Testing, debugging & verification

Srinivas Pinisetty

This course

Introduction to techniques to get (some) certainty that your program does what it’s supposed to.

Specification: An unambiguous description of what a function (program) should do.

Bug: failure to meet specification.

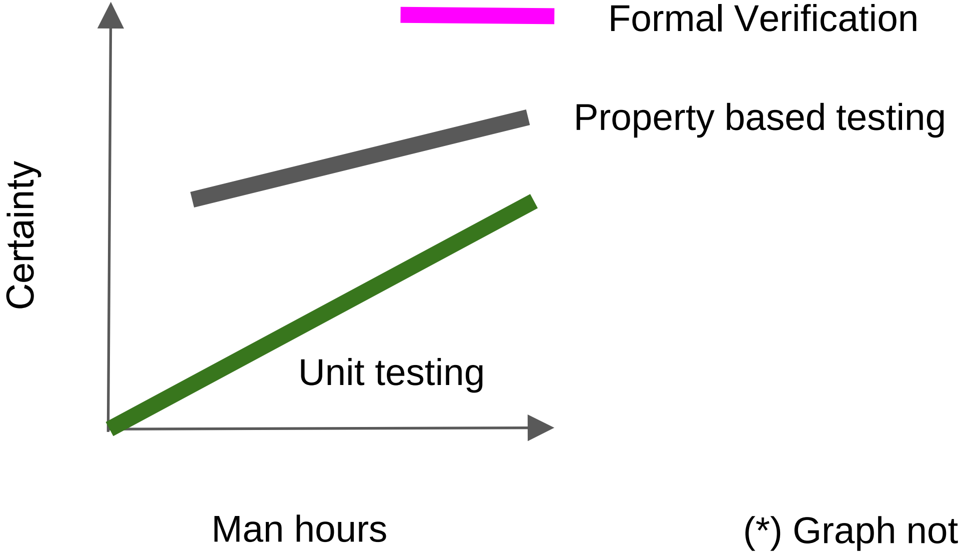
What is a Bug? Basic Terminology

- Defect (aka bug, fault) introduced into code by programmer (not always programmer's fault, if, e.g., requirements changed)
- Defect may cause infection of program state during execution (not all defects cause infection)
- Infected state propagates during execution (infected parts of states may be overwritten or corrected)
- Infection may cause a failure: an externally observable error (including, e.g., non-termination)

Terminology

- Testing - Check for bugs
- Debugging – Relating a failure to a defect (systematically find source of failure)
- Specification - Describe what is a bug
- (Formal) Verification - Prove that there are no bugs

Cost of certainty



(*) Graph not based on data,
only indication

More certainty = more work

Contract metaphor

Supplier: (**callee**)
Implementer of method

Client: (**caller**) Implementer of
calling method or user

Contract:

Requires *(precondition):* *What the **client** must ensure*

Ensures (postcondition)*: What the **supplier** must ensure*

- **Testing**
  - Unit testing
    - Coverage criteria
      - Control-Flow based
      - Logic Based
    - Extreme Testing
    - Mutation testing
  - Input space partitioning
  - Property based testing
  - Black-box, white-box
  - Levels of detail
  - Test driven development
- **Debugging**
  - Input Minimisation (Shrinking) :
    - 1-minimal
    - ddMin

  - Backwards dependencies:
    - data-dependent
    - control-dependent
    - backward dependent

- **Formal specification**
  - Logic
    - Propositional logic
    - Predicate Logic
    - SAT
    - SMT
  - Dafny
    - Assertions
    - range predicates
    - method, function, predicate, function method
    - modifies, framing
    - Loop invariant
    - Loop variant
    - ……
- **Formal verification**
  - Weakest precondition calculus
  - Prove program correct
  - Loop - Partial correctness
  - Loop - Total correctness

Testing

- **Unit testing**
    - Coverage criteria
        - Control-flow based
        - Logic based
    - Mutation testing
- Input space partitioning
- Property based testing
- Black-box, white-box
- Levels of detail
- Test driven development

Testing

Testing can give some certainty about code, but typically not any guarantees

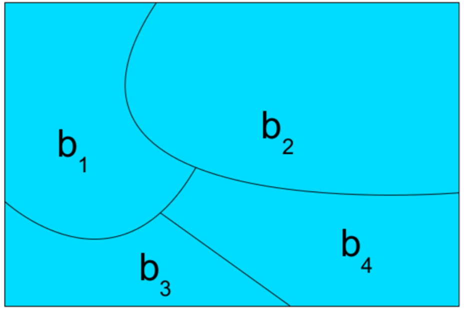
Almost always cannot test all possible inputs (practically infinite)

Unit Test

A unit(= method) tests consists of:

- Initialization
- Call to method
- Check if test fails or not

How do we pick tests?



A guideline: **Input space partitioning**

1. Look at specification
2. Divide input space into regions with for which the program acts "similar".
3. Take some inputs from each *region*, especially from *borders*

Use multiple partitions, or subdivide partitions when sensible

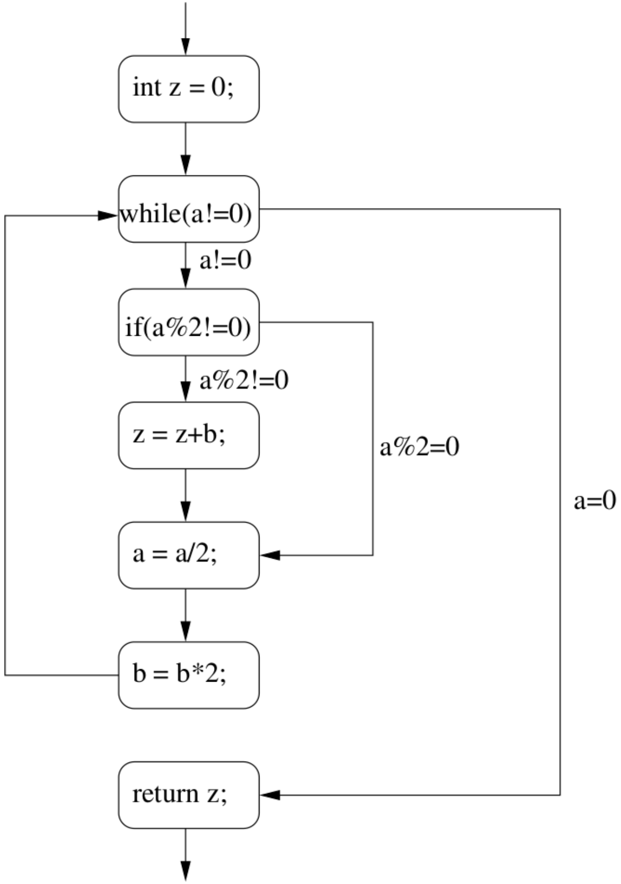
This is a guideline, not a formal procedure: use common sense to define "similar","border" and "sensible"

Coverage criteria

- **Motivation:**
  - How do we know if enough unit tests?
- **An answer**: Check how much of the code is “covered” by the unit tests?
- **Ways of defining covered**:
  - **Control Flow based**
    - Statement coverage
    - Branch coverage
    - Path coverage
  - **Logic Based**
    - Decision coverage
    - Condition coverage
    - Modified condition decision coverage
  - Full coverage does not give guarantee!

