272: Software Engineering
Fall 2018

Instructor: Tevfik Bultan

Lecture 10: Exhaustive Bounded Testing and Feedback-Directed Random Testing

Automated Testing

- Automated testing refers to the techniques which generate the test sets automatically
- We will talk about several automated testing techniques
- We will start with Korat
- Korat is also a kind of functional (black-box) testing tool
  - Requires the user to write a specification as a method in the class that is being tested
- It is used for unit testing
  - Especially for testing of complex data structure implementations
- It automatically generates test cases from specifications
  - It exhaustively generates all non-isomorphic test cases within a given scope

Korat

- An automated testing tool
  - Application domain is unit testing of complex data structures

- It uses Java predicates to generate the test cases
  - These are Java methods which return a boolean value
  - For example, pre and post-conditions of methods

- Korat generates the test cases from pre and postconditions of methods

- There is no need to write an extra specification if the class contract is written as Java predicates

Korat

- Korat uses the method precondition to automatically generate all nonisomorphic test cases up to a given small size
  - Given a predicate and a bound on the size of its inputs Korat generates all nonisomorphic inputs for which the predicate returns true
- Korat then executes the method on each test case and uses the method postcondition as a test oracle to check the correctness of output
  - Korat exhaustively explores the bounded input space of the predicate but does so efficiently by monitoring the predicate’s executions and pruning large portions of the search space

An Example: Binary Tree

```
import java.util.*;
class BinaryTree {
  private Node root;
  private int size;
  static class Node {
    private Node left;
    private Node right;
  }
  public boolean repOk() {
    // this method checks the class invariant:
    // checks that empty tree has size zero
    // checks that the tree has no cycle
    // checks that the number of nodes in the tree is
    // equal to its size
  }
```

An Example: Binary Tree

```
public boolean repOk() {
  // checks that empty tree has size zero
  if (root == null) return size == 0;
  Set visited = new HashSet();
  visited.add(root);
  LinkedList workList = new LinkedList();
  workList.add(root);
  while (!workList.isEmpty()) {
    Node current = (Node)workList.removeFirst();
    if (current.left != null) {
      // checks that tree has no cycle
      if (!visited.add(current.left)) return false;
      workList.add(current.left);
    }
    if (current.right != null) {
      // checks that tree has no cycle
      if (!visited.add(current.right)) return false;
      workList.add(current.right);
    }
  } // checks that size is consistent
  if (visited.size() != size) return false;
  return true;
}
```

Finitization

- Korat uses a finitization description to specify the finite bounds on the inputs (scope)

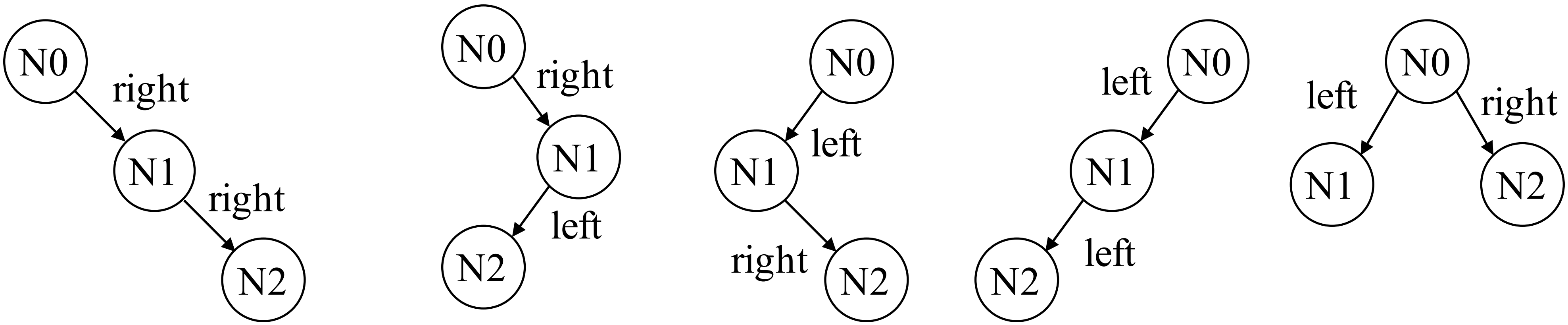
```
public static Finitization finBinaryTree(int NUM_node){
  Finitization f = new Finitization(BinaryTree.class);
  ObjSet nodes = f.createObjects("Node", NUM_node);
  nodes.add(null);
  f.set("root", nodes);
  f.set("size", NUM_node);
  f.set("Node.left", nodes);
  f.set("Node.right", nodes);
  return f;
}
```

