



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY



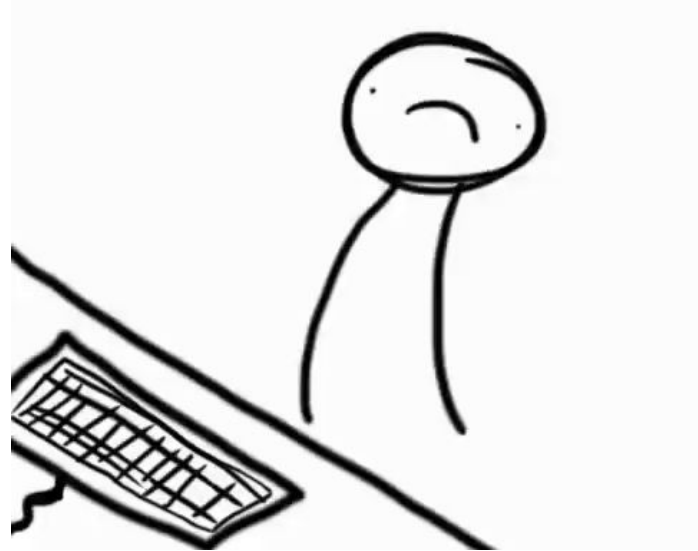
UNIVERSITY OF GOTHENBURG

# Lecture 15: Automated Test Case Generation

Gregory Gay  
DIT636/DAT560 - March 4, 2024

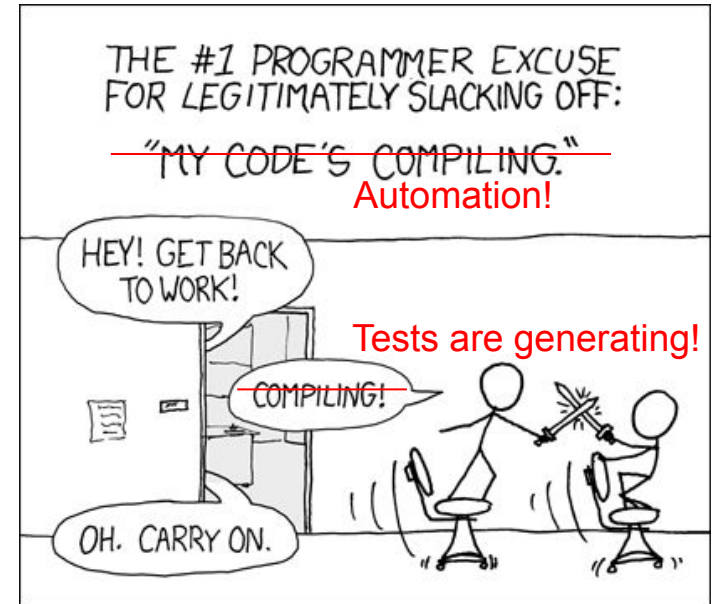
# Automating Test Creation

- 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.**
    - Just need to add assertions.
    - (Or check for crashes, memory leaks, other problems that can be measured automatically.)

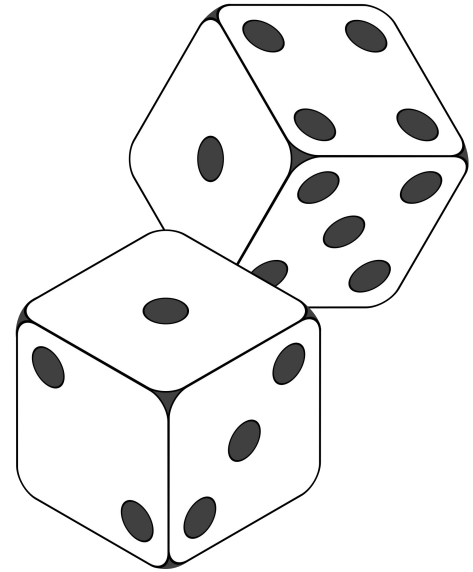


# Today's Goals

- Introduce Search-Based Test Generation
  - (AKA: Fuzzing)
  - Test Creation as a Search Problem
    - Metaheuristic Search Algorithms
    - Fitness Functions

# Random Generation

- Randomly formulate test cases.
  - 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 you run out of time.