Control-flow based coverage

Control-flow based coverage

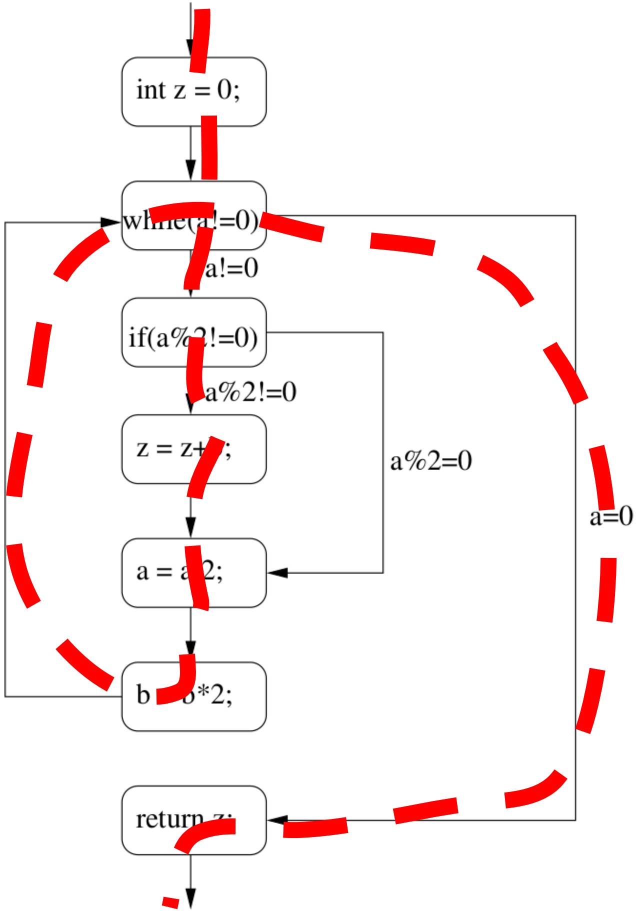


- ***Control flow graph*:**
  - *Node* = statement or start of while/if/for
  - *Edge* from **a** to **b** iff next execution step after **a** can be **b**
  - *Label* on edge = condition which should hold to traverse edge (or no condition)
- *Execution path of unit test*: path followed through graph by executing test

- ***Statement coverage***: for each ***node***, there exists a test, such that the node is visited by the execution path of that test
- ***Branch coverage***: for each ***edge***, there exists a test, such that the edge is traversed in the execution path of that test

Statement coverage: Example

```
int russianMultiplication(int a, int b){
  int z = 0;
  while(a != 0){
    if(a%2 != 0){
      z = z+b;
    }
    a = a/2;
    b = b*2;
  }
  return z;
}
```



Each test case has an execution path.

`russianMultiplication(1,0) == 0`

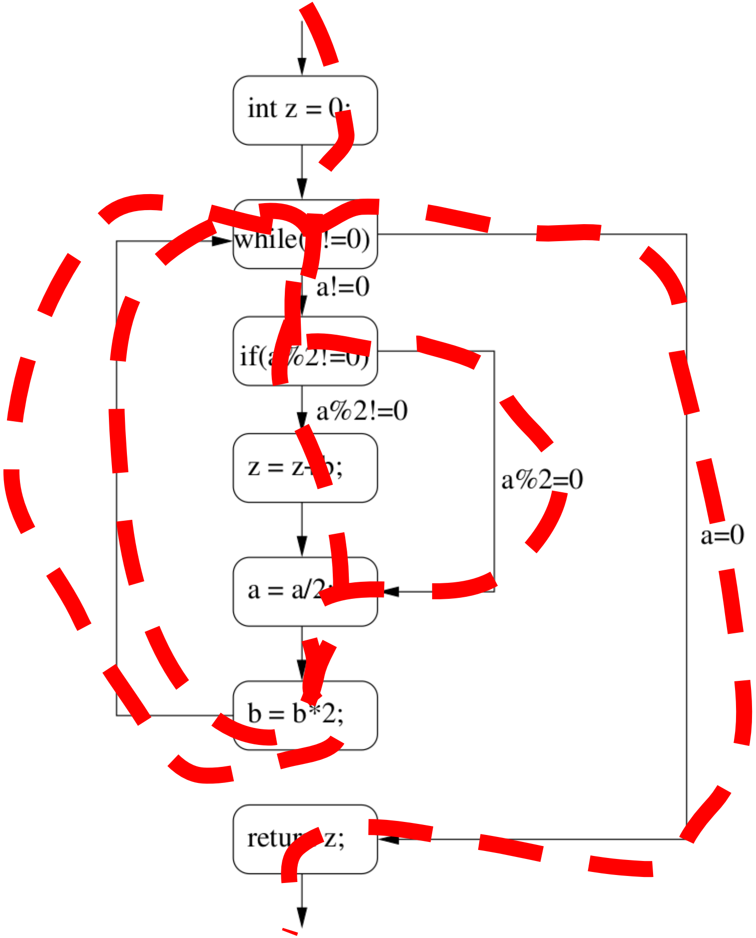
Note: all nodes are visited, so statement coverage

Branch coverage: Example

```
int russianMultiplication(int a, int b){
  int z = 0;
  while(a != 0){
    if(a%2 != 0){
      z = z+b;
    }
    a = a/2;
    b = b*2;
  }
  return z;
}
```

Branch coverage: is each *edge* taken in a test case?

`russianMultiplication(2,0) == 0`



Logic-based coverage

Logic-based coverage

Decision: Boolean expression

Condition: Atomic boolean sub-expression (does not contain other boolean sub-expression)

Decision coverage: Each outcome(T,F) of each decision occurs in a test (implies branch coverage)

Condition coverage: Each outcome of each condition of each decision occurs in a test

Decision

```
((a < b) || D) && (m ≥ n * o)
```

conditions: (a < b), D, (m >= n * o)

Modified Condition Decision Coverage (MCDC)

Condition/decision coverage + show that each condition influences its decision independently

Condition c *independently influences* decision d if:

Changing only c changes outcome of d (for some choice of outcomes of other conditions)

Example: `((a < b) || D) && (m ≥ n ∗ o)`

Conditions: (a < b), D , m >= n * o

Show that (a < b) influences decision independently, set
{D = False, m =2, n =1, o = 1}

a	b	a < b	Result
1	2	T	T
2	1	F	F

Logical decision coverage

Decision: **if(x < 1 || y > z)**

Do the following satisfy decision, condition, MCDC?