Creates a set of objects of Type "Node" with NUM_node objects in the set

The value of size is set to NUM_node

- This finitization if for binary trees with exactly NUM_node nodes
- Korat automatically generates a finitization skeleton based on the type declarations in the Java code
  - Developers can restrict or extend this default finitization

Non-isomorphic Instances for finBinaryTree(3)



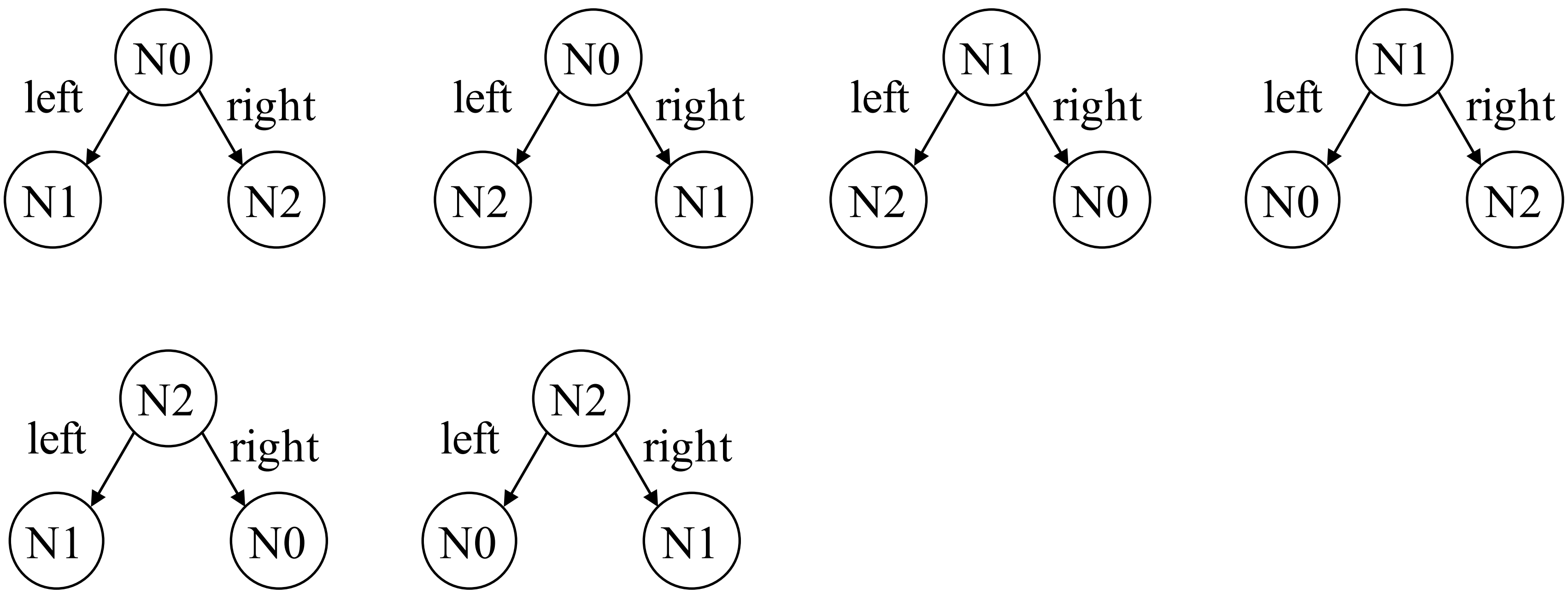
Korat automatically generates non-isomorphic instances within a given bound

For finBinaryTree(3) Korat generates the 5 non-isomorphic trees shown above.

