CSE 403

Software Engineering

Winter 2023

Program analysis

This week: Program analysis

- Reasoning about programs (Today)
- Statistical fault localization (Wednesday)
- In-class exercise (Friday)

Recap: Delta Debugging

Recap: Delta Debugging

Step	Test case									
1	$\Delta_1 = \nabla_2$	1	2	3	4	.	.	.	.	Testing Δ_1, Δ_2
2	$\Delta_2 = \nabla_1$	.	.	.	.	5	6	7	8	$\Rightarrow$ Increase granularity
3	Δ_1	1	2	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_4$
4	Δ_2	.	.	3	4	.	.	.	.	
5	Δ_3	.	.	.	.	5	6	.	.	
6	Δ_4	.	.	.	.	.	.	7	8	
7	∇_1	.	.	3	4	5	6	7	8	Testing complements
8	∇_2	1	2	.	.	5	6	7	8	$\Rightarrow$ Reduce to $c'_{✘} = \nabla_2$; continue with $n = 3$
9	Δ_1	1	2	.	.	.	.	.	.	Testing $\Delta_1, \Delta_2, \Delta_3$
10	Δ_2	.	.	.	.	5	6	.	.	* same *test* carried out in an earlier step
11	Δ_3	.	.	.	.	.	.	7	8	
12	∇_1	.	.	.	.	5	6	7	8	Testing complements
13	∇_2	1	2	.	.	.	.	7	8	$\Rightarrow$ Reduce to $c'_{✘} = \nabla_2$; continue with $n = 2$
14	$\Delta_1 = \nabla_2$	1	2	.	.	.	.	.	.	Testing Δ_1, Δ_2
15	$\Delta_2 = \nabla_1$	.	.	.	.	.	.	7	8	$\Rightarrow$ Increase granularity
16	Δ_1	1	.	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_4$
17	Δ_2	.	2	.	.	.	.	.	.	
18	Δ_3	.	.	.	.	.	.	7	.	
19	Δ_4	.	.	.	.	.	.	.	8	
20	∇_1	.	2	.	.	.	.	7	8	Testing complements
21	∇_2	1	.	.	.	.	.	7	8	$\Rightarrow$ Reduce to $c'_{✘} = \nabla_2$; continue with $n = 3$
22	Δ_1	1	.	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_3$
23	Δ_2	.	.	.	.	.	.	7	.	
24	Δ_3	.	.	.	.	.	.	.	8	
25	∇_1	.	.	.	.	.	.	7	8	Testing complements
26	∇_2	1	.	.	.	.	.	.	8	
27	∇_3	1	.	.	.	.	.	7	.	Done
Result		1	.	.	.	.	.	7	8	

