FACULDADE ENGENHARIA
Departamento de Informática

Plataformas e Serviços X-Ops (16233)

Automated Test Case Generation

adapted from lecture notes of the “DIT 635 - Software Quality and Testing” unit,
delivered by Professor Gregory Gay, at the Chalmers and the University of Gothenburg, 2022)

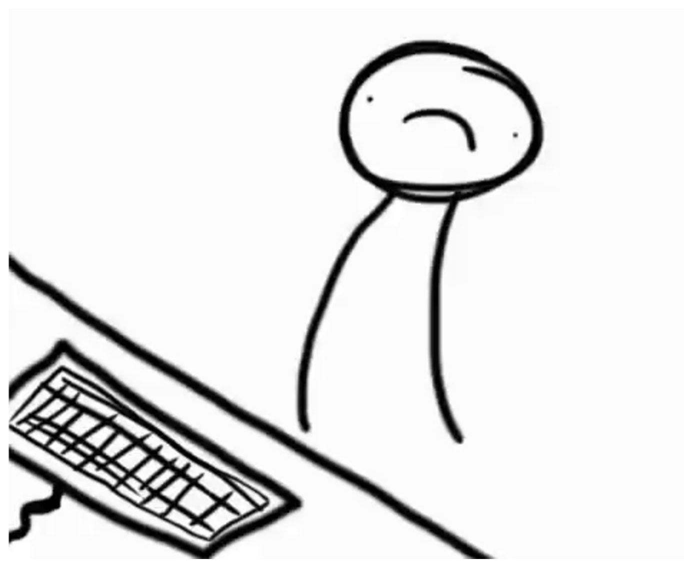
Today's Goals

- ✧ Introduce Search-Based Test Generation
  - (a.k.a. : Fuzzing)
  - Test Creation as a Search Problem
  - Metaheuristic Search
  - Fitness Functions
- ✧ Example - Generating Covering Arrays for Combinatorial Interaction Testing

Automating Test Creation

✧ Testing is invaluable, but expensive.

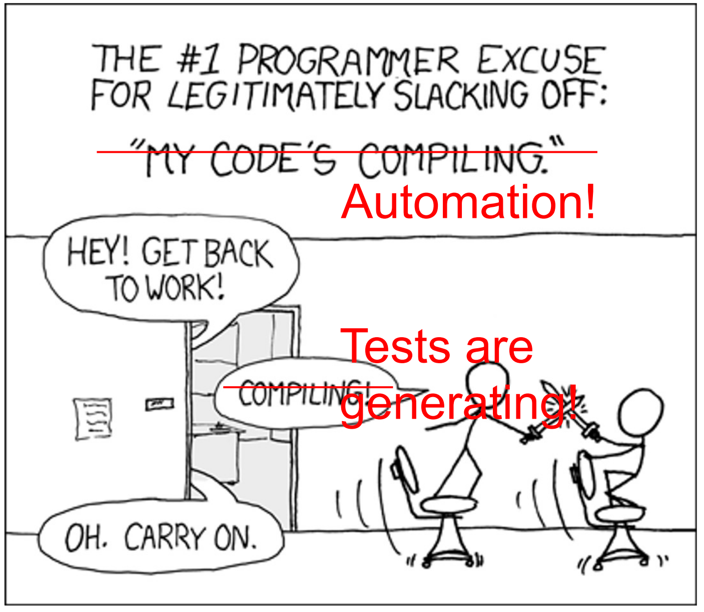
- We test for ***many*** purposes.
- Near-infinite number of possible tests we could try.
- Hard to achieve meaningful volume.



Automation of Test Creation

✧ Relieve cost by automating test creation.

- Repetitive tasks that do not **need** human attention.
- **Generate test input**.
  - Need to add assertions.
  - Or just look for crashes.