# Example - BMI Calculation

$$BMI = \frac{weight}{(height)^2}$$

	Age						
Classification	[2, 4]	(4, 7]	(7, 10]	(10, 13]	(13, 16]	(16, 19]	> 19
Underweight	≤ 14	≤ 13.5	≤ 14	≤ 15	≤ 16.5	≤ 17.5	< 18.5
Normal weight	≤ 17.5	≤ 14	≤ 20	≤ 22	≤ 24.5	≤ 26.5	< 25
Overweight	≤ 18.5	≤ 20	≤ 22	≤ 26.5	≤ 29	≤ 31	< 30
Obese	> 18.5	> 20	> 22	> 26.5	> 29	> 31	< 40
Severely obese	—	—	—	—	—	—	≥ 40

BMI Calc
height
weight
age
bmi_value()
classify_bmi_adults()
classify_bmi_teens_and_children()

# Example - BMI Calculation

```
def test_bmi_value_valid():
    bmi_calc = BMICalc(150, 41, 18)
    bmi_value = bmi_calc.bmi_value()
    assert bmi_value == 18.2

def test_bmi_adult():
    bmi_calc = BMICalc(160, 65, 21)
    bmi_class = bmi_calc.classify_bmi_adults()
    assert bmi_class == "Overweight"

def test_bmi_children_4y():
    bmi_calc = BMICalc(100, 13, 4)
    bmi_class = bmi_calc.classify_bmi_teens_and_children()
    assert bmi_class == "Underweight"
```

<b>BMICalc</b>
height weight age
bmi_value() classify_bmi_adults() classify_bmi_teens_and_children()

# Random Generation - BMI Example

- Create an empty test case:

```
def test_1():
```

- Instantiate the class-under-test with random values:

```
def test_1():  
    cut = BMICalc(180, 50, 40)
```

- Insert 1+ method calls or assignments to class variables.
  - Number of calls is random
  - Which method/variable is random
  - Method parameters are random values

BMICalc
height weight age
bmi_value() classify_bmi_adults() classify_bmi_teens_and_children()

```
def test_1():  
    cut = BMICalc(180, 50, 40)  
    output = cut.bmi_value()  
    cut.height = 15681  
    output2 = cut.classify_bmi_adults()
```



# Random Search

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

• However...



# 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.
- Search-based test generation based on **guess-and-check** process.

# Test Creation as a Search Problem

- Many testing goals can be measured:
  - How many exceptions were thrown?
  - How many representative output values were returned?
  - What percentage of lines of code were covered?
  - How diverse is our input?
- 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 test suites.
- **Check whether goal is met.**
  - Score each guess.
- **Try until time runs out.**
  - Alter the solution based on feedback and try again!

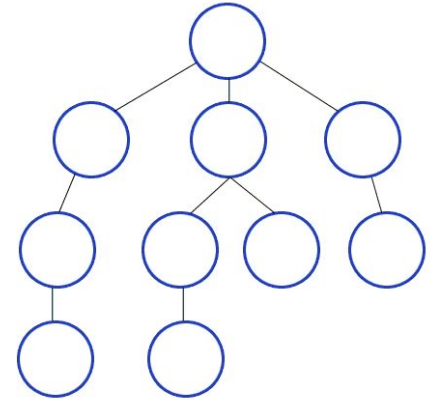


# Search Strategy

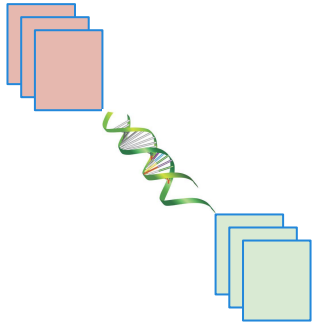
- The order that solutions are tried is the key to efficiently finding a solution.
- A search follows some defined strategy.
  - Called a “**metaheuristic**”.
- Metaheuristics are used to choose solutions and to ignore solutions known to be unviable.
  - Smarter than pure random guessing!

# 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.



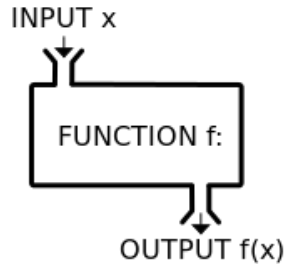
# Search-Based Test Generation



## 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,  
(...)

# Solution Representation



# Solution Representation

- Must decide what a solution “looks like”.
- For unit testing:
  - A solution is a test suite.
  - A test suite contains 1+ test cases.
  - Each test case interacts with a class-under-test.
  - Each test case initialized the class-under-test.
  - Each test case contains one or more actions.
    - An action is a method call or variable assignment.
    - Each action has parameters (method parameters or values to assign to variables).