Subsets

Recap: Delta Debugging

Step	Test case									
1	$\Delta_1 = \nabla_2$	1	2	3	4	.	.	.	.	Testing Δ_1, Δ_2
2	$\Delta_2 = \nabla_1$	.	.	.	.	5	6	7	8	$\Rightarrow$ Increase granularity
3	Δ_1	1	2	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_4$
4	Δ_2	.	.	3	4	.	.	.	.	
5	Δ_3	.	.	.	.	5	6	.	.	
6	Δ_4	.	.	.	.	.	.	7	8	
7	∇_1	.	.	3	4	5	6	7	8	Testing complements
8	∇_2	1	2	.	.	5	6	7	8	$\Rightarrow$ Reduce to $c'_{\times} = \nabla_2$; continue with $n = 3$
9	Δ_1	1	2	.	.	.	.	.	.	Testing Δ_1, Δ_2, Δ_3
10	Δ_2	.	.	.	.	5	6	.	.	* same *test* carried out in an earlier step
11	Δ_3	.	.	.	.	.	.	7	8	
12	∇_1	.	.	.	.	5	6	7	8	Testing complements
13	∇_2	1	2	.	.	.	.	7	8	$\Rightarrow$ Reduce to $c'_{\times} = \nabla_2$; continue with $n = 2$
14	$\Delta_1 = \nabla_2$	1	2	.	.	.	.	.	.	Testing Δ_1, Δ_2
15	$\Delta_2 = \nabla_1$	.	.	.	.	.	.	7	8	$\Rightarrow$ Increase granularity
16	Δ_1	1	.	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_4$
17	Δ_2	.	2	.	.	.	.	.	.	
18	Δ_3	.	.	.	.	.	.	7	.	
19	Δ_4	.	.	.	.	.	.	.	8	
20	∇_1	.	2	.	.	.	.	7	8	Testing complements
21	∇_2	1	.	.	.	.	.	7	8	$\Rightarrow$ Reduce to $c'_{\times} = \nabla_2$; continue with $n = 3$
22	Δ_1	1	.	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_3$
23	Δ_2	.	.	.	.	.	.	7	.	
24	Δ_3	.	.	.	.	.	.	.	8	
25	∇_1	.	.	.	.	.	.	7	8	Testing complements
26	∇_2	1	.	.	.	.	.	.	8	
27	∇_3	1	.	.	.	.	.	7	.	Done
Result		1	.	.	.	.	.	7	8	

Granularity

Recap: Delta Debugging

Step	Test case									
1	$\Delta_1 = \nabla_2$	1	2	3	4	.	.	.	.	Testing Δ_1, Δ_2
2	$\Delta_2 = \nabla_1$	.	.	.	.	5	6	7	8	$\Rightarrow$ Increase granularity
3	Δ_1	1	2	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_4$
4	Δ_2	.	.	3	4	.	.	.	.	
5	Δ_3	.	.	.	.	5	6	.	.	
6	Δ_4	.	.	.	.	.	.	7	8	
7	∇_1	.	.	3	4	5	6	7	8	Testing complements
8	∇_2	1	2	.	.	5	6	7	8	$\Rightarrow$ Reduce to $c'_{\mathsf{x}} = \nabla_2$; continue with $n = 3$
9	Δ_1	1	2	.	.	.	.	.	.	Testing $\Delta_1, \Delta_2, \Delta_3$
10	Δ_2	.	.	.	.	5	6	.	.	* same *test* carried out in an earlier step
11	Δ_3	.	.	.	.	.	.	7	8	
12	∇_1	.	.	.	.	5	6	7	8	Testing complements
13	∇_2	1	2	.	.	.	.	7	8	$\Rightarrow$ Reduce to $c'_{\mathsf{x}} = \nabla_2$; continue with $n = 2$
14	$\Delta_1 = \nabla_2$	1	2	.	.	.	.	.	.	Testing Δ_1, Δ_2
15	$\Delta_2 = \nabla_1$	.	.	.	.	.	.	7	8	$\Rightarrow$ Increase granularity
16	Δ_1	1	.	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_4$
17	Δ_2	.	2	.	.	.	.	.	.	
18	Δ_3	.	.	.	.	.	.	7	.	
19	Δ_4	.	.	.	.	.	.	.	8	
20	∇_1	.	2	.	.	.	.	7	8	Testing complements
21	∇_2	1	.	.	.	.	.	7	8	$\Rightarrow$ Reduce to $c'_{\mathsf{x}} = \nabla_2$; continue with $n = 3$
22	Δ_1	1	.	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_3$
23	Δ_2	.	.	.	.	.	.	7	.	
24	Δ_3	.	.	.	.	.	.	.	8	
25	∇_1	.	.	.	.	.	.	7	8	Testing complements
26	∇_2	1	.	.	.	.	.	.	8	
27	∇_3	1	.	.	.	.	.	7	.	Done
Result		1	.	.	.	.	.	7	8	