Test Automation

- ✧ **Test Automation** is the development of software to separate repetitive tasks from the creative aspects of testing.
- ✧ Automation allows control over *how* and *when* tests are executed.
  - Control the environment and preconditions.
  - Automatic comparison of predicted and actual output.
  - Automatic hands-free re-execution of tests.

Manual vs Automation

✧ Scaling

- Manual generation can be an exhaustive and a time-consuming process. It scales with the size of the Project which can hinder the development speed of the software;
- Automated generation, being an automated process, can help reduce the time needed to perform testing activities.

✧ Coverage and Mutation

- Automated generation of unit tests usually provides a higher capability of achieving better coverage values than the manual approach.
- The ability to identify mutants in unit tests (identification of allocated defects) is generally better in unit tests generated automatically.

Test Creation as a Search Problem

- Do you have a **goal** in mind when testing?
  - *Make the program crash, achieve code coverage, cover all 2-way interactions, …*
- You are **searching** for a test suite that achieves that goal.
  - Algorithm samples possible test input to find those tests.

Test Creation as a Search Problem

- ✧ "I want to find all faults" cannot be measured.
- ✧ *However, a lot of testing goals can be.*
    - Check whether properties satisfied (boolean)
    - Measure code coverage (%)
    - Count the number of crashes or exceptions thrown (#)
- ✧ If goal can be measured, search can be automated.

Search-Based Test Generation

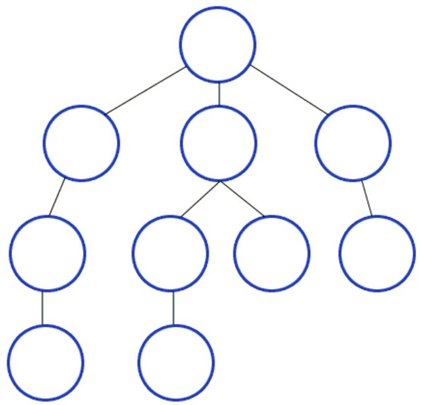
- **Make one or more guesses.**
  - Generate one or more individual test cases or full suites.
- **Check whether goal is met.**
  - Score each guess.
- **Try until time runs out.**
  - Alter the population based on strategy and try again!

Search Strategy

- ✧ The order that solutions are tried is the key to efficiently finding a solution.
- ✧ A search follows some defined strategy.
  - Called a “**heuristic**”.
- ✧ Heuristics are used to choose solutions and to ignore solutions known to be unviable.
  - Smarter than pure random guessing!

Heuristics - Graph Search

- ✧ Arrange nodes into a hierarchy.
  - Breadth-first search looks at all nodes on the same level.
  - Depth-first search drops down hierarchy until backtracking must occur.



- ✧ Attempt to estimate shortest path.
  - A* search examines distance traveled and estimates optimal next step.
  - Requires domain-specific scoring function.

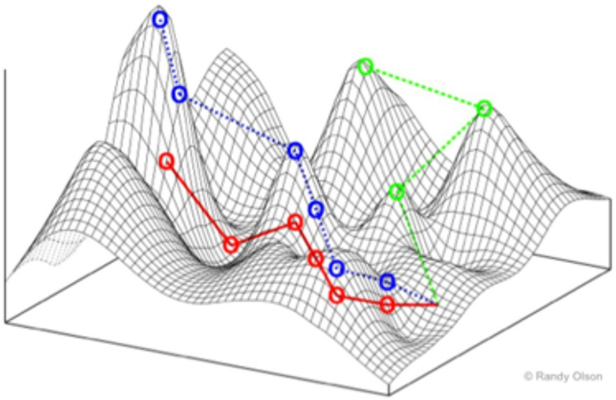
How Long Do We Spend Searching?

- Exhaustive search not viable.
- Search can be bound by a **search budget**.
  - Number of guesses.
  - Time allotted to the search (number of minutes/seconds).
- **Optimization problem**:
  - *Best solution possible before running out of budget.*

Generation as Optimization Problem

✧ Search heuristic becomes important.