Each of the above trees correspond to 6 isomorphic trees. Korat only generates one tree representing the 6 isomorphic trees.

For finBinaryTree(7) Korat generates 429 non-isomorphic trees in less than a second

Isomorphic Instances



How many Instances are There?

- How many instances are there? What is the size of the state space?
- Consider the binary tree example with scope 3
  - There are three fields: root, left, right
  - Each of these fields can be assigned a node instance or null
  - There is one root field and there is one left and one right field for each node instance

- We can consider each test case for the binary tree with scope 3 a vector with 8 values
  - The value of the root (has 4 possible values, null or a node object)
  - The value of the size (has only one possible value, which is 3)
  - For each node object (there are three of them)
    - The value of the left field (4 possible values, null or a node object)
    - The value of the right field (4 possible values, null or a node object)
- State space : $4 \times 1 \times (4 \times 4)^3$

How many Instances are There?

- Given n node instances, the state space (the set of all possible test cases) for the binary tree example is:

  $$(n+1)^{2n + 1}$$

- Most of these structures are not valid binary trees
  - They do not satisfy the class invariant
- Most of these structures are isomorphic
  - they are equivalent if we ignore the object identities

Generating test cases

- The challenge is generating all the non-isomorphic test cases that satisfy the class invariant

1. Korat only generates non-isomorphic test cases

2. Korat prunes the state space by eliminating sets of test cases which do not satisfy the class invariant

Isomorphism

- In Korat isomorphism is defined with respect to a root object
  - for example `this`

- Two test cases are defined to be isomorphic if the parts of their object graphs reachable from the root object are isomorphic

- The isomorphism definition partitions the state space (i.e., the input domain) to a set of isomorphism partitions
  - Korat generates only one test case for each partition class

Isomorphism

- The isomorphism definition used in Korat is the following
  - $O_1, O_2, ..., O_n$ are sets objects from n classes
  - $O = O_1 \cup O_2 \cup ... \cup O_n$
  - P: the set consisting of null and all values of primitive types that the fields of objects in O can contain
  - $r \in O$ is a special root object
  - Given a test case C, O_C is the set of objects reachable from r in C
- Two test cases C and C' are *isomorphic* iff there is a permutation π on O, mapping objects from O_i to objects from O_i for all $1 \leq i \leq n$, such that

$$\forall o, o' \in O_C \,.\, \forall f \in \text{fields}(o) \,.\, \forall p \in P \,.$$

$$o.f == o' \text{ in } C \text{ iff } \pi(o).f == \pi(o') \text{ in } C' \text{ and}$$

$$o.f == p \text{ in } C \text{ iff } \pi(o).f == p \text{ in } C'$$

Generating Test Cases

- Korat only generates the test cases which satisfies the input predicate: class invariant and the precondition

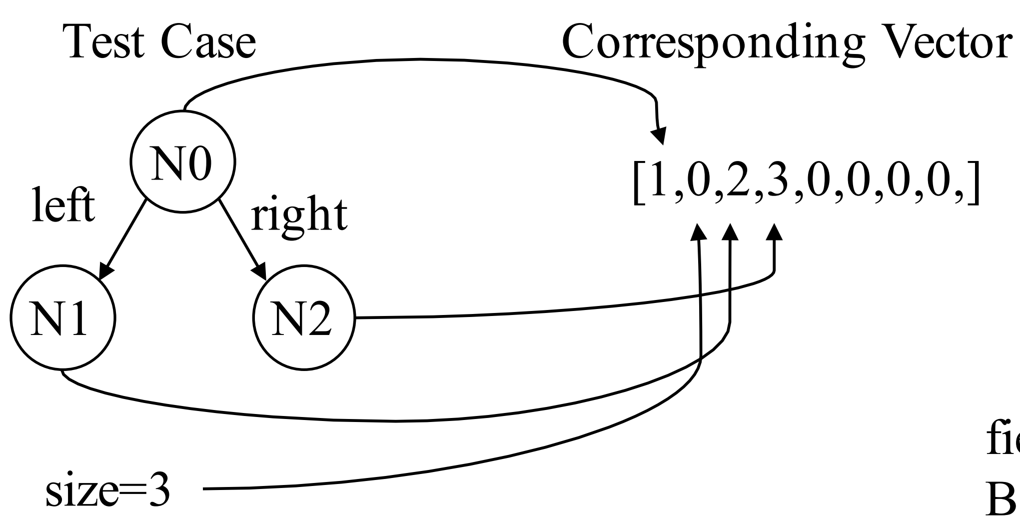
- Korat explores the state space efficiently using backtracking
  - It does not generate all instances one by one and check the input predicate
  - It prunes its search of the state space based on the evaluation of the input predicate
    - If the method that checks the input predicate returns false without checking a field then there is no need to generate test cases which assign different values to that field
      - ***In order to exploit this, Korat keeps track of the fields that are accessed before the predicate returns false***
  - For this to work well, predicate method should return false as soon as it detects a violation