Complements

Recap: Delta Debugging

Step	Test case									
1	$\Delta_1 = \nabla_2$	1	2	3	4	.	.	.	.	Testing Δ_1, Δ_2
2	$\Delta_2 = \nabla_1$	.	.	.	.	5	6	7	8	$\Rightarrow$ Increase granularity
3	Δ_1	1	2	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_4$
4	Δ_2	.	.	3	4	.	.	.	.	
5	Δ_3	.	.	.	.	5	6	.	.	
6	Δ_4	.	.	.	.	.	.	7	8	
7	∇_1	.	.	3	4	5	6	7	8	Testing complements
8	∇_2	1	2	.	.	5	6	7	8	$\Rightarrow$ Reduce to $c'_{\times} = \nabla_2$; continue with $n = 3$
9	Δ_1	1	2	.	.	.	.	.	.	Testing $\Delta_1, \Delta_2, \Delta_3$
10	Δ_2	.	.	.	.	5	6	.	.	* same *test* carried out in an earlier step
11	Δ_3	.	.	.	.	.	.	7	8	
12	∇_1	.	.	.	.	5	6	7	8	Testing complements
13	∇_2	1	2	.	.	.	.	7	8	$\Rightarrow$ Reduce to $c'_{\times} = \nabla_2$; continue with $n = 2$
14	$\Delta_1 = \nabla_2$	1	2	.	.	.	.	.	.	Testing Δ_1, Δ_2
15	$\Delta_2 = \nabla_1$	.	.	.	.	.	.	7	8	$\Rightarrow$ Increase granularity
16	Δ_1	1	.	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_4$
17	Δ_2	.	2	.	.	.	.	.	.	
18	Δ_3	.	.	.	.	.	.	7	.	
19	Δ_4	.	.	.	.	.	.	.	8	
20	∇_1	.	2	.	.	.	.	7	8	Testing complements
21	∇_2	1	.	.	.	.	.	7	8	$\Rightarrow$ Reduce to $c'_{\times} = \nabla_2$; continue with $n = 3$
22	Δ_1	1	.	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_3$
23	Δ_2	.	.	.	.	.	.	7	.	
24	Δ_3	.	.	.	.	.	.	.	8	
25	∇_1	.	.	.	.	.	.	7	8	Testing complements
26	∇_2	1	.	.	.	.	.	.	8	
27	∇_3	1	.	.	.	.	.	7	.	Done
Result		1	.	.	.	.	.	7	8	