`[x=0, y=0, z=1] and [x=2, y=2, z=1]` **: CC**

`[x=2, y=2, z=1] and [x=2, y=0, z=1]` **: DC**

`[x=2, y=2, z=2], [x=0, y=0, z=1], [x=2, y=0, z=0], [x=2, y=2, z=1]` **:**

CC, DC, MCDC

Black box - white box

- Black-box testing: Create tests only based on externals (specification) without knowing internals (source code)

- White-box testing: Create test based on externals & internals

Mutation Testing

How do we know we have enough test cases?

One answer: coverage criteria
Another answer: **mutation testing**

Mutation testing:
Can lead to identifying some holes in test set, but does not give certainty!

Mutation testing overview

1. (Automatically, randomly) Change (**mutate**) the function under test a bit

2. The new function(**mutant**) should now be incorrect (we hope)

3. Is there a test that now fails (test that “kills” the mutant)? If so, good. If not, maybe a test is missing?

Trivial example

Requires:

Ensures: result == a ⇒b

```
boolean implies(boolean a, boolean b){ return !a || b; }
```

Tests:

```
implies(true,true) (== true)
!implies(true,false)
```

Mutant not killed! Add more tests!

Mutant:

```
boolean implies(boolean a, boolean b){ return a && b; }
```

Extra Tests:

```
implies(false,true)
implies(false,false)
```

Mutant killed! Good!

Example mutation steps

- Delete statement
- Statement duplication
- Replace boolean expression with true or false
- Replace > with ≥
- Replace 0 with 1
- ...

Another example

Requires: arr is sorted in non-decreasing order and low <= high
Ensures: result = number of values in arr in interval [low,high]

```
int nrInInterval(int[] vals, int low, int high) {
  int i = 0;
  while(i < arr.length && arr[i] < low)  { i+= 1; }
  int res = 0;
  while(i < arr.length && arr[i] <= high) { i+=1;
res+=1; }
  return res;
```

```
tests: nrInterval({1,2,4,6,8,11}, 2, 7) == 3
```

```
int nrInInterval(int[] vals, int low, int high) {
  int i = 0;
  while(i < arr.length && arr[i] < low)  { i+= 1; }
  int res = 0;
  while(i < arr.length && arr[i] < high) { i+=1; res+=1;
}
  return res;
```

Mutant that is not killed?

Mutation testing

- **Tools**:
    - MuJava
    - Mutator (Java, Ruby, JavaScript and PHP)
- **Gives some indication of test set quality, but**:
    - If input space/output space is infinite and nr. tests finite, it is always possible to change program such that all tests succeed but does not conform to spec (if all changes allowed)
    - Perfect mutation score (i.e. all mutants killed) does not mean perfect test set (randomness, not all possible changes covered)

Property based testing

Property based Testing - motivation

- Writing units test takes a lot of effort!
- More unit test = more certainty
- Automate!

cr01s025 www.fotosearch.com

Property based testing =

Generate *random* inputs and check that a property of the output holds

Different properties to test:

- Postcondition holds
- …

Example - test that postconditon holds

```
int[] sort(int[] input)
```

Specification:
Requires: A non-null array as input
Ensures: A new array that is sorted in ascending order, that is a permutation of the input array

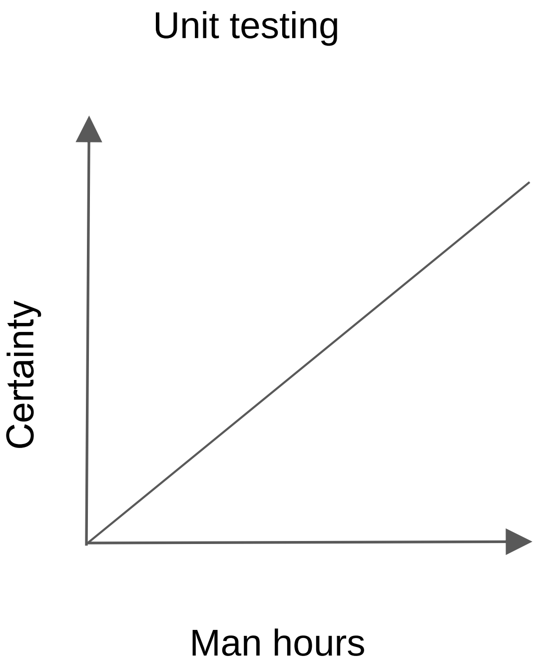
Property based Testing

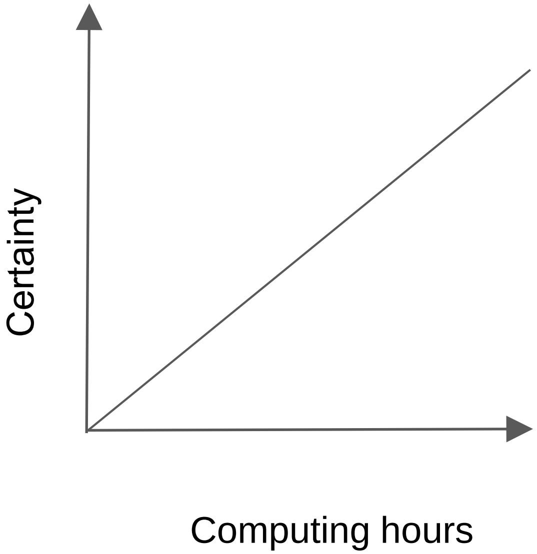
- Generate a *random* input that satisfies the *precondition*
- Feed it to the function
- Check that a property on the output holds (postcondition)

```
bool singleTest(){
  int[] input  = generateRandomArr();
  int[] output = sort(input);
  return isSorted(output) && isPermutationOf(output, input);
}
```

Doesn’t have to be efficient! Run many times!

Unit Testing vs Property based testing





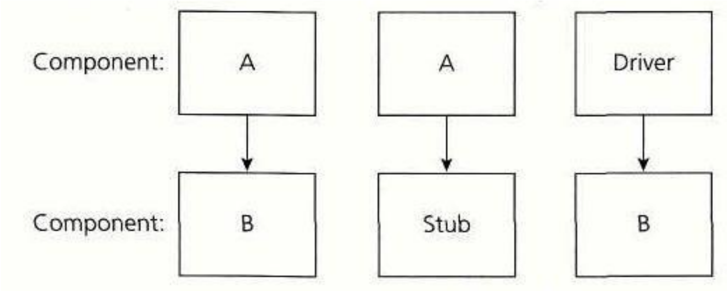
Man hours = expensive, Compute time = cheap

Terminology

- Regression testing: (Automatically) run all tests again after change in code
- Automated testing: Store tests (and their outcomes) so that we can automatically run them
- Continuous integration: A server checks out the current version of the code periodically, builds code and runs tests

- Stub: placeholder implementation of a leaf function

- Driver: placeholder for a calling function, sets up the context



Testing levels

- *Acceptance testing:* Test against user-requirements
- *System Testing :* Test against system-level specification
- *Integration Testing*: Testing interaction between modules
- *Unit testing:* Testing unit (method)

Debugging

- Debugging steps
- Input Minimisation (Shrinking)
  - ddMin
- Backwards dependencies:
  - data-dependent
  - control-dependent
  - backward dependent