# External vs Internal Representation

## Internal (Genotype) Representation

Can be easily manipulated by metaheuristic

```
[ Test Suite
  [ Test Case
    [-1, [246, 680, 2]],
    [2, [18]],
    [4, []],
    [1, [466]],
    [5, []],
    [4, []],
    [1, [26]],
    [5, []]
  ]
]
```

Actions,  
with ID  
(method or  
variable),  
parameters

## External (Phenotype) Representation

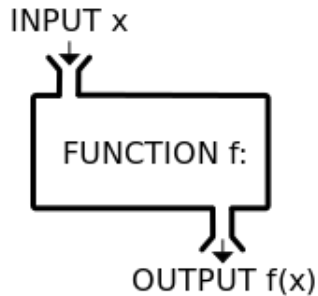
Executable, human-readable

```
1 import pytest
2 import bmi_calculator
3
4 def test_0():
5     cut = bmi_calculator.BMICalc(246, 680, 2)
6     cut.age = 18
7     cut.classify_bmi_teens_and_children()
8     cut.weight = 466
9     cut.classify_bmi_adults()
10    cut.classify_bmi_teens_and_children()
11    cut.weight = 26
12    cut.classify_bmi_adults()
```

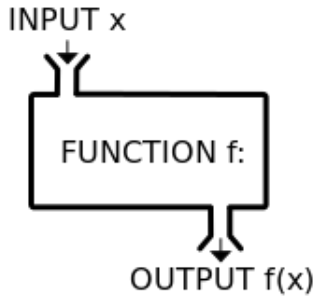
# Fitness Functions

# Fitness Functions

- Domain-based scoring functions that determine how good a potential solution is.
  - Should represent goals of tester.
  - Must return a numeric score.
    - % of a checklist
    - raw number
    - NOT Boolean (no feedback)
  - Can be maximized or minimized.



# Fitness Functions



- **Should offer feedback:**
  - Small change in solution should not lead to large change in score.
  - Best functions calculate *distance* to optimality.
- **Can optimize more than one at once.**
  - Independently optimize functions
  - Combine into single score.

# Example - Code Coverage

- **Goal:** Attain Branch Coverage over the code.
  - Tests must reach all branching points (i.e., if-statement) and execute all possible outcomes.

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

In this code:

- Two Branches
- Each must evaluate to true and false.

# Example - Code Coverage

- **Goal:** Attain Branch Coverage over the code.
- **Fitness function (Basic):**
  - Measure coverage and try to maximize % covered.
  - **Good:** Measurable indicator of progress. Can use standard tools (pytest-cov, Cobertura).
  - **Bad:** No information on how to improve coverage.

# Example - Code Coverage

- Advanced: 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  $x \neq 10$  evaluates to false, branch distance =  $(\text{abs}(x-10)+k)$ .
- 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)**

# Bloat Penalty

- Small penalty subtracted from fitness.
- Limits number of tests and number of actions.

$$\text{bloat\_penalty}(\text{solution}) = (\text{num\_test\_cases} / \text{num\_tests\_penalty}) \quad \text{ex. 10}$$
$$+ (\text{average\_test\_length} / \text{length\_test\_penalty})$$

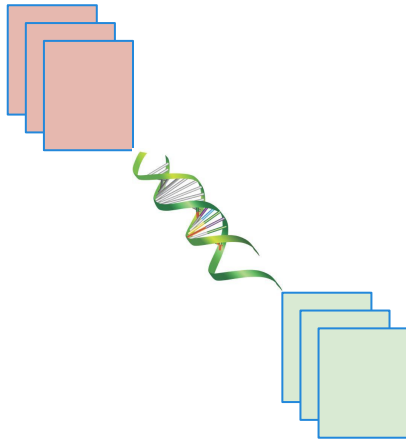
ex. 30

- Important not to penalize too heavily.