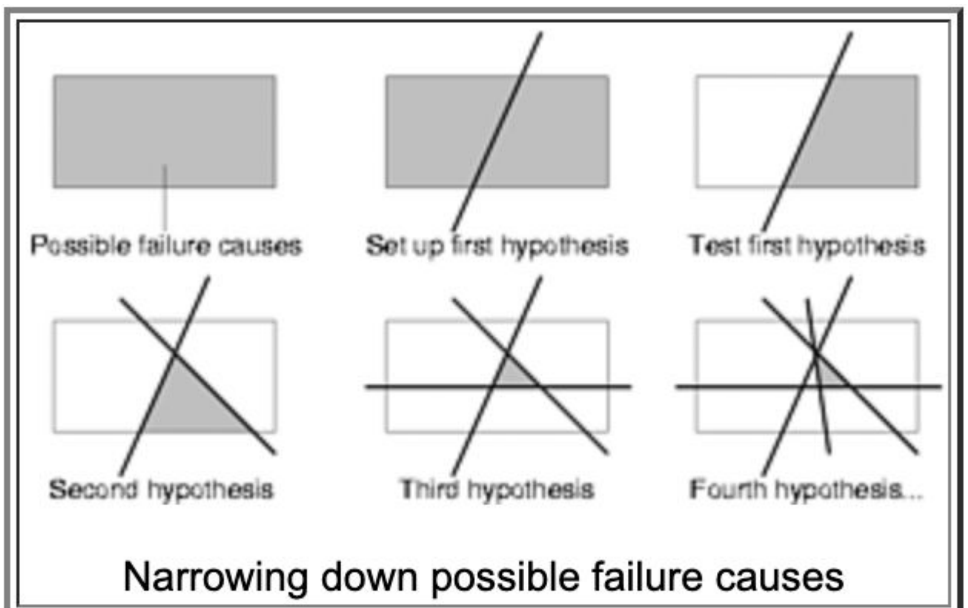
Reduce

Recap: Delta Debugging

Step	Test case									
1	$\Delta_1 = \nabla_2$	1	2	3	4	.	.	.	.	Testing Δ_1, Δ_2
2	$\Delta_2 = \nabla_1$	.	.	.	.	5	6	7	8	$\Rightarrow$ Increase granularity
3	Δ_1	1	2	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_4$
4	Δ_2	.	.	3	4	.	.	.	.	
5	Δ_3	.	.	.	.	5	6	.	.	
6	Δ_4	.	.	.	.	.	.	7	8	
7	∇_1	.	.	3	4	5	6	7	8	Testing complements
8	∇_2	1	2	.	.	5	6	7	8	$\Rightarrow$ Reduce to $c'_{\text{✘}} = \nabla_2$; continue with $n = 3$
9	Δ_1	1	2	.	.	.	.	.	.	Testing $\Delta_1, \Delta_2, \Delta_3$
10	Δ_2	.	.	.	.	5	6	.	.	* same *test* carried out in an earlier step
11	Δ_3	.	.	.	.	.	.	7	8	
12	∇_1	.	.	.	.	5	6	7	8	Testing complements
13	∇_2	1	2	.	.	.	.	7	8	$\Rightarrow$ Reduce to $c'_{\text{✘}} = \nabla_2$; continue with $n = 2$
14	$\Delta_1 = \nabla_2$	1	2	.	.	.	.	.	.	Testing Δ_1, Δ_2
15	$\Delta_2 = \nabla_1$	.	.	.	.	.	.	7	8	$\Rightarrow$ Increase granularity
16	Δ_1	1	.	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_4$
17	Δ_2	.	2	.	.	.	.	.	.	
18	Δ_3	.	.	.	.	.	.	7	.	
19	Δ_4	.	.	.	.	.	.	.	8	
20	∇_1	.	2	.	.	.	.	7	8	Testing complements
21	∇_2	1	.	.	.	.	.	7	8	$\Rightarrow$ Reduce to $c'_{\text{✘}} = \nabla_2$; continue with $n = 3$
22	Δ_1	1	.	.	.	.	.	.	.	Testing $\Delta_1, \ldots, \Delta_3$
23	Δ_2	.	.	.	.	.	.	7	.	
24	Δ_3	.	.	.	.	.	.	.	8	
25	∇_1	.	.	.	.	.	.	7	8	Testing complements
26	∇_2	1	.	.	.	.	.	.	8	
27	∇_3	1	.	.	.	.	.	7	.	Done
Result		1	.	.	.	.	.	7	8	

Delta debugging: discussion

- Applicability: Is this useful (as a concept and/or automated tool)?
- Optimality: minimal vs. minimum test case.
- Complexity: Best-case vs. worst-case.
- Assumptions: monotonicity and determinism.



Narrowing down possible failure causes

https://www.st.cs.uni-saarland.de/dd/

Reasoning about programs

Reasoning about programs

Use cases

- Verification/testing: ensure code is correct
- Prove facts to be true, e.g.:
  - x is never null
  - y is always greater than 0
  - input array a is sorted
- Debugging: understand why code is incorrect

Reasoning about programs

Use cases

- Verification/testing: ensure code is correct
- Prove facts to be true, e.g.:
  - x is never null
  - y is always greater than 0
  - input array a is sorted
- Debugging: understand why code is incorrect

Approaches

- Testing (403)
- (Delta) Debugging (403)
- Fault localization (403)
- Abstract interpretation (primer in 403, covered in 503)
- Theorem proving (primer in 403, covered in 507)
- ...