- If time bound: time to create, execute, and evaluate.
- If attempt bound: strategy used to choose next solution.
  - Ignoring bad solutions, learning what makes a solution good.
- In practice, **efficiency in both categories is desired**.

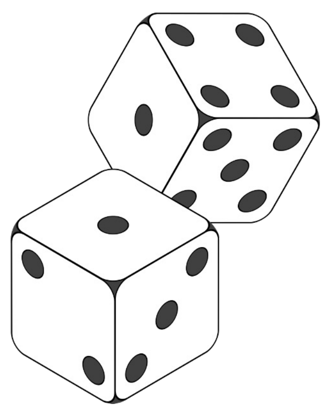


Random Search

✧ Randomly formulate a solution.

- Unit testing: choose a class in the system, choose random methods, call with random parameter values.
- System-level testing: choose an interface, choose random functions from interface, call with random values.

✧ Keep trying until goal attained or budget expires.



Random Search

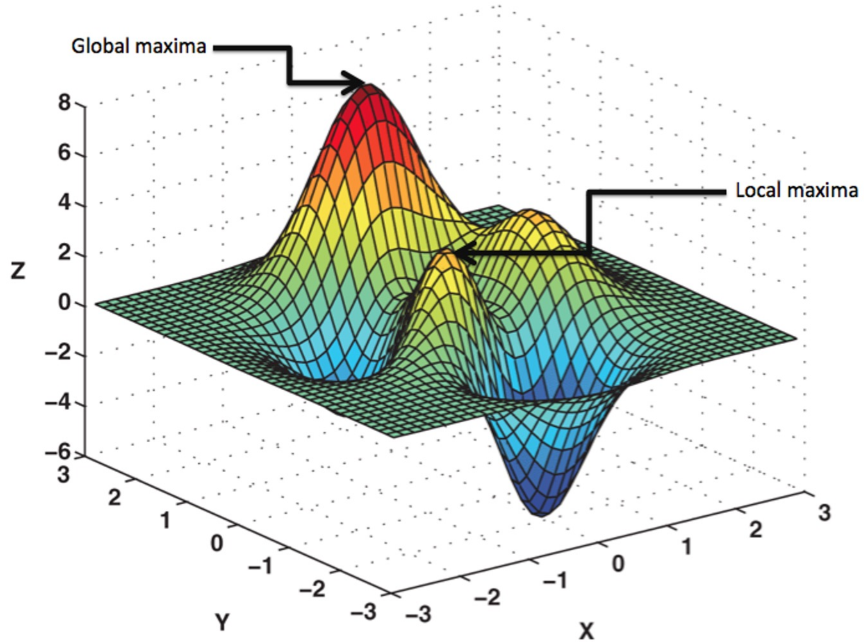
✧ Sometime viable:

- Extremely fast.
- Easy to implement, easy to understand.
- All inputs considered equal, so no designer bias.

✧ However…

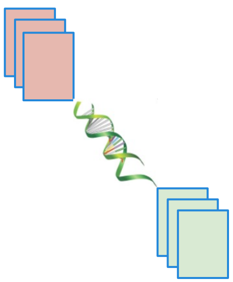
Metaheuristic Search

- ✧ Random search is naive.
  - Only possible to cover a small % of full input space.
- ✧ Metaheuristic search adds intelligence to random.
  - Feedback and sampling strategies.
  - Still fast, able to learn from bad guesses.

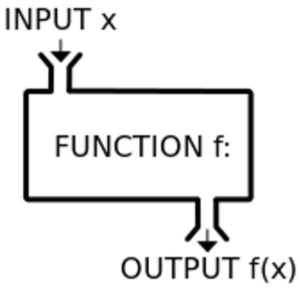


Mechanics of Optimization

AKA: How can I get a computer to search?