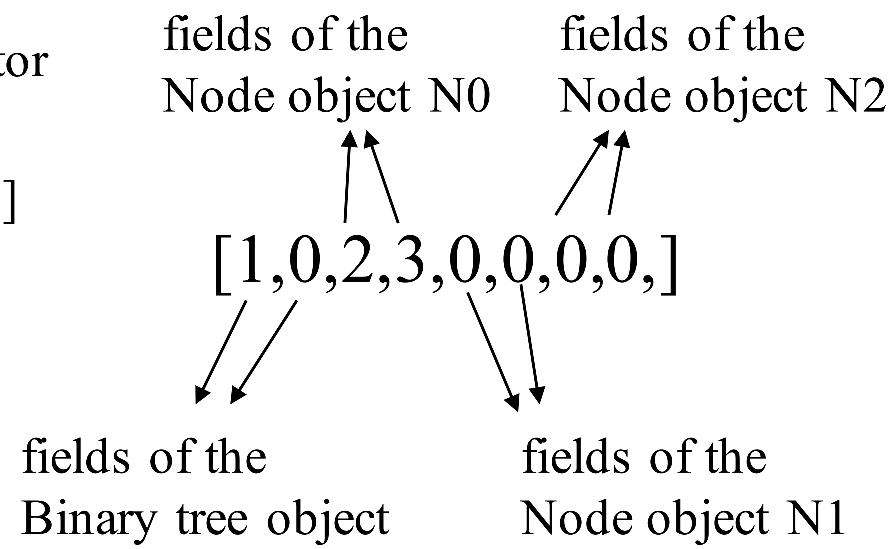
Generating Test Cases

- Korat orders all the elements in every class domain and every field domain
- Each test case is represented as a vector of indices into the corresponding field domains

For the Binary Tree example assume that:

- The class domain is ordered as N0 < N1 < N2
- The field domains for root, left and right are ordered as null < N0 < N1 < N2 (with indices 0, 1, 2, 3)
- The size domain has one element which is 3 (with index 0)





Generating Test Cases

- Search starts with a candidate vector set to all zeros.
- For each candidate vector, Korat sets fields in the objects according to the values in the vector.
- Korat then invokes repOk (i.e., class invariant) to check the validity of the current candidate.
  - During the execution of repOk, Korat monitors the fields that repOK accesses, it stores the indices of the fields that are accessed by the repOK (field ordering)
  - For example, for the binary tree example, if the repOK accesses root, N0.left and N0.right, then the field ordering is 0, 2, 3
- Korat generates the next candidate vector by backtracking on the fields accessed by repOk.
  - First increments the field domain index for the field that is last in the field-ordering
  - If the field index exceeds the domain size, then Korat resets that index to zero and increments the domain index of the previous field in the field ordering

Generating Test Cases

- Korat achieves non-isomorphic test case generation using the ordering of field domains and the vector representation
- While generating the test cases, Korat ensures that the indices of the objects that belong to the same class domain are listed in nondecreasing order in the generated candidate vectors
- This means that during backtracking, Korat looks for fields
  - that precede the field that is accessed last and
  - that have an object from the same class domain as the field that is accessed last
  - and makes sure that the object assigned to the field that is accessed last is higher in the ordering than those objects