Debugging Steps

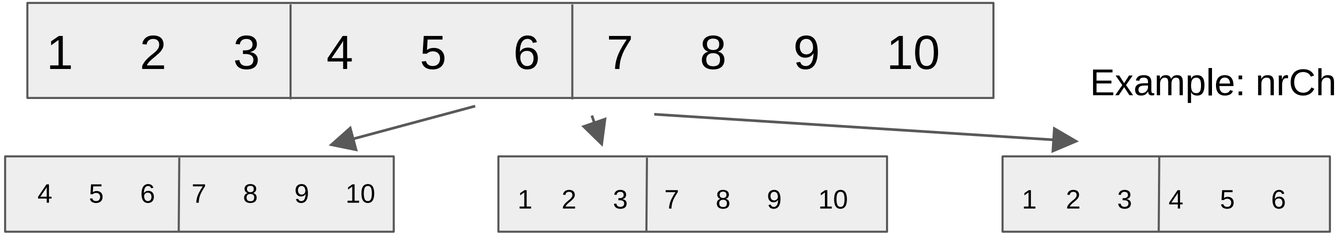
1. Reproduce the error, understand
2. Isolate and Minimize (shrink)– Simplification
3. Eyeball the code, where could it be?– Reason backwards
4. Devise and run an experiment to test your hypothesis
5. Repeat 3,4 until you understand what is wrong
6. Fix the Bug and Verify the Fix
7. Create a Regression Test

Separate relevant from irrelevant

Being systematic: avoid repetition, ensure progress, use tools

The ddMin Algorithm (Automatic input simplification)

- Overview
  - Consider input C
  - Divide input into chunks (num. of chunks n, initially n=2)
  - Cut away a part of input Ci, does the test still fail? If so, continue without that part (i.e., C = C\Ci), with n = max(n-1,2)
  - When no failure occurs when we cut away any part: Increase granularity (* 2) (number of chunks n = min(2*n, |C|))
  - Done when cutting away doesn’t help anymore and nrChunks = length of input



1 2 3 | 4 5 6 | 7 8 9 10

4 5 6 | 7 8 9 10

1 2 3 | 7 8 9 10

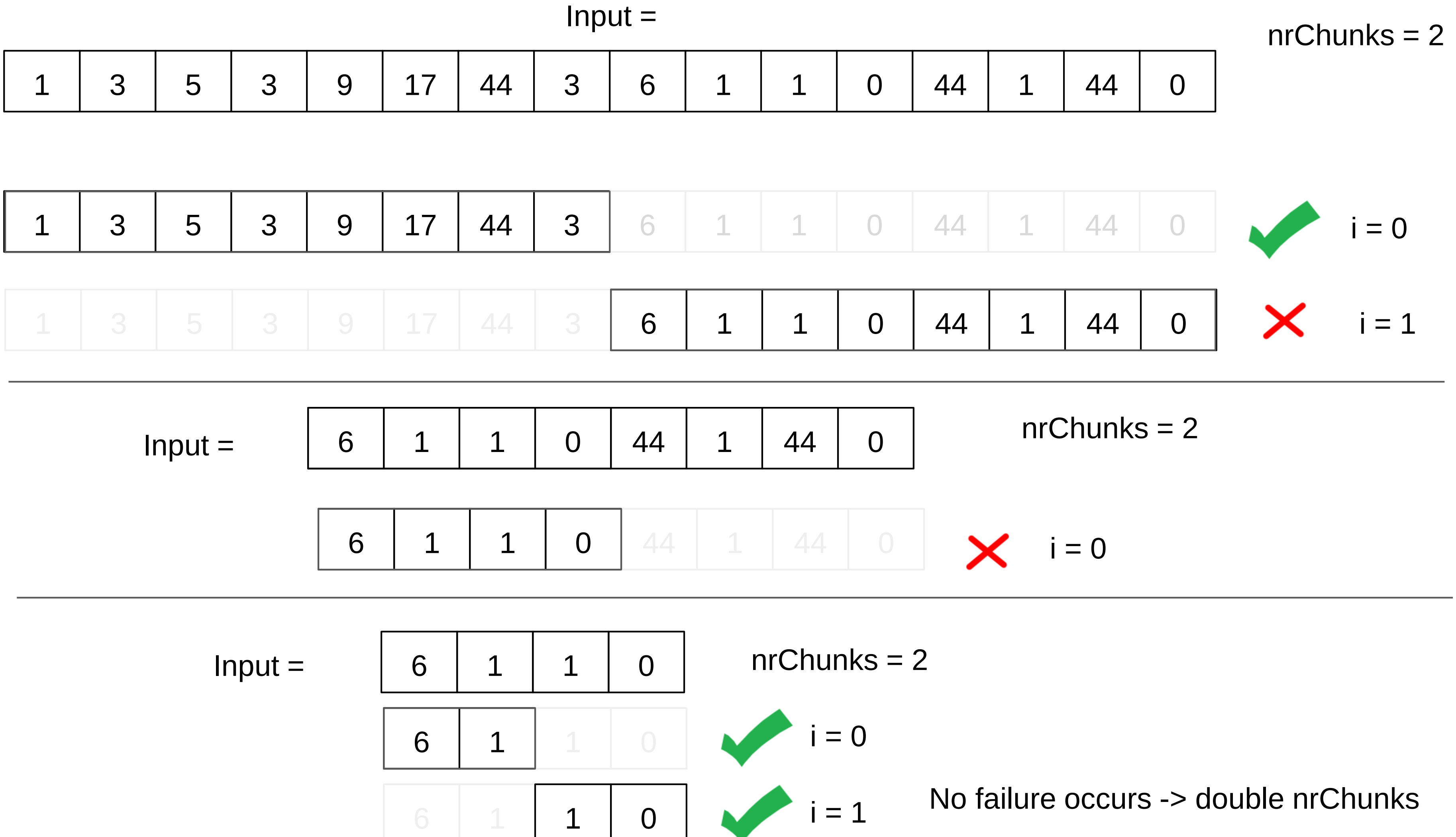
1 2 3 | 4 5 6

Example: nrChunk = 3

public static int checkSum(int[] a)

- is supposed to compute the checksum of an integer array
- gives wrong result, whenever the array contains two identical consecutive numbers *(but we don’t know that yet!)*
- we have a failed test case, e.g., from protocol transmission: {1,3,5,3,9,17,44,3,6,1,1,0,44,1,44,0}

Want to get: {1,1},{3,3} or {44,44}



Input =

nrChunks = 2

1	3	5	3	9	17	44	3	6	1	1	0	44	1	44	0

1	3	5	3	9	17	44	3	6	1	1	0	44	1	44	0	✓	i = 0
1	3	5	3	9	17	44	3	6	1	1	0	44	1	44	0	✗	i = 1