**Let's take a break.**

# Metaheuristic Algorithms

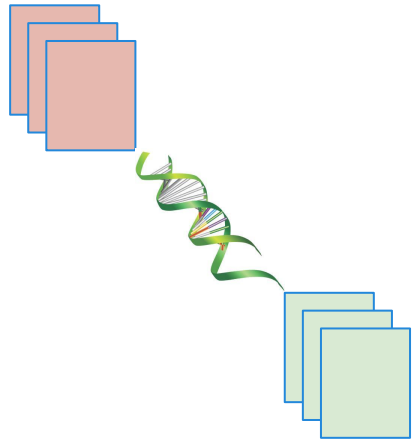
# The Metaheuristic



- 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 changes to single solution (**local search**).
  - Large changes to many solutions (**global search**).
  - Often based on natural phenomena.
    - (swarm behavior, evolution)
  - Trade-off between speed, complexity, and understandability.



# How Long Do We Spend Searching?

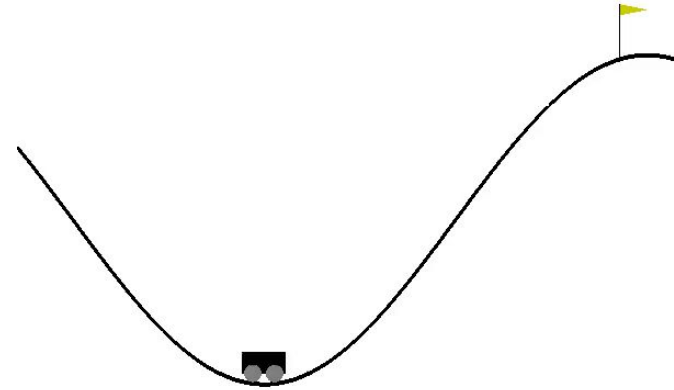
- Exhaustive search not viable.
- Search can be bound by a **search budget**.
  - Number of guesses.
  - Time allotted to the search (number of minutes/seconds).
- **Optimization problem:**
  - *Best solution possible before running out of budget.*



# Local Search

- Generate and score a single 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.