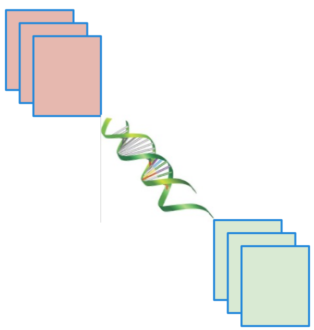


Metaheuristic

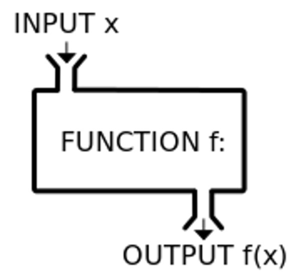


Fitness Function(s)

Search-Based Test Generation



+



=

The Metaheuristic (Sampling Strategy)

Genetic Algorithm
Simulated Annealing
Hill Climber
(...)

The Fitness Functions (Feedback Strategies)

Distance to Coverage Goals
Count of Executions Thrown
Input or Output Diversity
(...)

(Goals)

Cause Crashes
Cover Code Structure,
Generate Covering Array,
(...)

The Metaheuristic

- Decides how to select and revise solutions.
  - Changes approach based on past guesses.
  - Fitness functions give feedback.
  - Population mechanisms choose new solutions and determine how solutions evolve.

The Metaheuristic

- ✧ Decides how to select and revise solutions.
  - Small adjustments **(local search)** or sampling from the whole space **(global search)**.
  - One solution at a time or entire populations.
  - Often based on natural phenomena (swarm behavior, evolution).
  - Trade-off between speed, complexity, and understandability.

"Solutions"

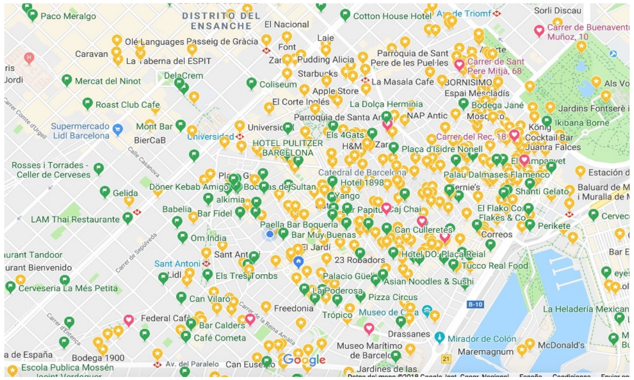
- ✧ What is a solution?
  - **Test Case:** Evolved in isolation from other test cases.
  - **Test Suite:** A set of test cases, evolved together.
- ✧ Depends on how goal attainment measured.
  - Code Coverage
    - Test Case: Target one code section at a time.
    - Test Suite: Target coverage of entire class/system.

Local Search

- ✧ Generate and score a potential solution.
- ✧ Attempt to improve by looking at its **neighborhood**.
  - Make small, incremental improvements.
- ✧ Very fast, efficient if good initial guess.
  - Get “stuck” if bad guess.
  - Often include reset strategies.

Exploring the Neighborhood

- ✧ Small changes to solution.
  - For each call:
    - Switch value of boolean, other values from an enumerated set, bounded range of numeric choices.
  - Full test case:
    - Insert a new call.
    - Delete or replace an existing call.
      - Can replace by changing the function called or its parameters.