Input =

6	1	1	0	44	1	44	0

nrChunks = 2

6	1	1	0	44	1	44	0	✗	i = 0

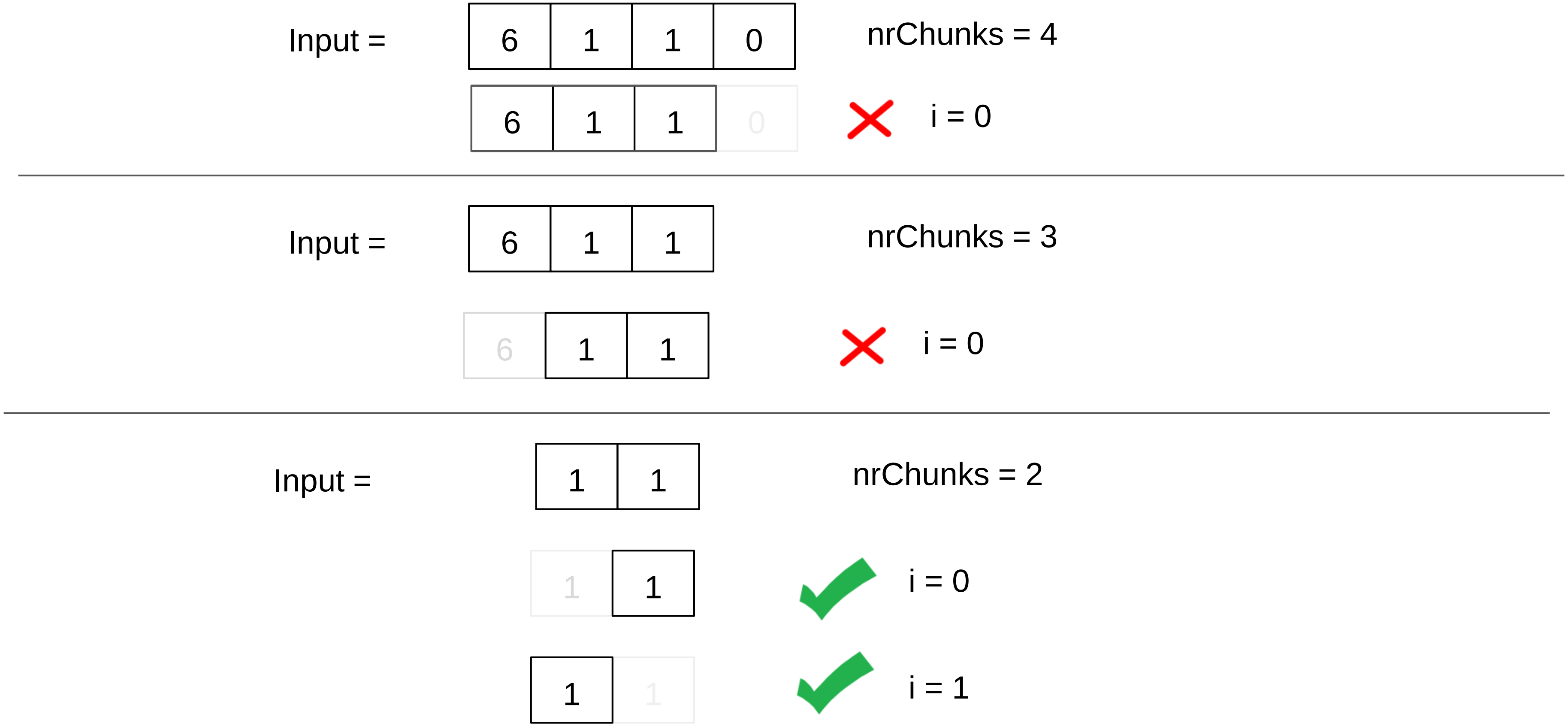
Input =

6	1	1	0

nrChunks = 2

6	1	1	0	✓	i = 0
6	1	1	0	✓	i = 1

No failure occurs -> double nrChunks



Input = | 6 | 1 | 1 | 0 | nrChunks = 4

| 6 | 1 | 1 | 0 | ✗ i = 0

Input = | 6 | 1 | 1 | nrChunks = 3

| 6 | 1 | 1 | ✗ i = 0

Input = | 1 | 1 | nrChunks = 2

| 1 | 1 | ✓ i = 0

| 1 | 1 | ✓ i = 1

No failure occurs and nrChunks = length of input --> done!

Result = | 1 | 1 |

ddMin

- See lecture slides for algorithm, examples
- Practice other examples (exercises, sample exams)..

Backwards dependencies

Statement **B** is *control-dependent* on **A** iff **A** influences whether **B** is executed.

More formally, Statement **B** is control dependent on statement **A** iff:

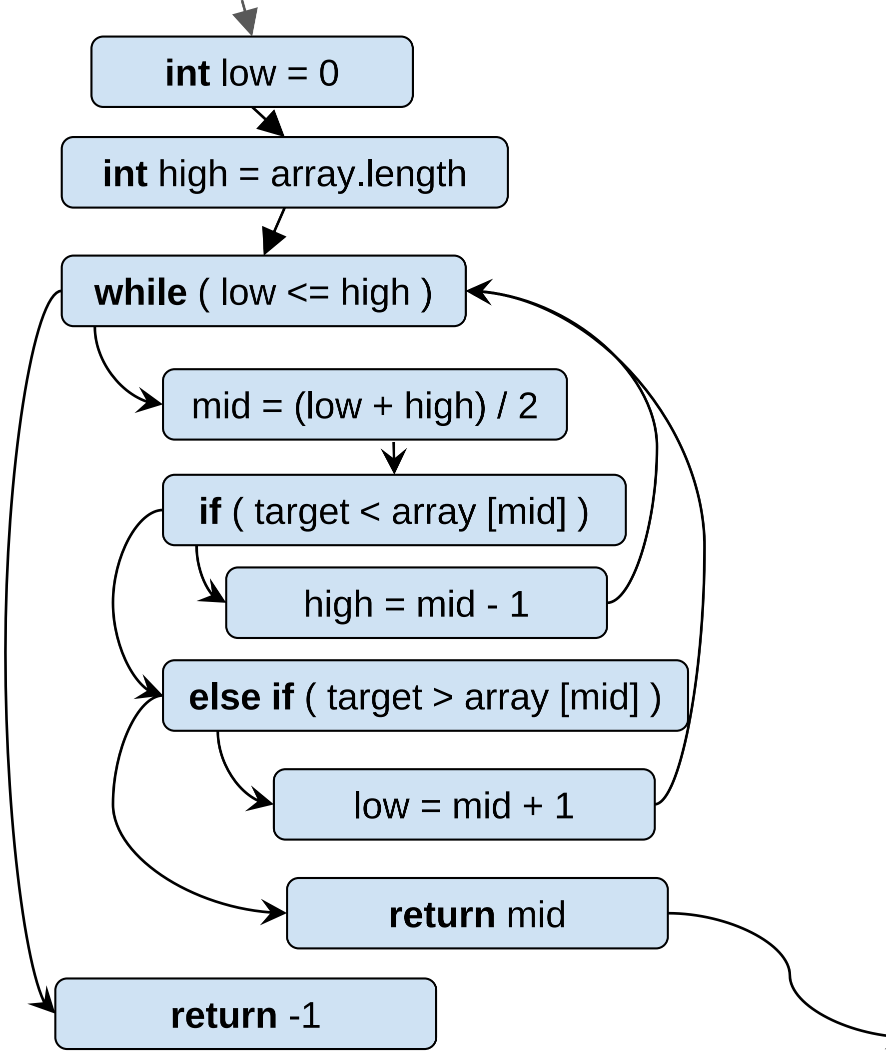
- **A** is a control statement (while, for, if or else if)
- Every path in the control flow graph from the start to **B** must go through **A**.