Forward vs. backward reasoning

Forward reasoning

- Knowing: **a fact** that is **true before execution**.
- Reasoning: **what must be true after execution**.
- Given a precondition, what postcondition(s) are true?

Forward vs. backward reasoning

Forward reasoning

- Knowing: **a fact** that is **true before execution**.
- Reasoning: **what must be true after execution**.
- Given a precondition, what postcondition(s) are true?

Backward reasoning

- Knowing: **a fact** that is **true after execution**.
- Reasoning: **what must have been true before execution**.
- Given a postcondition, what precondition(s) must hold?

What are the pros and cons for each approach?

Forward vs. backward reasoning

Forward reasoning

- More intuitive for most people
- Helps understand what will happen (simulates the code)
- Introduces facts that may be irrelevant to the goal
- Set of current facts may get large
- Takes longer to realize that the task is hopeless

Backward reasoning

- Usually more helpful
- Helps understand what should happen
- Given a specific goal, indicates how to achieve it
- Given an error, gives a test case that exposes it

Pre/Post-conditions and Invariants

Terminology

Pre-condition (to a function)

- A condition that must be true when entering (the function)
- May include expectations about the arguments

Post-condition (to a function)

- A condition that must be true when leaving (the function)

Terminology

Pre-condition (to a function)

- A condition that must be true when entering (the function)
- May include expectations about the arguments

Post-condition (to a function)

- A condition that must be true when leaving (the function)

Loop invariant

- A condition that must be true for every loop iteration
- Must be true at the beginning and end of the loop body

Terminology

Pre-condition (to a function)

- A condition that must be true when entering (the function)
- May include expectations about the arguments

Post-condition (to a function)

- A condition that must be true when leaving (the function)

Loop invariant

- A condition that must be true for every loop iteration
- Must be true at the beginning and end of the loop body

Pre-conditions define execution validity. Post-conditions and loop invariants define expected properties of a correct implementation, given a valid execution.

Pre-conditions and post-conditions


```
double avgAbs(double[] nums) {
 int n = nums.length;
 double sum = 0;

 int i = 0;
 while (i != n) {
   if(nums[i]>0) {
     sum = sum + nums[i];
   else {
     sum = sum - nums[i];
   }
   i = i + 1;
 }

 return sum / n;
}
```

Entry point

Exit point

What are pre-conditions and post-conditions of this method (at the entry and exit points)?

Pre-conditions and post-conditions

```
double avgAbs(double[] nums) {
 int n = nums.length;
 double sum = 0;

 int i = 0;
 while (i != n) {
   if(nums[i]>0) {
    sum = sum + nums[i];
   else {
    sum = sum - nums[i];
   }
   i = i + 1;
 }

 return sum / n;
}
```

Pre-conditions

- nums is not null
- `nums.length > 0`

Post-conditions

- nums has not changed
- `n > 0`
- `sum >= 0`
- `return value >= 0`
- ...

(Loop) invariants


```
double avgAbs(double[] nums) {
  int n = nums.length;
  double sum = 0;

  int i = 0;
  while (i != n) {
    if(nums[i]>0) {
      sum = sum + nums[i];
    else {
      sum = sum - nums[i];
    }
    i = i + 1;
  }

  return sum / n;
}
```

Does this loop terminate?
What are pre-conditions,
post-conditions,
and loop invariants?

Summary

Pre-condition (to a function)

- A condition that must be true when entering (the function)
- May include expectations about the arguments

Post-condition (to a function)

- A condition that must be true when leaving (the function)

Loop invariant

- A condition that must be true for every loop iteration
- Must be true at the beginning and end of the loop body

How are these related to software testing and debugging?

Dynamic vs. static analysis

Dynamic vs. static analysis: overview

Dynamic analysis

- Reason about the program based on some program executions.
- Observe concrete behavior at run time.
- Improve confidence in correctness.