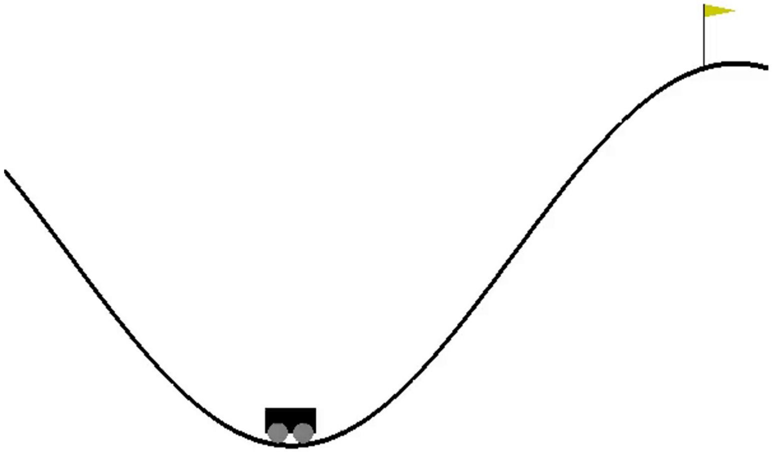


Hill Climbing

- ✧ Pick a initial solution at random.
- ✧ Examine the local neighborhood.
- ✧ Choose the best neighbor and “move” to it.
- ✧ Repeat until no better solution can be found.
  - Climbs mountains in fitness function landscape.
  - Restart when no improvement can be found.

Hill Climbing Strategies

- ✧ **Steepest Ascent**
  - Examine all neighbors
  - Pick one with highest improvement.
- ✧ **Random Ascent**
  - Examine random neighbors.
  - Choose first to show *any* improvement.

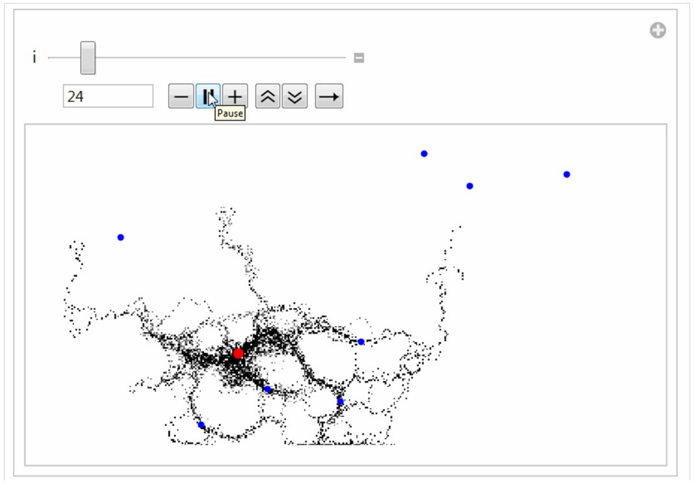


Simulated Annealing

- Choose a neighboring test case.
  - If better, select it. If not, select it at probability:
    $prob(score, newScore, time, temp) = e^{((score - newScore) * (time / temp))}$
  - Governed by temperature function:
    $temp(time, maxTime) = (maxTime - time) / maxTime$
- Initially, large jumps around search space.
  - Stabilizes over time.

Global Search

- ✧ Generate multiple solutions.
- ✧ Evolve by examining whole search space.
- ✧ Typically based on natural processes.
    - Swarm patterns, foraging behavior, evolution.
    - Models of how populations interact and change.



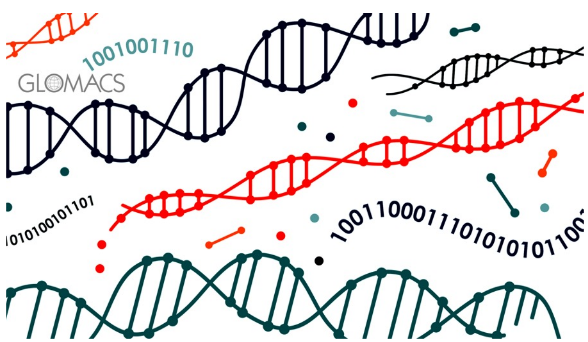
Genetic Algorithms

✧ Over multiple generations, evolve a population.

- Good solutions persist and reproduce.
- Bad solutions are filtered out.

✧ Diversity is introduced by:

- Keeping the best solutions.
- Some random solutions.
- Creating “offspring” through **mutation** and **crossover**.