Statement **B** is *data-dependent* on **A** iff **A** writes a variable that **B** reads.

More formally, Statement **B** is data dependent on statement **A** iff:

- **A** writes to a variable **v** that is read by **B**
- There is at least one execution path between **A** and **B** in which **v** is not assigned another value.

*``The outcome of **A** can directly influence a variable read in **B**''*



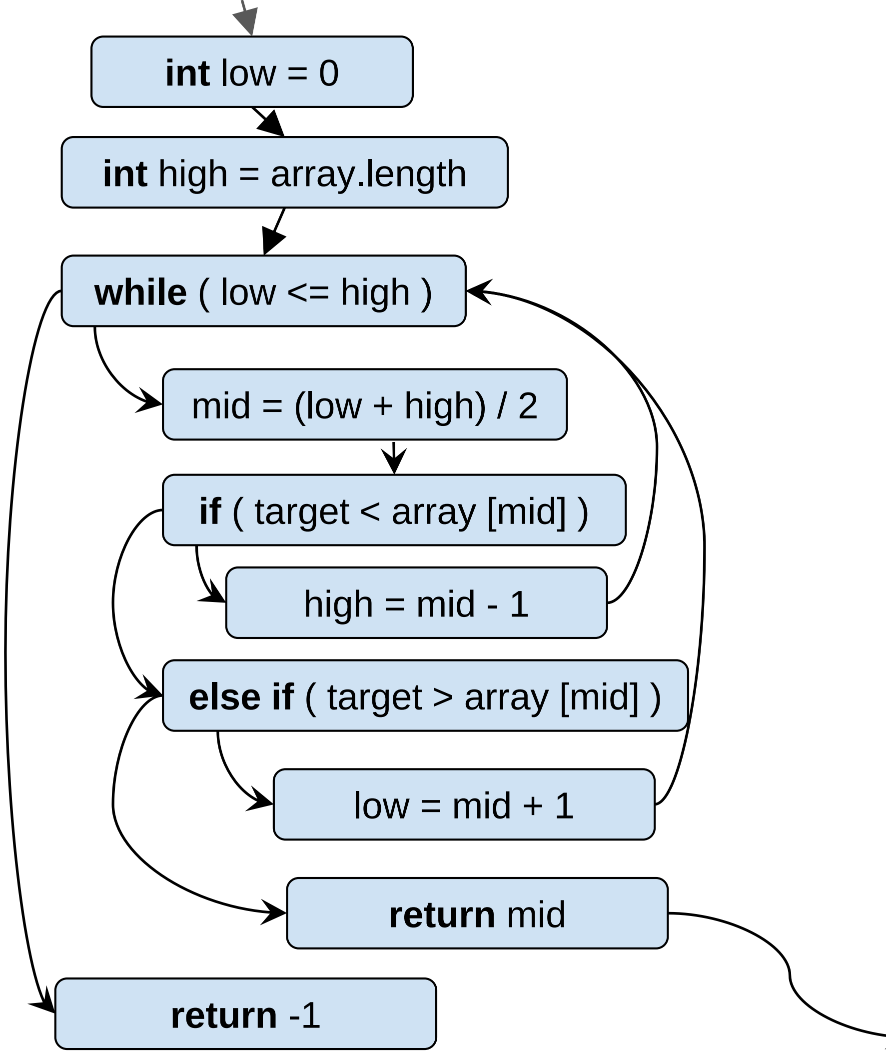
Backwards dependencies

Statement **B** is (Directly) backwards dependent on **A** if either or both:

B is control-dependent on **A**

B is data-dependent on **A**

Statement **B** is backwards dependent on **A** if **B** is directly backwards dependent on **A** in one or more steps



- **Formal specification**
  - **Logic**
    - Propositional logic
    - Predicate Logic
    - SAT
    - SMT
  - **Dafny**
    - Programming & Specification language
    - Framing
    - Loop invariant
    - Loop variant

Motivation: Write specification in fully formal language such that the computer can check for no bugs

Propositional Logic

Formula consist of Boolean variables and ¬ (!), ⋎(||), ⋏ (&&),⇒(==>) , ⇔ (<==>)

Truth table:

p	**q**	**p ⋎ q**	**q ⇒ p**	**(p ⋎ q) ⋏ (q ⇒ p)**
F	F	F	T	F
F	T	T	F	F
T	F	T	T	T
T	T	T	T	T

A propositional formula F is...

- ***satisfiable*** if F can be True (there is at least one row where the rightmost column is T)
- **valid** if F is always True (the rightmost column is T for each row)

First-order logic (Predicate logic)

Extends propositional logic by:

- **Types**, other than boolean e.g. int, real, BankCard, ...
- **Functions** (mathematical) e.g. +, max, abs, fibonacci,...
- Constants are functions with no arguments e.g. 0, 1,
- **Predicates** (functions returning a boolean) e.g. isEven, >, isPrime...
- **Quantifiers** for all (∀), there exists (∃)

First-order logic: Examples

All elements of arr are positive

$\forall i : \mathcal{Z},\ 0 \leq i < arr.Length \Rightarrow arr[i] \geq 0$

There is a positive element in the array

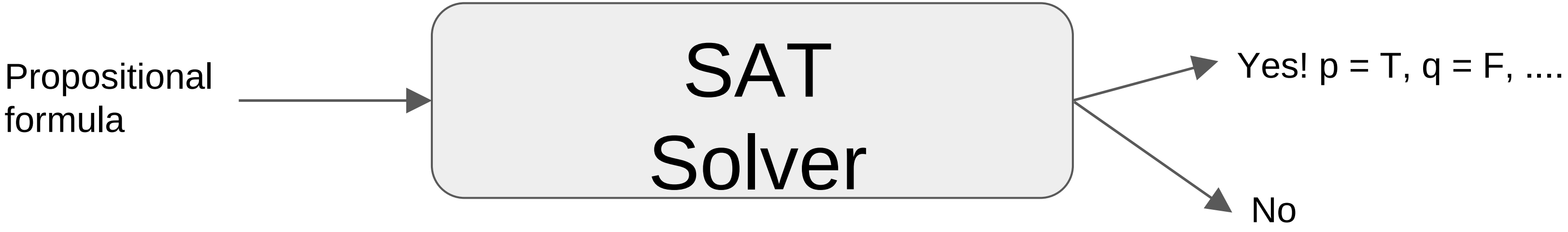
$\exists\ i : \mathcal{Z},\ 0 \leq i < arr.Length \wedge arr[i] \geq 0$

Expressing specifications in FOL

- Practice examples (lectures, exercises, labs,..)