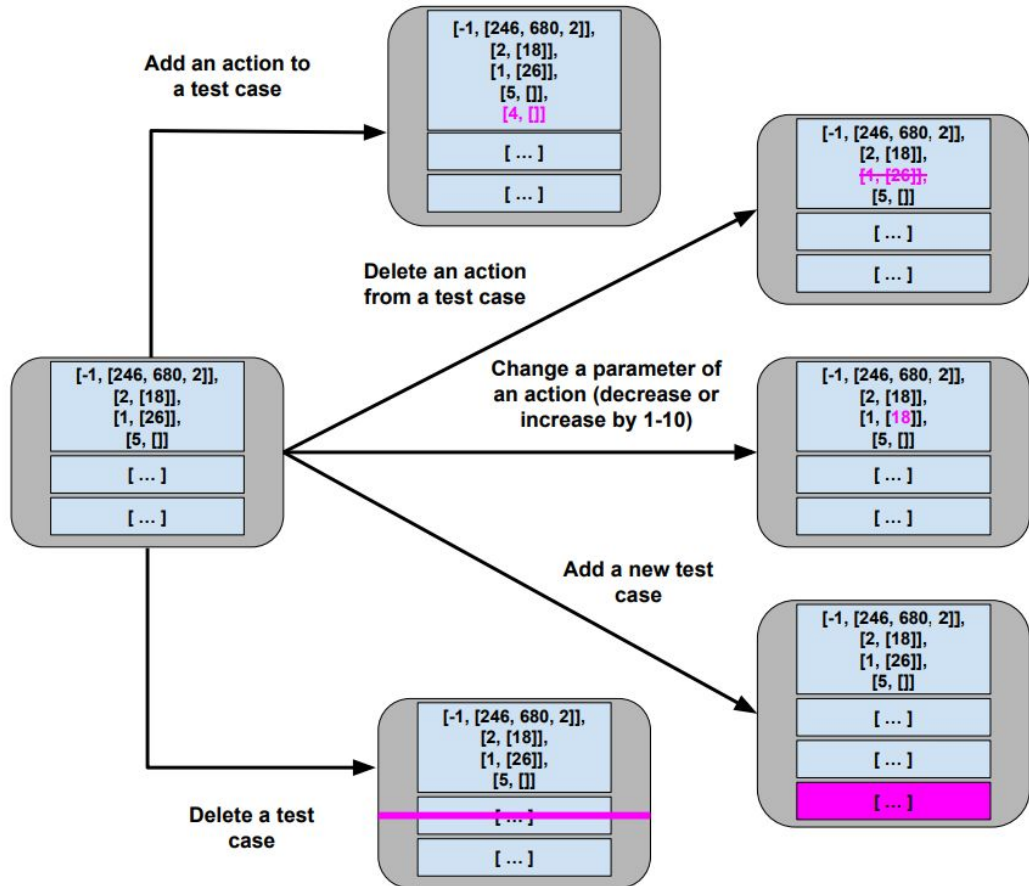
# Hill Climbing



- Generate a random initial solution.
- Each generation (while budget remains):
  - Attempt up to `max_tries` *mutations* to the solution.
    - If a mutation results in a better solution, set this as the new solution.
    - Keep track of the best mutation seen to date.
  - If we run out of tries, reset to a new random initial solution.

# Mutation

- Small change to current solution.
- Impose one of these changes at a time:

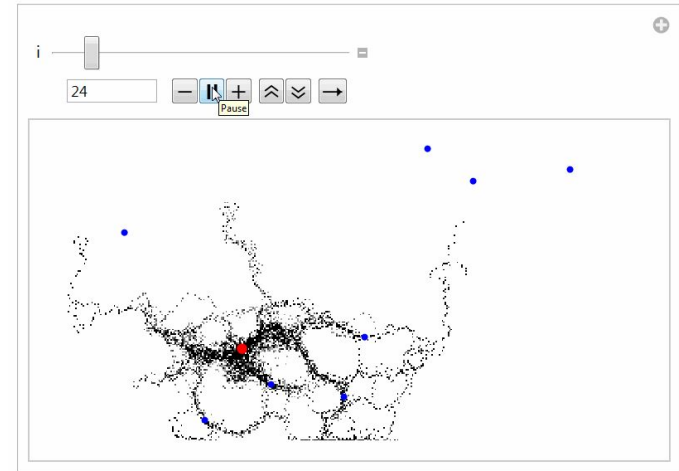


# Hill Climber

- User-Controlled Parameters:
  - Maximum mutations before a restart (ex: 200)
  - Maximum number of restarts (ex: 5)
- Easy to implement, faster than many other metaheuristics.
  - Reliant on initial guesses and restarts.

# 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 Algorithm

- Over multiple generations, evolve a population.
  - Good solutions persist and reproduce.
  - Bad solutions are filtered out.
- Diversity is introduced by:
  - **Selecting** the best solutions.
  - Creating “offspring” through **mutation** and **crossover**.