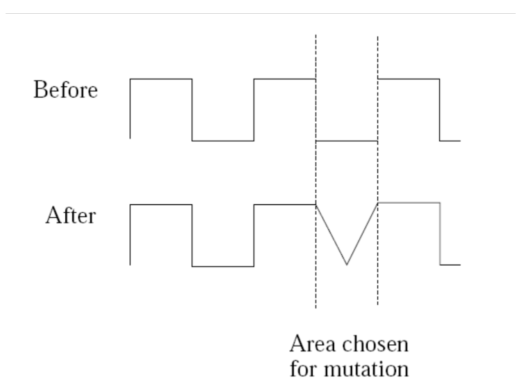


Genetic Algorithms - Mutation

✧ Copy a high-scoring solution.

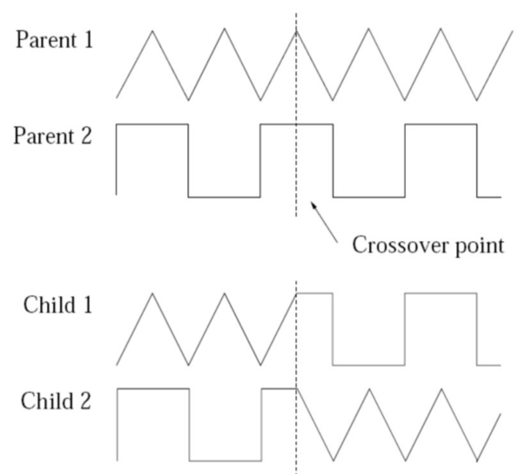
✧ Impose a small change.

- (add/delete/modify a function call, change an input value)
- Follow the rules for determining the neighbors of a test.
- Choose a neighbor from that set.



Genetic Algorithms - Crossover

- ✧ By “breeding” two good tests, we may produce better tests.
- ✧ Form two new solutions.
  - Sample from probability distribution to decide which parent to inherit from.



Genetic Algorithms - Crossover

- One Point Crossover
  - Splice at crossover point.

A	B	C	D		A	B	3	4
1	2	3	4		1	2	C	D

- Uniform Crossover
  - Flip coin at each line, second child gets other option.

A	B	C	D		A	2	3	D
1	2	3	4		1	B	C	4

- Discrete Recombination
  - Flip coin at each line for both children.

A	B	C	D		A	2	C	4
1	2	3	4		A	B	3	4

Particle Swarm Optimization

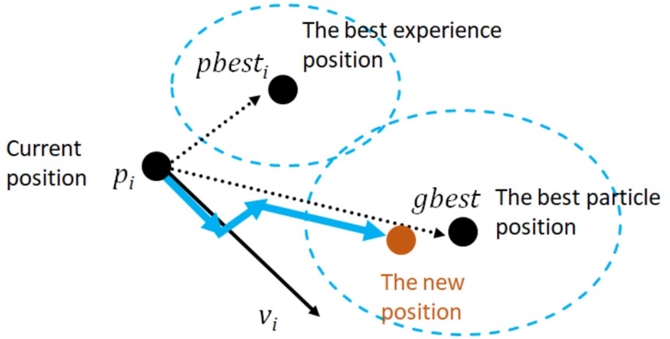
- ✧ A swarm of agents each attempt to search for good test cases.
- ✧ When another agent finds a better solution than the best known "worldwide", they tell everybody.
- ✧ Each agent mutates their solution based on their knowledge of the best local solution and the best global solution.
- ✧ Over time, the agents converge on the best solutions.

Particle Swarm Optimization

✧ Each agent has velocity and position.

- *Position*: Their current solution.
- *Velocity*: The amount of change to be made to the solution. Bound by a maximum velocity.
  - *Vectors* along all dimensions in the solution. (i.e., method parameters).

✧ Each round, velocity and position are updated based on current local and global knowledge.