SAT and SMT solving

Programs that solve whether formula is satisfiable



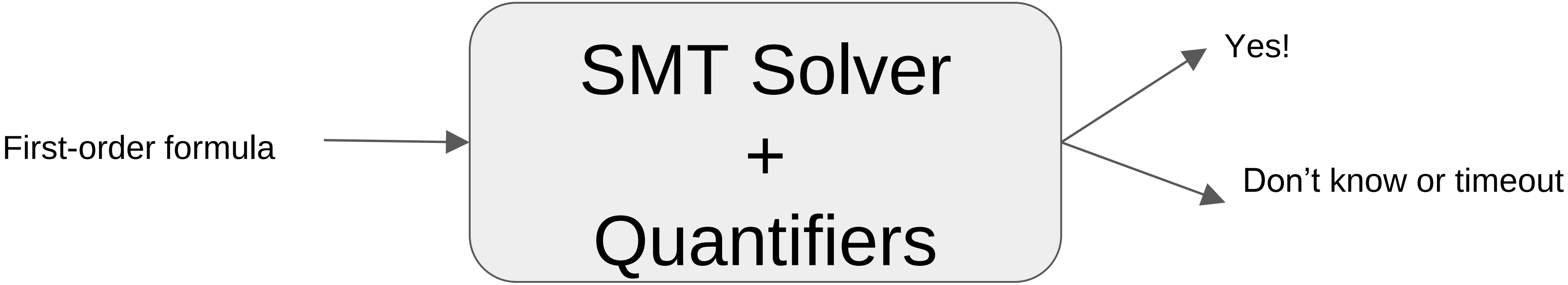
Propositional formula

SAT Solver

Yes! p = T, q = F, ...

No

Can also be used to check if formula P is a tautology:
Check that ¬P is *not* satisfiable



First-order formula

SMT Solver + Quantifiers

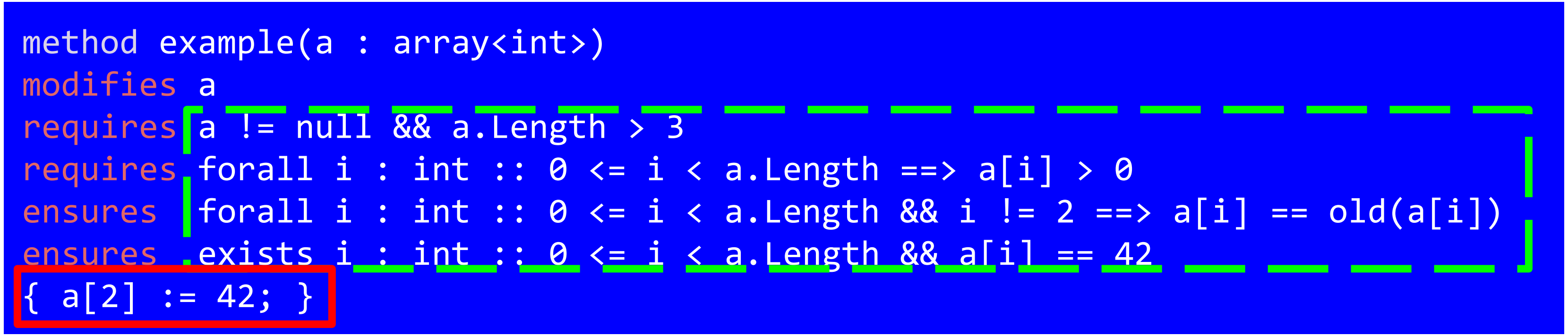
Yes!

Don’t know or timeout

Dafny

- Dafny is an imperative language with integrated support for formal specification and verification.
- Assert = prove, not check
- Pre/post conditions written in first order logic
- Automatically proved, rejected otherwise

Dafny: 2 for the price of 1



```
method example(a : array<int>)
modifies a
requires a != null && a.Length > 3
requires forall i : int :: 0 <= i < a.Length ==> a[i] > 0
ensures forall i : int :: 0 <= i < a.Length && i != 2 ==> a[i] == old(a[i])
ensures exists i : int :: 0 <= i < a.Length && a[i] == 42
{ a[2] := 42; }
```

2 languages in Dafny. Their unique properties:

Programming language

- Assignments
- While loops
- Methods
- Executed at runtime

Specification language

- Quantifiers
- Old values still available
- Functions
- Used only for checking, ignored at runtime

Dafny: Syntax Example

```
method BSearch(a : array<int>, e : int) returns (r : int)
requires a != null && Sorted(a)
ensures if (exists i :: 0 <= i < a.Length && a[i] == elem)
        then 0 <= r < a.Length && a[r] == elem else r < 0
{
    var low, high := 0 , arr.Length;
    while(low < high)
    invariant 0 <= low <= high <= arr.Length
    invariant forall i :: (0 <= i < low ||
           high <= i < arr.Length) ==> arr[i] != elem
    {
        var mid := (low + high) / 2;
        if e < a[mid]       { high := mid; }
        else if e > a[mid]  { low := mid + 1; }
        else                { return mid; }
    }
    return -1;
}
```

Inside test

```
method abs(a : int) returns (r : int)
ensures r >= 0
{
  if a < 0 {r := -a; }
  else {r := a; }
}

method test(){
         var r := abs(-3);
         assert r == 3;
}
```

This is rejected by Dafny! Why?

Dafny only uses annotations (requires & ensures) of *other* methods to prove things.

Framing

Dafny requires you to state which variables are:

- Read (for functions)
- Modified (for methods)

```
class Set{
   var elems : array<int>;
   var nr : int;

function nrFree() : int
  requires elems != null
  reads `nr, `elems
  { elems.Length - nr }

   method addAll(Set other) {
   modifies elems, `nr
   ...
}
```

Efficiency

We know that a the value of an expression only changes if:

Something is *modified* that the expression *reads*

```
var a = Set();
var b = Set();

a.add(1); a.add(2); a.add(3);
b.add(4); b.add(5);

// we know that b.nrFree() is the
// same before and after this
// statement
a.addAll(b);
// we also know 3 + 2 always
// gives the same, since + does
// not read anything
```

