGLE]

Constraints Example - Computer Customization

- **Model**
  - Model number
    - malformed **[error]**
    - not in database **[error]**
    - valid
  - Number of required slots
    - 0 **[single]**
    - 1 **[property RSNE] [single]**
    - many **[property RSNE], [property RSMANY]**
  - Number of optional slots
    - 0 **[single]**
    - 1 **[property OSNE][single]**
    - many **[property OSNE], [property OSMANY]**
- **Product Database**
  - Number of models in database
    - 0 **[error]**
    - 1 **[single]**
    - many
  - Number of components in database
    - 0 **[error]**
    - 1 **[single]**
    - many
- **Components**
  - Correspondence of selection with model slots
    - omitted slots **[error]**
    - extra slots **[error]**
    - mismatched slots **[error]**
    - complete correspondence
  - Number of required components with non-empty selections
    - 0 **[if RSNE] [error]**
    - < number required **[if RSNE] [error]**
    - = number required **[if RSMANY]**
  - Number of optional components with non-empty selections
    - 0
    - < number optional **[if OSNE]**
    - = number optional **[if OSMANY]**
  - Selected components for required (optional) slots
    - some default **[single]**
    - all valid
    - >= 1 incompatible with slot
    - >= 1 incompatible with another component
    - >= 1 not in database **[error]**

Generate Test Cases



`substr(string str, int index)`

Specification:

`str`: length >=2, contains special characters

`index`: value > 0

Test Case:

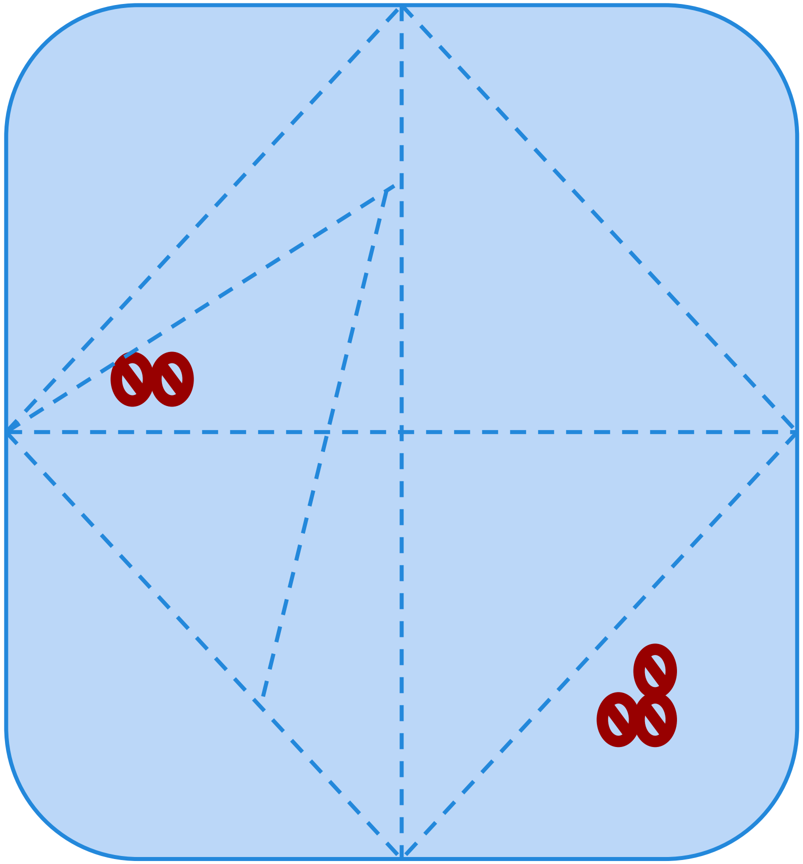
`str` = “ABCC!\n\t7”

`index`= 5

Boundary Values

Basic Idea:

- Errors tend to occur at the boundary of a partition.
- Remember to select inputs from those boundaries.

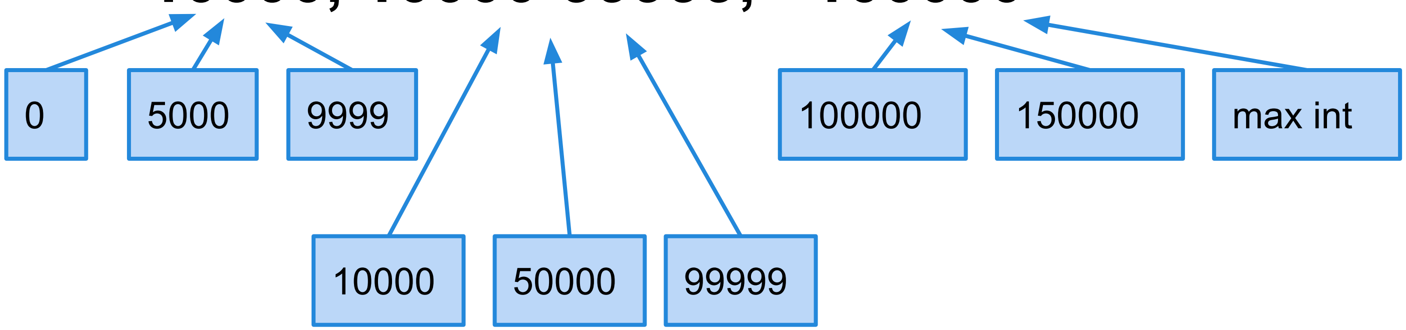


Choosing Test Case Values

Choose test case values at the boundary (and typical) values for each partition.

- If an input is intended to be a 5-digit integer between 10000 and 99999, you want partitions:

  <10000, 10000-99999, >100000



Activity: BILL Partitioning

Consider the BILL system you are designing for your homework.

1. **What are the independently testable features of BILL?**
2. **Choose one - how would you partition the input domain? Define the inputs and outputs for at least one of the independently testable features and identify partitions for each input.**

Activity: BILL Partitioning

1. **How would you partition the BILL functionality? What are the independently testable features?**

- View bill
- View transaction history
- View profile
- Edit profile
- Pay bill
- ...

Activity: BILL Partitioning

2. **How would you partition the input domain? Define the inputs and outputs and identify partitions for each input.**

View Bill

Inputs: Student ID, semester, contents of student profile (each field is an input that can be varied), profile database.

How would we partition these?

We Have Learned

- Requirements-based tests are derived by
  - identifying independently testable features
  - partitioning their input/output to identify equivalence partitions
  - combining inputs into test specifications
    - and removing impossible combinations
  - then choosing concrete test values for each specification

Next Time

- Arguing for the correctness of our specifications.
  - The World and Machine Model
- Reading:
  - Paper: “Will it Work?”
    - Available on Dropbox
- Homework
  - Due 09/27
  - Any questions?