Using Contracts in Testing

- Korat checks the contracts written in JML on the generated instances

```
 //@ public invariant repOk();     //class invariant

/*@ requires has(n)                // precondition
  @ ensures !has(n)                // postcondition
  @*/
public void remove(Node n) {
 ...
}
```

- JML (Java Modeling Language) is an annotation language for writing contracts (pre, post-conditions for methods, class invariants) for Java classes in the style of Design-by-Contract
- Korat uses JML tool-set to translate JML annotations into runtime Java assertions

Using Contracts in Testing

- Given a finitization, Korat generates all non-isomorphic test cases within the given scope (defined by the finitization) that satisfy the class invariant and the pre-condition

- The post-conditions and class invariants provide a test oracle for testing
  - For each generated test case, execute the method that is being tested and check the class invariant and the post-condition

- Korat uses JML tool-set to automatically generate test oracles from method post-conditions written as annotations in JML

Korat Performance

- Checking a BinaryTree implementation
  - with scope 8 takes 1/53 seconds,
  - with scope 11 takes 56.21 seconds,
  - with scope 12 takes 233.59 seconds.

- Test case generation with Korat is more efficient than a previous approach that used SAT solvers for generating test cases

Small Scope Hypothesis

- Success of Korat depends on the small scope hypothesis

- Small scope hypothesis
  - If there is a bug, there is typically a counter-example demonstrating the bug within a small scope.

- This is not a bad assumption if you are analyzing data structures (the target for Korat)

Another Approach: Random Testing

- We discussed that random testing is hopeless is one uses a uniform distribution for inputs
  – The chances of covering a particular condition is very low
- However, if we use the program being tested to guide how we choose the test cases, then random test generation can be more effective
  – Construct the inputs using the methods that are part of the program that is being tested
- This is the approach used in *feedback-directed random test generation*

Feedback-Directed Random Unit Testing

- This is a unit testing approach where it is assumed that the unit under test consists of a set of methods
  - A class, or a set of classes in an object oriented program for example

- The basic idea is to build inputs incrementally by randomly selecting a method call to apply and finding arguments from among previously constructed inputs
  - The result (object) returned by the newly generated call is first checked against
    - Contracts (to look for contract violations)
    - Filters (to eliminate duplicate or uninteresting cases)
  - Then it is added to the available set of inputs to be used as an input in future method calls

Randoop

- Randoop is a fully automated testing tool that implements the feedback-directed random test generation for .NET and Java
- An object-oriented unit test consists of a sequence of method calls that set up the state (by creating and mutating objects) and an assertion about the result of the final call
  - Randoop generates such unit tests
- Randoop found serious errors in widely-deployed commercial open-source software

Method Sequences

- A method sequence is a sequence of method calls:

```
A a1 = new A();
B b3 = a1.m1(a1);
```

- Each call in the sequence includes a method name and input arguments, which can be
  - Primitive values such as 0, true, null, or
  - Objects returned by previous calls
  - The receiver of the method call is treated as the first argument of the method call

- A method sequence can be written as code and executed

Extending sequences

- Randoop uses an incremental approach to test generation
  - It generates method sequences by extending sequences by one method call at a time
- It uses an extension operation that takes zero or more sequences as input and generates a new sequence

  extend(m, seqs, vals)

  - m is a method with formal parameters (including the receiver) of type T_1, T_2, …, T_k
  - seqs is a list of sequences (possibly empty)
  - vals is a list of values v_1: T_1, v_2:T_2, … v_k:T_k
    - Each value is a primitive value or it is the return value s.i of the i-th method call for a sequence s appearing in seqs
- The result of extend(m, seqs, vals) is a new sequence that is the concatenation of the input sequences seqs in the order that they appeared, followed by the method call m(v_1, v_2, …, v_k)

Generating sequences

- GenerateSequences algorithm takes a set of classes, contracts, filters and timeLimit as input
- It starts from an empty set of sequences
  - Builds sequences incrementally by extending previous sequences
- As soon as a sequence is built, it executes it to ensure that it creates a non-redundant and legal objects, as specified by filters and contracts

Algorithm for generating method sequences

```
GenerateSequences(classes, contracts, filters, timeLimit)
errorSeqs := {}
nonErrorSeqs := {}
while timeLimit not reached do
   m(T1,T2, …, Tk) := randomPublicMethod(classes)
   <seqs, vals> := randomSeqsAndVals(nonErrorSeqs, T1,T2, …, Tk)
   newSeq := extend(m, seqs, vals)
   if newSeq ∈ nonErrorSeqs ∪ errorSeqs then
      continue
   endif
   <o, violated> := execute(newSeq, contracts)
   if violated = true then
      errorSeqs := errorSeqs ∪ {newSeq}
   else
      nonErrorSeqs : = nonErrorSeqs ∪ {newSeq}
      setExtensibleFlags(newSeq, filters, o)
   endif
endwhile
return <nonErrorSeqs, errorSeqs>
```

Random selection of sequences and values

- The function randomSeqsAndVals(nonErrorSeqs, T_1,T_2, …, T_k) incrementally builds a list of sequences seqs and a list of values vals
- At each step it adds a value to vals and potentially also a sequence to seqs
- For each input argument type T_i it does the following
  - If it is a primitive type, pick a value from a fixed pool of values
    - In the implementation they use a small primitive pool that can be augmented by the user or other tools
  - If it is an object, they pick one of the following three possibilities randomly:
    1. Use a value v from a sequences that is already in seqs
    2. Add a sequence from nonErrorSeqs to seqs and use a value from it
    3. Use null
  - When using a value produced by an existing sequence, the value must be extensible (v.extensible = true)

Tracking contract violations

- execute(newSeq, contracts) executes each method call in the sequence and checks the contracts after each call.
- Contracts specify the invariants that hold before and after each call.
- The output of execute(newSeq, contracts) is a tuple <**o**, violated> that consists of
  - runtime values created during the execution of the sequence and
  - the boolean flag violated that is set to true if at least one contract was violated during execution
- A sequence that causes a contract violation is added to the errorSeqs
- If the sequence does not lead to a contract violation then it is added to the nonErrorSeqs

Filtering to identify extensible values

- Filtering determines which values are extensible (i.e., can be used to extend a sequence after they are created)
- Each sequence has an associated boolean vector:
  - Each value s.i (the value returned by the ith method call in the sequence) has a boolean flag s.i.extensible
- s.i.extensible indicates whether the given value may be used as an input to a new method call
  - This is used to prune the state space
- Extensible flag is set to false if the value is considered redundant or illegal for the purpose of creating a new sequence
  - Assume that the s.i corresponds to the object o, then s.i.extensible is set to false if there exists another extensible object o' that was created earlier and o.equals(o') returns true
  - If the s.i is null, then s.i.extensible is set to false
  - If s.i throws an exception, then s.i.extensible is set to false since exceptions typically correspond to a violation

Evaluation

- They compared Randoop with an exhaustive bounded testing approach that uses JPF (Java Path Finder)
  - They focused on container classes (BinTree, Bheap, FibHeap)
  - Randoop achieved better coverage in shorter amount of time

- They used Randoop to check API contracts in 14 widely-used libraries (java.util, javax.xml, Jakarta Commons, .NET Framework)
  - JPF based exhaustive bounded testing does not scale to this type of testing
  - Randoop produced 4200 distinct violation inducing test cases
  - They developed a reduction technique to automatically reduce the reported violations (if two test cases cause the same violation, only report one)
  - After the reduction, they obtained 254 error-revealing test cases that pointed to 210 distinct errors

Errors

Some discovered errors:

- Eight of the methods in the JDK create collections that return false on s.equals(s)
- In Jakarta
  - a hashCode implementation fails to handle a valid object configuration where a field is null
  - An iterator object throws a NullPointerException if initialized with zero elements
- In .NET libraries
  - 155 errors are NullReferenceExceptions in the absence of null inputs
  - 21 are IndexOutOfRangeExceptions
  - 20 are violations of equals, hashCode and toString contracts