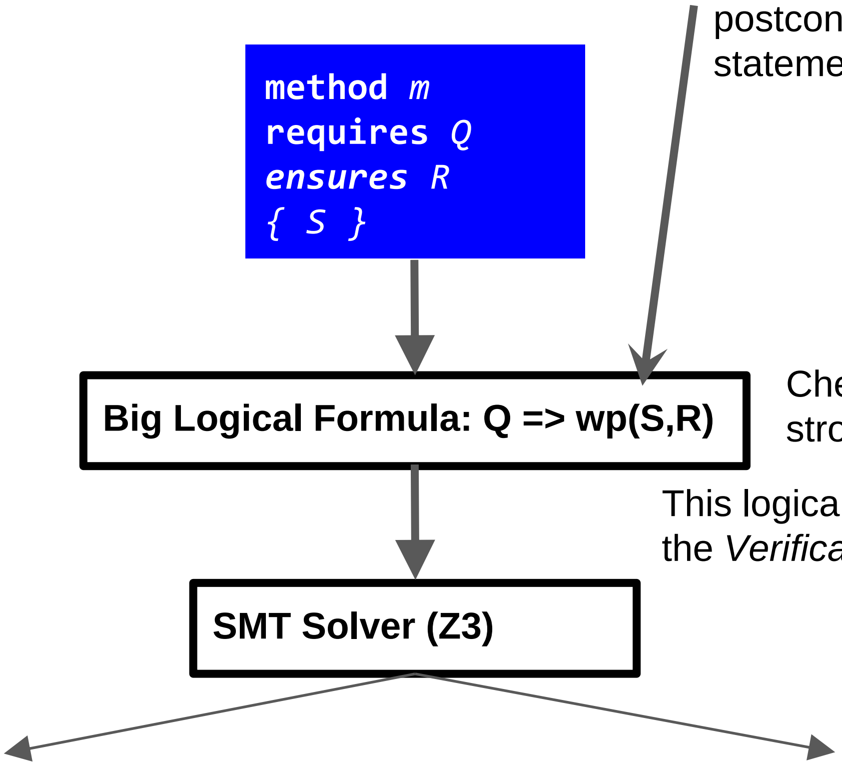
Dafny loops

```
method simpleInvariant(n : int) returns (m : int)
requires n >= 0
ensures  n == m {
  m := 0;
  while m < n // <- this is called the loop guard
  decreases (n - m)
  invariant m <= n
  { m := m + 1; }
}
```

- Dafny cannot prove correctness of loops automatically (undecidable)
- **Need: loop invariant**
  - Holds after any number of iterations of the loop (including 0)
  - invariant should be useful (help to prove postcondition)
- For full correctness we also need termination
  - **Need: loop variant (decreases clause)**
    - Must always be >= 0
    - Must decrease after each iteration

- **Formal verification**
    - How does dafny prove?
    - Weakest precondition calculus & Correctness
    - Loop - Partial correctness
    - Loop - Total correctness

How dafny works



```
method m
requires Q
ensures R
{ S }
```

wp(S,R) = weakest precondition such that postcondition R holds after executing statements S

Big Logical Formula: Q => wp(S,R)

Check if precondition is at least as strong as weakest precondition

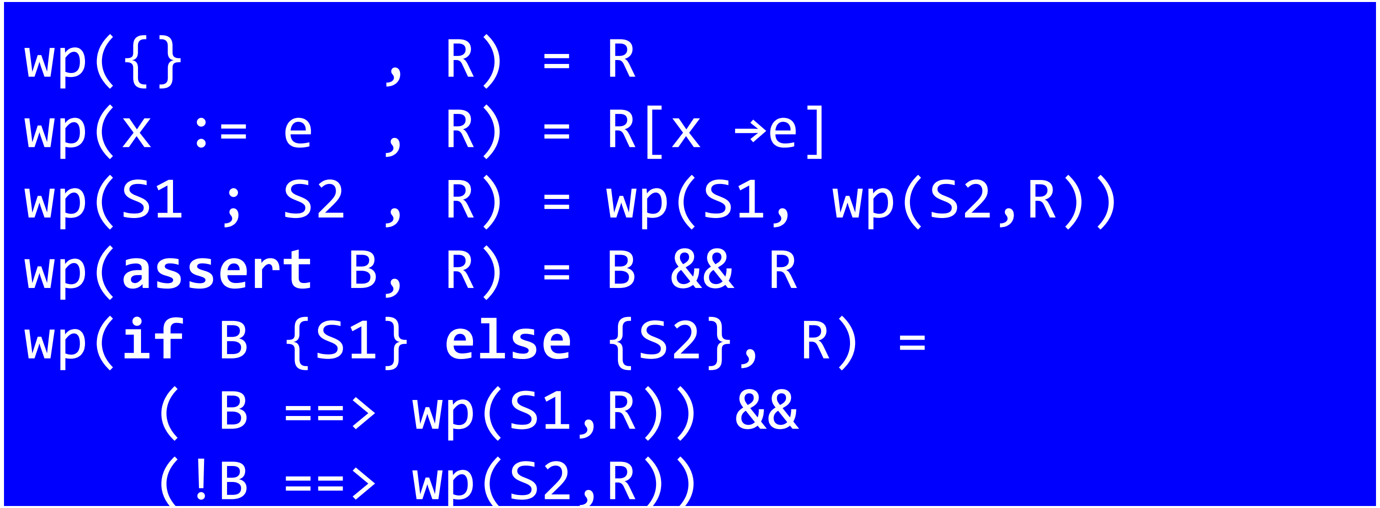
This logical formula is also called the *Verification condition*

SMT Solver (Z3)

Yes, that formula is true, and hence program is correct!

Not true or I don’t know, need more info

Weakest precondition calculus (no loops)



```
wp({}        , R) = R
wp(x := e    , R) = R[x →e]
wp(S1 ; S2 , R) = wp(S1, wp(S2,R))
wp(assert B, R) = B && R
wp(if B {S1} else {S2}, R) =
    ( B ==> wp(S1,R)) &&
    (!B ==> wp(S2,R))
```

Weakest precondition calculus (loops)

`wp(while B { S }, R) =?`

not **computable!**

No algorithm ***can*** exist that always computes `wp(while B { S }, R)` correctly!

Need: loop invariants & variants

Weakest precondition calculus (loops)

Loop guard

Loop Invariant

Decreases expression

Statements

Post-condition

```
wp(while B I D S, R) =
   I
   && (B && I ==> wp(S,I))
   && (!B && I ==> R)

   && (I ==> D >= 0)
   && (B && I ==>
      wp(tmp := D ; S, tmp > D))
```

Partial correctness

Invariant holds before loop

If invariant holds before loop, then it also hold afterwards

The failure of the loop guard (!B) and the invariant imply the postcondition R

Termination

Decreases expression is always >= 0

Decreases expression decreases each iteration

Total correctness = partial correctness + termination

Testing debugging & Verification

How do we get some certainty that that your program does what it’s supposed to?

- Testing: Try out inputs, does what you want?
    - Input space partitioning
    - How do we know we have enough tests? Coverage criteria, mutation testing
    - Property based (trade man power for compute power)
- Debugging: what to do when things go wrong
    - 7 steps
    - Minimize example
    - Reason backwards
- Formal specification & Verification
    - Prove that there are no bugs
    - Express specification using logic
    - How do we check that: Weakest precondition calculus

Testing debugging & Verification

- Lecture material (suggested additional readings when needed)
- Practice all exercises, labs
- Practice and go through sample exams

All the best for your exam !!