Dynamic vs. static analysis: overview

Dynamic analysis

- Reason about the program based on some program executions.
- Observe concrete behavior at run time.
- Improve confidence in correctness.

Static analysis

- Reason about the program without executing it.
- Build an abstraction of run-time states.
- Reason over abstract domain.
- Prove a property of the program.

Soundly approximate program behavior.

Dynamic vs. static analysis: overview

Dynamic analysis

- Reason about the program based on some program executions.
- Observe concrete behavior at run time.
- Improve confidence in correctness.

Static analysis

- Reason about the program without executing it.
- Build an abstraction of run-time states.
- Reason over abstract domain.
- Prove a property of the program.

[y:=2, x:=2]

y = x++

[y:=2, x:=3]

Dynamic vs. static analysis: overview

Dynamic analysis

- Reason about the program based on some program executions.
- Observe concrete behavior at run time.
- Improve confidence in correctness.

Static analysis

- Reason about the program without executing it.
- Build an abstraction of run-time states.
- Reason over abstract domain.
- Prove a property of the program.

<y is even, x is even>

y = x++

<y is even, x is odd>

Dynamic vs. static analysis: overview

Dynamic analysis

- Reason about the program based on some program executions.
- Observe concrete behavior at run time.
- Improve confidence in correctness.

Static analysis

- Reason about the program without executing it.
- Build an abstraction of run-time states.
- Reason over abstract domain.
- Prove a property of the program.

The statement "**f returns a non-negative value**" is weaker (but easier to establish) than the statement "**f returns the absolute value of its argument**".

Dynamic analysis: examples

Software testing

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;

  int i = 0;
  while (i<n)
    sum = sum + nums[i];
    i = i + 1;

  double avg = sum / n;

  return avg;
}
```

A test for the avg function:

```
@Test
public void testAvg() {
  double nums =
       new double[]{1.0, 2.0, 3.0});
  double actual = Math.avg(nums);
  double expected = 2.0;
  assertEquals(expected,actual,EPS);
}
```

Static analysis: examples

```
static OSStatus
SSLVerifySignedServerKeyExchange(...) {
        OSStatus err;
        ...
        if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        err = sslRawVerify(ctx, ctx->peerPubKey, dataToSign, dataToSignLen, signature, signatureLen);
        if(err) {
                sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify returned %d\n", (int)err);
                goto fail;
        }
        fail:
                SSLFreeBuffer(&signedHashes);
                SSLFreeBuffer(&hashCtx);
                return err;
}
```

Anything wrong with this code?

Static analysis: examples

Apple's "goto fail" bug: a security vulnerability for 2 years!

```
static OSStatus
SSLVerifySignedServerKeyExchange(...) {
    OSStatus err;
    ...
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    err = sslRawVerify(ctx, ctx->peerPubKey, dataToSign, dataToSignLen, signature, signatureLen);
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify returned %d\n", (int)err);
        goto fail;
    }
    fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

Static analysis: examples

Rule/pattern-based analysis (PMD, Findbugs, Error Prone, etc.)

```java
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;

  int i = 0;
  while (i<n)
    sum = sum + nums[i];
    i = i + 1;

  double avg = sum / n;

  return avg;
}
```

```java
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;

  int i = 0;
  while (i<n) {
    sum = sum + nums[i];
    i = i + 1;
  }
  double avg = sum / n;

  return avg;
}
```

Static analysis: examples

Compiler: type checking

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;

  int i = 0.0;
  while (i<n) {
    sum = sum + nums[i];
    i = i + 1;
  }
  double avg = sum / n;

  return avg;
}
```

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;

  int i = 0;
  while (i<n) {
    sum = sum + nums[i];
    i = i + 1;
  }
  double avg = sum / n;

  return avg;
}
```

Dynamic vs. static analysis: summary

Dynamic analysis

- Concrete domain
- Does not generalize
- Slow if exhaustive

Static analysis

- Abstract domain
- Sound but imprecise
- Slow if precise