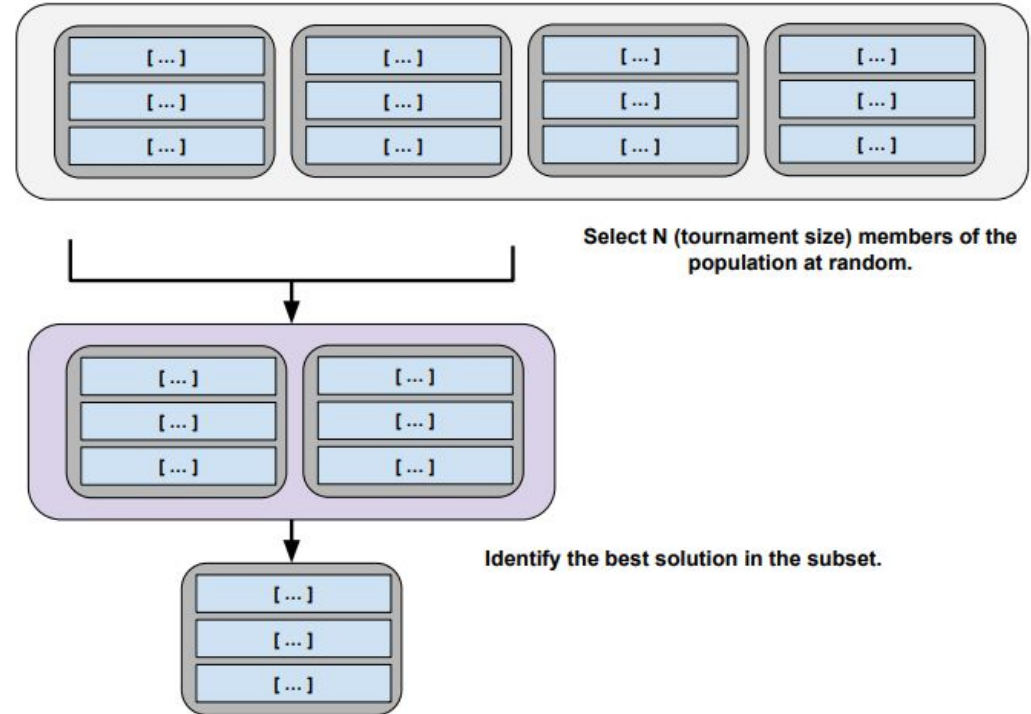


# Genetic Algorithm

- Create a random initial population.
- Start a new generation (while budget remains):
  - Create new empty population.
  - While space remains:
    - **Select** two “good” members of current population.
    - At a small probability, replace these members with “children” combining genes of members (**crossover**).
    - At a small probability, **mutate** each member.
    - Add members to **new population**.
  - If no better solution is found for N generations, terminate early (**stagnation**).

# Selection

- Rather than searching for best population member:
  - Select a random subset.
  - Calculate fitness for each.
  - Return best.





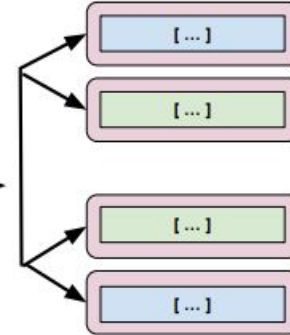
# Crossover

- Creates two “child” solutions by combining tests from each parent solution.

Select two “parent” test cases.



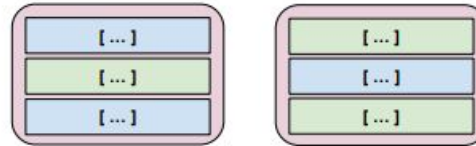
For each test case T,  
“flip a coin”



If (1), Child A gets test T from Parent A. Child B gets test T from Parent B.

If (2), the reverse happens.

Return “children” that blend elements of Parents A and B.



# Crossover

- Single-Point Crossover
  - Splice at crossover point.
- Uniform Crossover
  - Flip coin at each test, second child gets other option.
- Discrete Recombination
  - Flip coin at each test for both children.

A	B	C	D
1	2	3	4

A	B	3	4
1	2	C	D

A	B	C	D
1	2	3	4

A	2	3	D
1	B	C	4

A	B	C	D
1	2	3	4

A	2	C	4
A	B	3	4

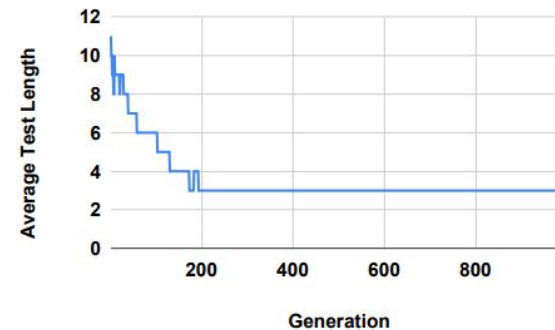
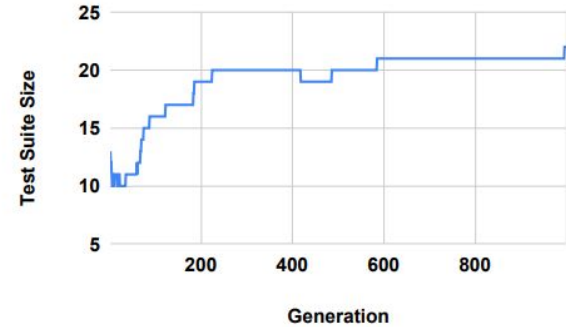
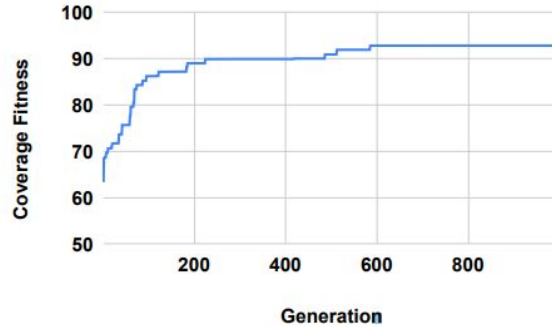
# Genetic Algorithm Parameters

- All parameters affect solution quality. Usually some experimentation required.
  - Population Size (default: 20)
  - Tournament Size (# population members compared during selection, default: 6)
  - Crossover Probability (default: 0.7)
  - Mutation Probability (default: 0.7)
  - Stagnation Threshold (# generations without improvement before ending, default: 30)

# Examining the Resulting Test Suites

# 1000 Generations of Evolution

- Genetic Algorithm run for 1000 generations for BMICalc.
- Stagnation turned off.
- Highly variable until ~ 200 generations, then small changes afterwards.