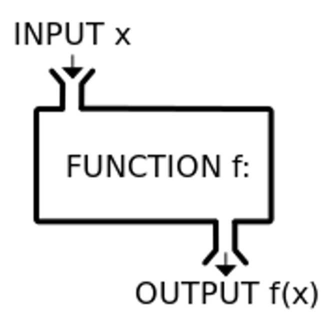


Fitness Functions

- ✧ Fitness functions play a crucial role in search-based test generation.
- ✧ Fitness functions must adhere to the following requirements:
  - Return **continuous scores** as to offer better feedback for the metaheuristic algorithms.
  - Return **only numeric values** in order to properly evaluate the generation of test cases each time.
  - Indication of how close the generation was to being optimal. It should not indicate quality but a distance to optimal quality.

Fitness Functions

- ✧ Domain-based scoring functions that determine how good a potential solution is.
  - Should offer feedback:
    - Percentage of goal attained.
    - *Better - information on how to improve solution.*
  - **Can optimize more than one at once.**
    - Independently optimize functions
    - Combine into single score.



Example - Branch Coverage

- ✧ **Goal:** Attain Branch Coverage over the code.
  - Tests reach branching point (i.e., if-statement) and execute all possible outcomes.
- ✧ **Fitness function (Attempt 1):**
  - Measure coverage and try to maximize % covered.
  - **Good:** Measurable indicator of progress.
  - **Bad:** No information on how to improve coverage.

Example - Branch Coverage

- ✧ Attempt 2: Distance-Based Function
- ✧ **fitness = branch distance + approach level**
  - **Approach level**
    - Number of branching points we need to execute to get to the target branching point.
  - **Branch distance**
    - If other outcome is taken, how “close” was the target outcome?
    - How much do we need to change program values to get the outcome we wanted?

Example - Branch Coverage

```
if(x < 10){ // Branch 1
    // Do something.
}else if (x == 10){ // Branch 2
    // Do something else.
}
```

Goal: Branch 2, True Outcome

Approach Level

- If Branch 1 is true, approach level = 1
- If Branch 1 is false, approach level = 0

Branch Distance

- If x==10 evaluates to false, branch distance = (abs(x-10)+k).
- Closer x is to 10, closer the branch distance.

Other Common Fitness Functions

- Number of methods called by test suite
- Number of crashes or exceptions thrown
- Diversity of input or output
- Detection of planted faults
- Amount of energy consumed
- Amount of data downloaded/uploaded
- **… (anything that reflects what a *good* test is)**

What Do I Do With These Inputs?

- ✧ If looking for crashes, just run generated input.
- ✧ If you need to judge correctness, add assertions.
  - General properties, not specific output.
    - **No**: `assertEquals(output, 2)`
    - **Yes:** `assertTrue(output % 2 == 0)`

Automated Program Repair

- ✧ Produce patches for common bug types.
- ✧ Many bugs can be fixed with just a few changes to the source code - inserting new code, and deleting or moving existing code.
  - Add null values check.
  - Change conditional expression.
  - Move a line within a try-catch block.

Generate and Validate

- ✧ **Genetic programming** - solutions represent sequences of edits to the source code.
- ✧ **Generate and validate approach**:
  - Fitness function: how many tests pass?
  - Patches that pass more tests create new population:
    - Mutation: Change one edit into another.
    - Crossover: Merge edits from two parent patches.

Risks of Automation

✧ Structural coverage is important.

- Unless we execute a statement, we’re unlikely to detect a fault in that statement.

✧ More important: how we execute the code.

- Humans incorporate context from a project.
- “Context” is difficult for automation to derive.
- One-size-fits-all approaches.

Limitations of Automation

- ✧ Automation produces different tests than humans.
  - "shortest-path" approach to attaining coverage.
  - Apply input different from what humans would try.
  - Execute sequences of calls that a human might not try.
- ✧ Automation **can be** very effective, but more work is needed to improve it.


HOW
WHY
HOW
WHEN
WHERE
WHERE
WHERE
WHY
WHO
WHO
WHEN
WHAT
WHAT
WHY
WHAT
WHEN
HOW
WHAT
WHAT
WHEN
WHO
WHERE
WHY
HOW