# Demo

# Examples of Generated Test Cases

```
def test_0():
    cut = bmi_calculator.BMICALc(120, 860, 13)
    cut.classify_bmi_teens_and_children()

def test_2():
    cut = bmi_calculator.BMICALc(43, 243, 59)
    cut.classify_bmi_adults()
    cut.height = 526
    cut.classify_bmi_adults()
    cut.classify_bmi_adults()

def test_5():
    cut = bmi_calculator.BMICALc(374, 343, 17)
    cut.age = 123
    cut.classify_bmi_adults()
    cut.age = 18
    cut.classify_bmi_teens_and_children()
    cut.classify_bmi_teens_and_children()
    cut.weight = 396
    cut.classify_bmi_teens_and_children()
```

```
def test_7():
    cut = bmi_calculator.BMICALc(609, -1, 94)

def test_11():
    cut = bmi_calculator.BMICALc(491, 712, 20)
    cut.classify_bmi_adults()

def test_17():
    cut = bmi_calculator.BMICALc(608, 717, 6)
    cut.classify_bmi_teens_and_children()
    cut.age = 91
    cut.classify_bmi_teens_and_children()
    cut.classify_bmi_teens_and_children()
```

# What Do I Do With These Inputs?

- If looking for crashes, just run generated input.
- If you need to judge correctness, add assertions.
  - Suggested: general properties, rather than specific expected output.
    - **No:** `assertEquals(output, 2)`
    - **Yes:** `assertTrue(output % 2 == 0)`





# Additional Concepts

# Not Just Test Generation...

Can be applied to any problem with:

- Large search space.
- Fitness function and solution generation with low computational complexity.
- Approximate continuity in fitness function scoring.
- No known optimal solution.

# 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.

# GenProg Results

- Repaired 55/105 bugs at average \$8 per bug.
  - Projects with over 5 million lines of code
  - Supported by 10000 test cases.
- Patch infinite loops, segmentation faults, buffer overflows, denial of service vulnerabilities, “wrong output” faults, and more.

# 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.

# I Want to Try This Out!

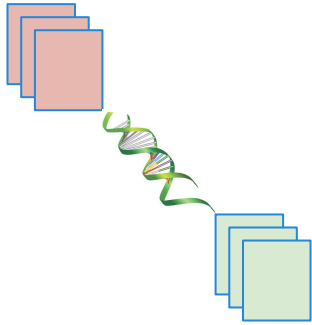
- Python:
  - Tutorial for beginners: <https://greg4cr.github.io/pdf/21ai4se.pdf>
  - <https://github.com/Greg4cr/PythonUnitTestGeneration>
- EvoSuite for Java: <http://www.evosuite.org/>



# I Want to Try This Out!

- Fuzzing often based on metaheuristic search.
  - AFL (American Fuzzy Lop), Google OSS-Fuzz use genetic algorithms, fitness = code coverage.
    - <http://lcamtuf.coredump.cx/afl/>
    - <https://google.github.io/oss-fuzz>
    - system-level tests
  - The Fuzzing Book has tutorials and code for many specialized approaches:
    - <https://www.fuzzingbook.org/>

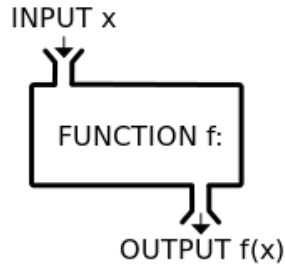
# Summary



## The Metaheuristic (Algorithm)

Genetic Algorithm  
Simulated Annealing  
Hill Climber  
(...)

+



=



## (Goals)

Cause Crashes  
Cover Code Structure,  
Maximize Battery Use,  
(...)

# Next Time

- Course summary and exam review
  - Try the practice test!
- Assignment 3 - Due March 10



UNIVERSITY OF  
GOTHENBURG

---



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY