The Scientific Method

Andreas Zeller

1

How Google™ Tests Software

James A. Whittaker, Engineering Director

Thursday, May 27, 2010, 16:15–17:45
Saarland University, Campus E1 3, HS002

2

Google releases software many times every day. Ever wonder what it takes to test in such an environment? James Whittaker talks about test methodology, tools and innovation surrounding the discipline of quality assurance at Google where testers are far outnumbered by developers. Specifically he will present how the webapp-chrome-chromium stack is tested to ensure that Google apps work well on Chrome browser and Chromium operating system. During the talk he presents how Google

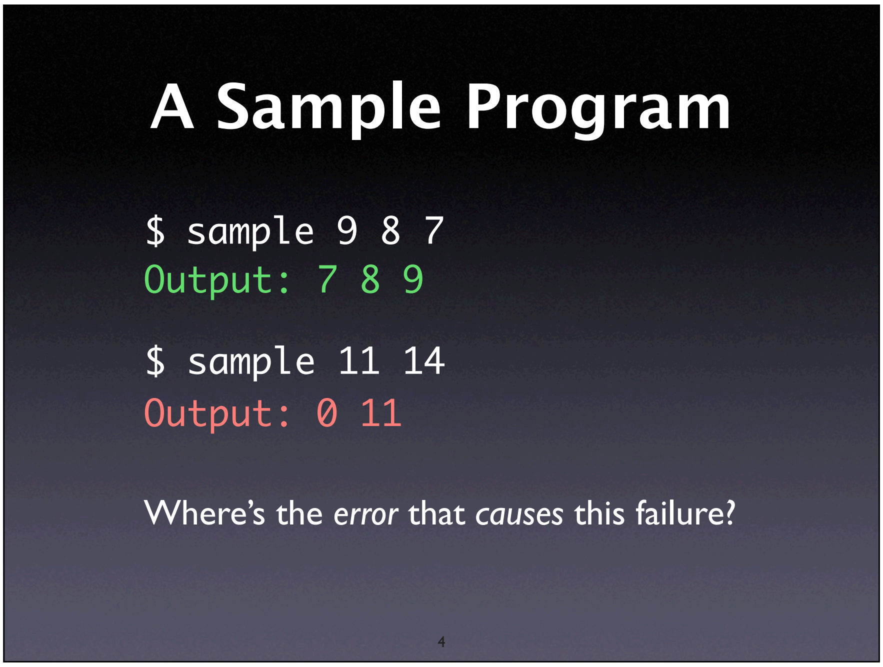
T · Mobile ·
2:47 PM
www.google.com: Google
Home Gmail Calendar Reader More
Google
hiking trails
Search
Web Results 1 - 10 of about 3,620,000 for hiking trails.
LocalHikes - Hikes and trails near U.S. Metropolitan Areas
Where to Hike: Top US Hiking Trails | GORP.com
MENU

3

Everything typed into T-Mobile G1 was taken as a shell command (i.e. “reboot”)

http://crave.cnet.co.uk/mobiles/0,39029453,49299782,00.htm

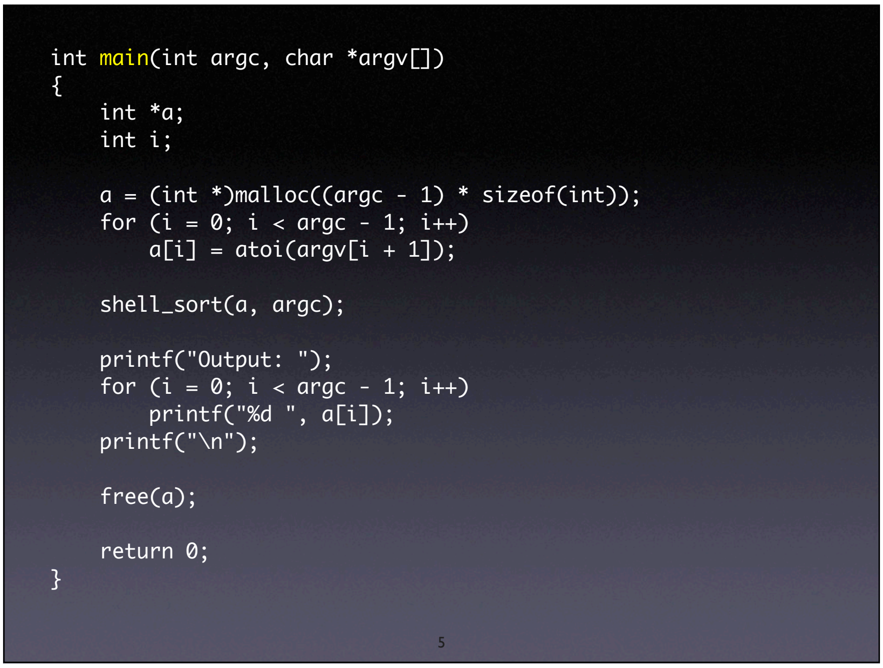
Recent **T-Mobile G1** update has caused a peculiar side-effect that's proving rather embarrassing for Google. RC29, as the update is known, causes certain text entered into the G1 to run commands.



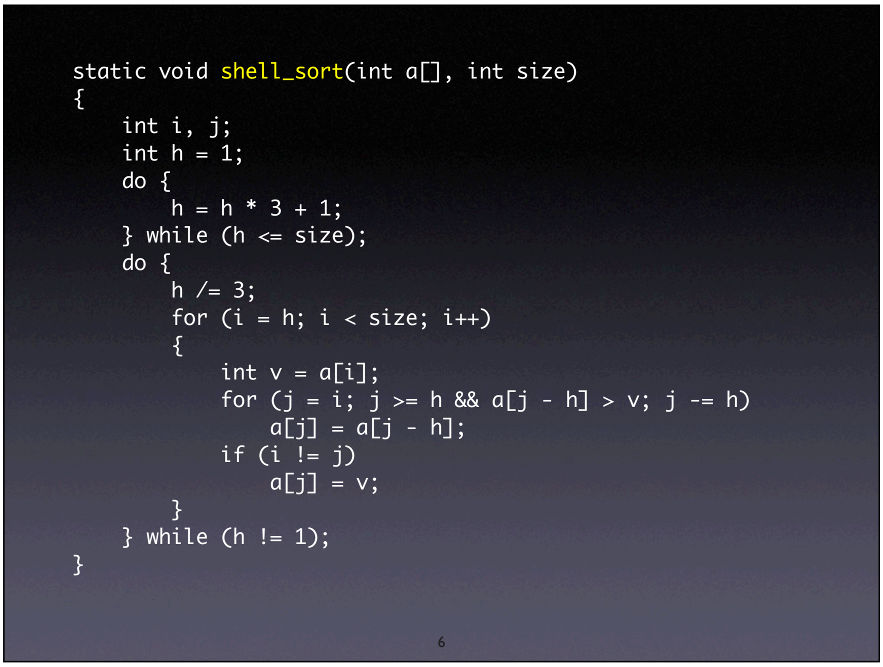
A Sample Program

```
$ sample 9 8 7
Output: 7 8 9

$ sample 11 14
Output: 0 11
```

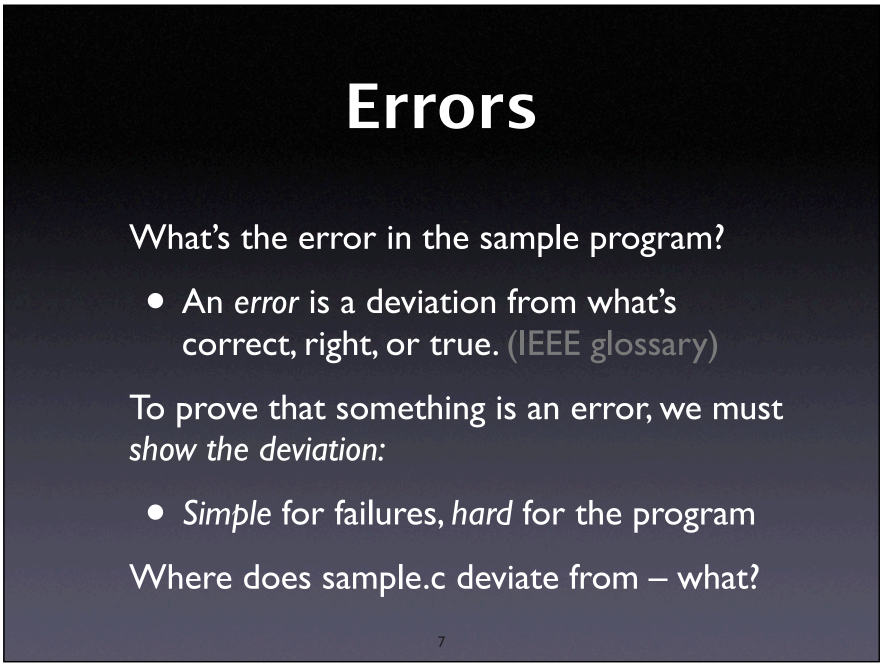
Where’s the *error* that *causes* this failure?



```c
int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);

    return 0;
}
```



```c
static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}
```



Errors

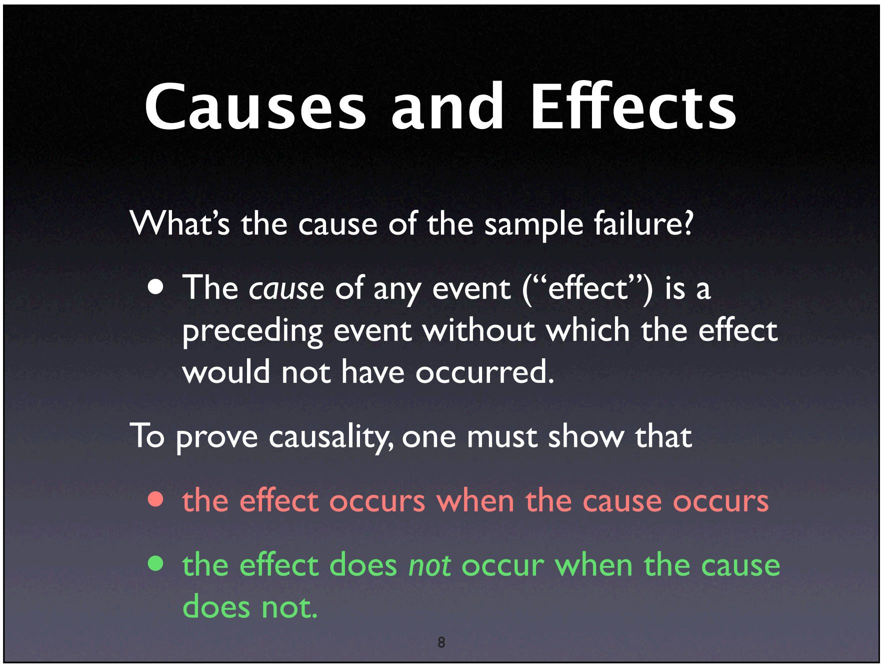
What's the error in the sample program?

- An *error* is a deviation from what's correct, right, or true. (IEEE glossary)

To prove that something is an error, we must *show the deviation:*

- *Simple* for failures, *hard* for the program

Where does sample.c deviate from – what?



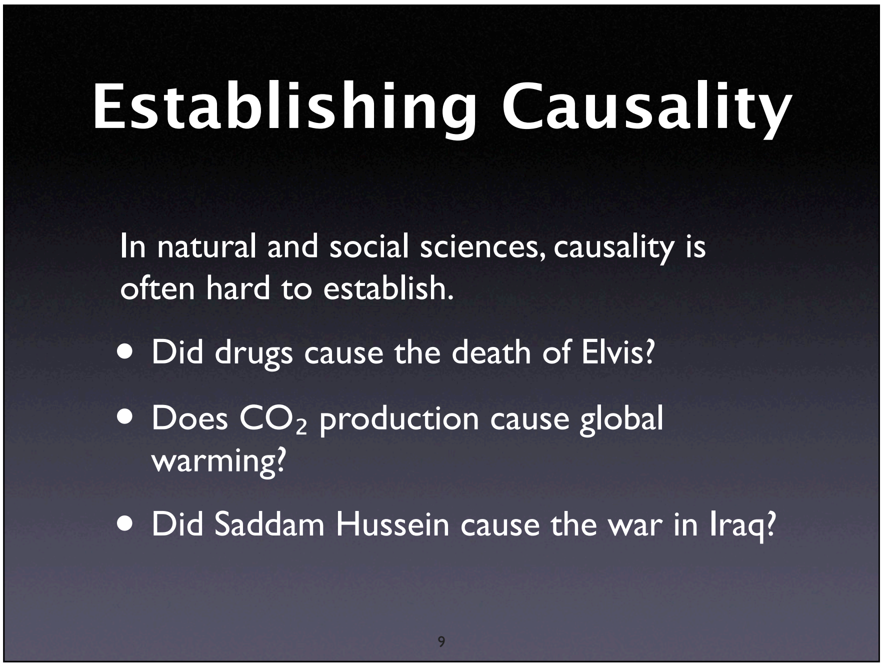
Causes and Effects

What's the cause of the sample failure?

- The *cause* of any event ("effect") is a preceding event without which the effect would not have occurred.

To prove causality, one must show that

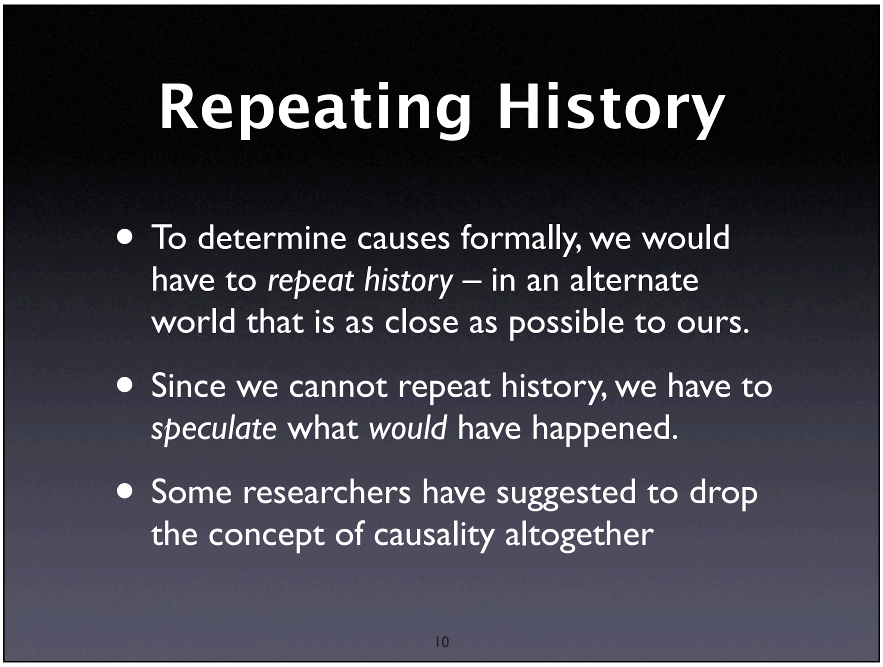
- the effect occurs when the cause occurs
- the effect does *not* occur when the cause does not.



Establishing Causality

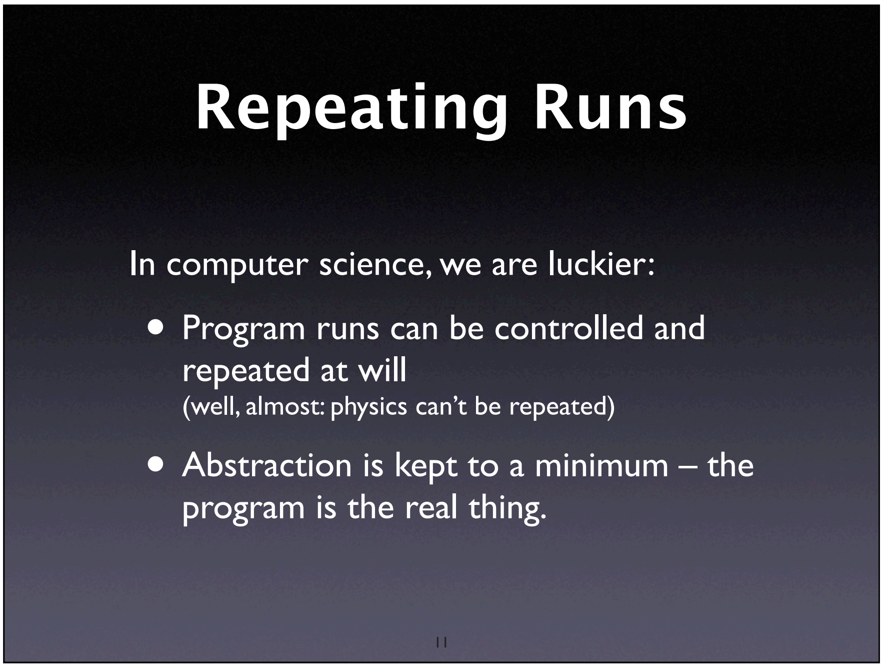
In natural and social sciences, causality is often hard to establish.

- Did drugs cause the death of Elvis?
- Does CO_2 production cause global warming?
- Did Saddam Hussein cause the war in Iraq?



Repeating History

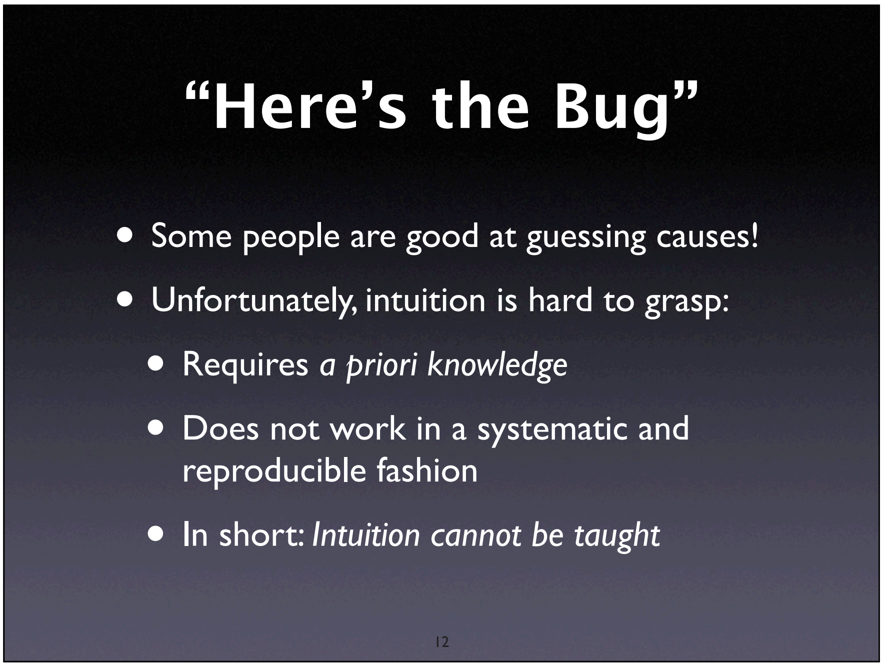
- To determine causes formally, we would have to *repeat history* – in an alternate world that is as close as possible to ours.
- Since we cannot repeat history, we have to *speculate* what *would* have happened.
- Some researchers have suggested to drop the concept of causality altogether



Repeating Runs

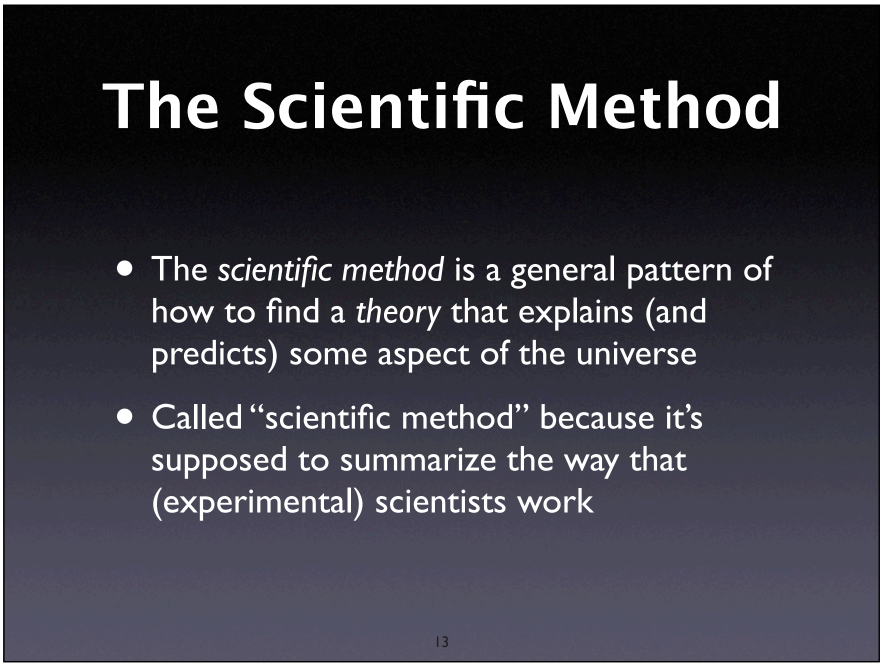
In computer science, we are luckier:

- Program runs can be controlled and repeated at will
  (well, almost: physics can’t be repeated)
- Abstraction is kept to a minimum – the program is the real thing.



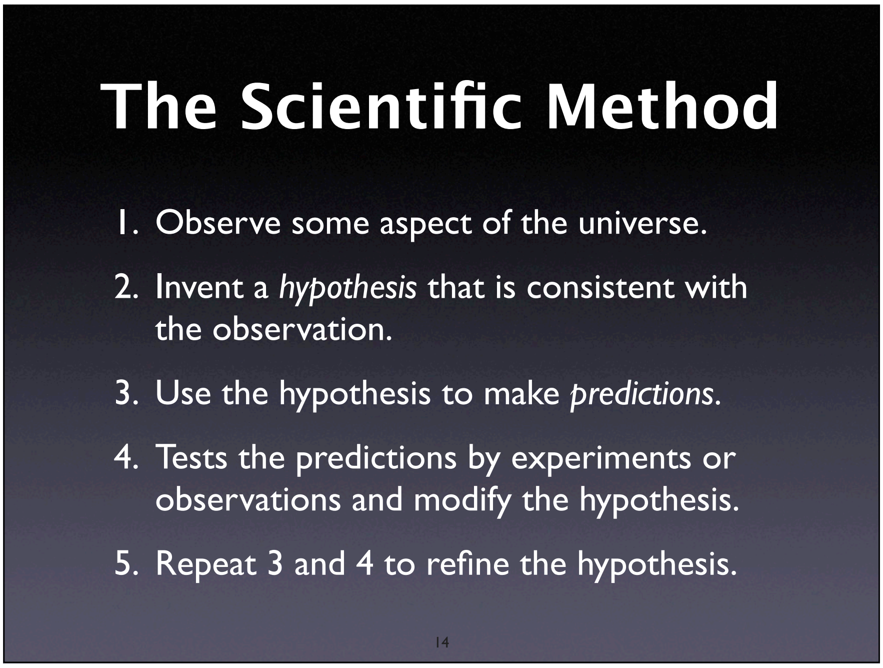
“Here’s the Bug”

- Some people are good at guessing causes!
- Unfortunately, intuition is hard to grasp:
  - Requires *a priori knowledge*
  - Does not work in a systematic and reproducible fashion
  - In short: *Intuition cannot be taught*



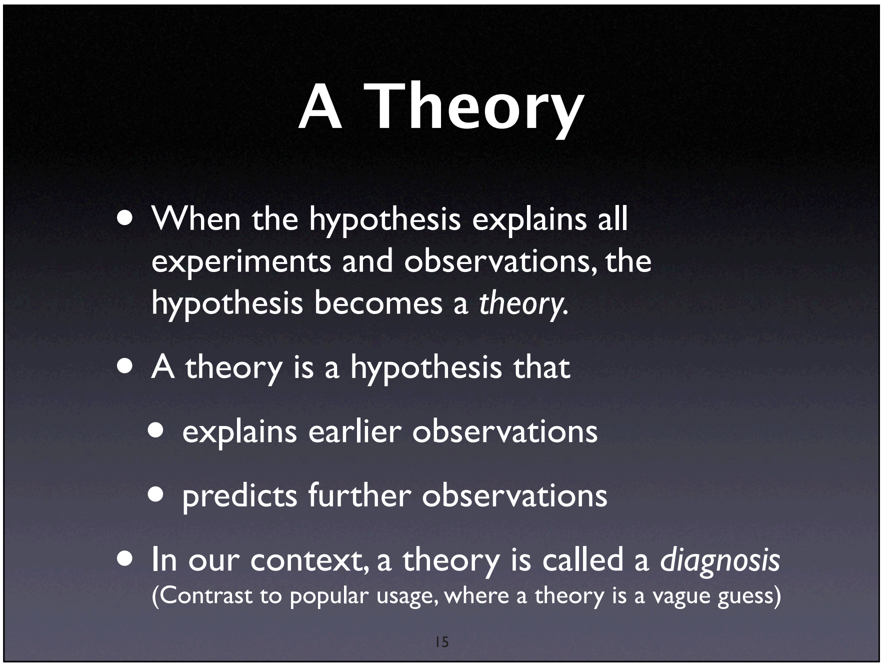
The Scientific Method

- The *scientific method* is a general pattern of how to find a *theory* that explains (and predicts) some aspect of the universe
- Called "scientific method" because it's supposed to summarize the way that (experimental) scientists work



The Scientific Method

1. Observe some aspect of the universe.
2. Invent a *hypothesis* that is consistent with the observation.
3. Use the hypothesis to make *predictions*.
4. Tests the predictions by experiments or observations and modify the hypothesis.
5. Repeat 3 and 4 to refine the hypothesis.

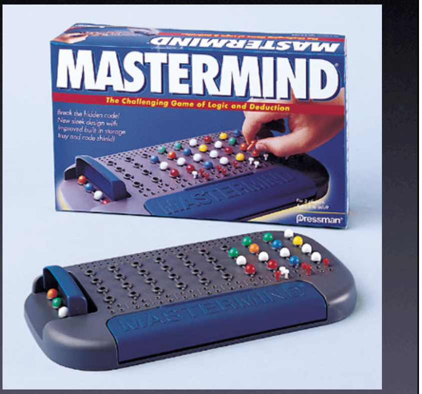


A Theory

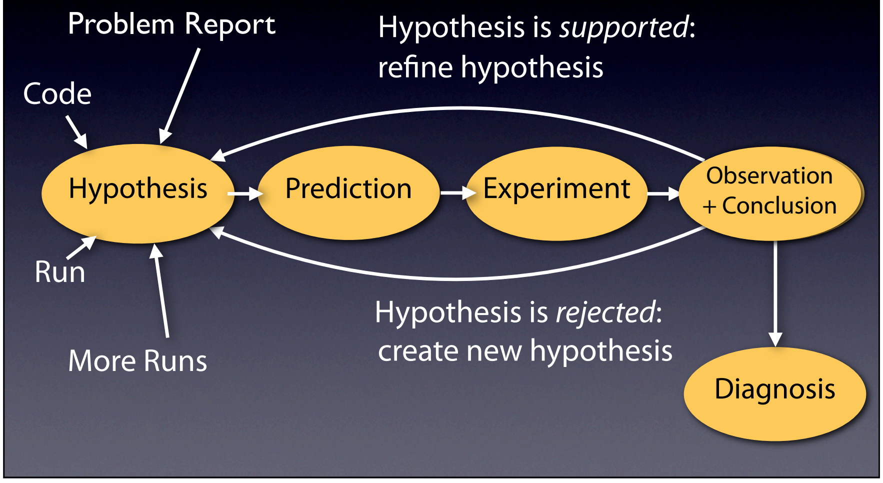
- When the hypothesis explains all experiments and observations, the hypothesis becomes a *theory.*
- A theory is a hypothesis that
  - explains earlier observations
  - predicts further observations
- In our context, a theory is called a *diagnosis*
  (Contrast to popular usage, where a theory is a vague guess)

Mastermind

- A Mastermind game is a typical example of applying the scientific method.
- Create hypotheses until the theory predicts the secret.



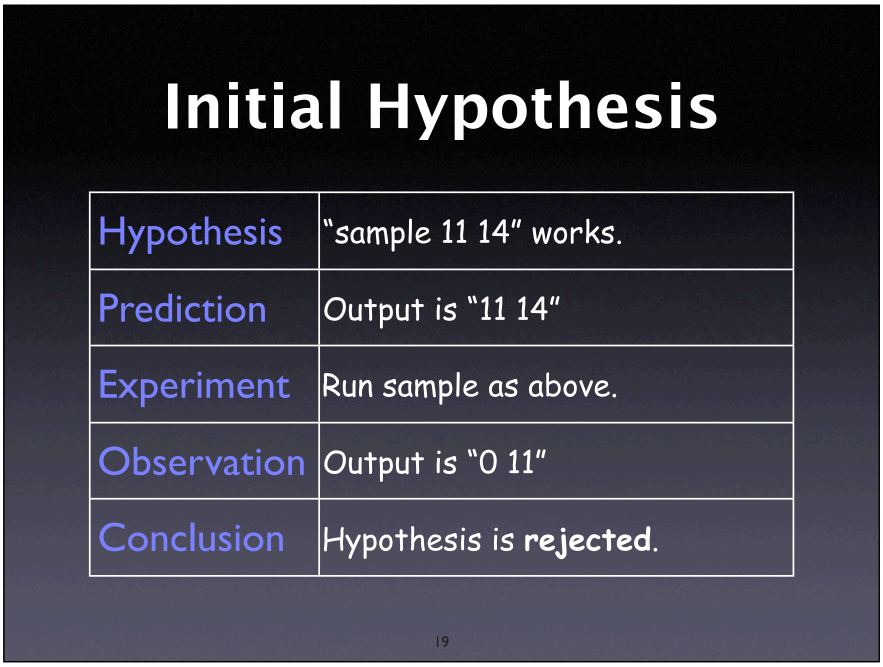
Scientific Method



A Sample Program

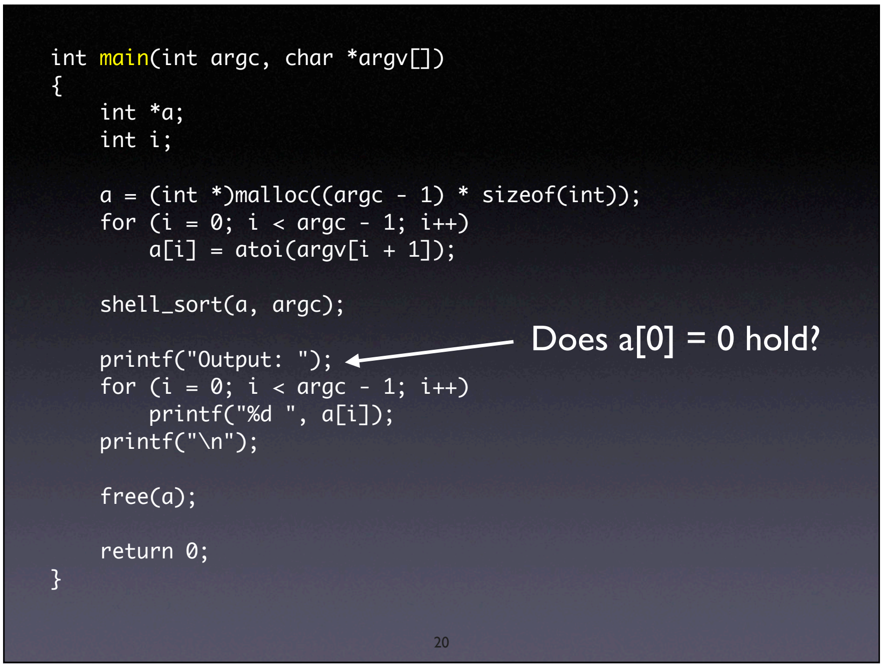
```
$ sample 9 8 7
Output: 7 8 9

$ sample 11 14
Output: 0 11
```

Let’s use the scientific method to debug this.



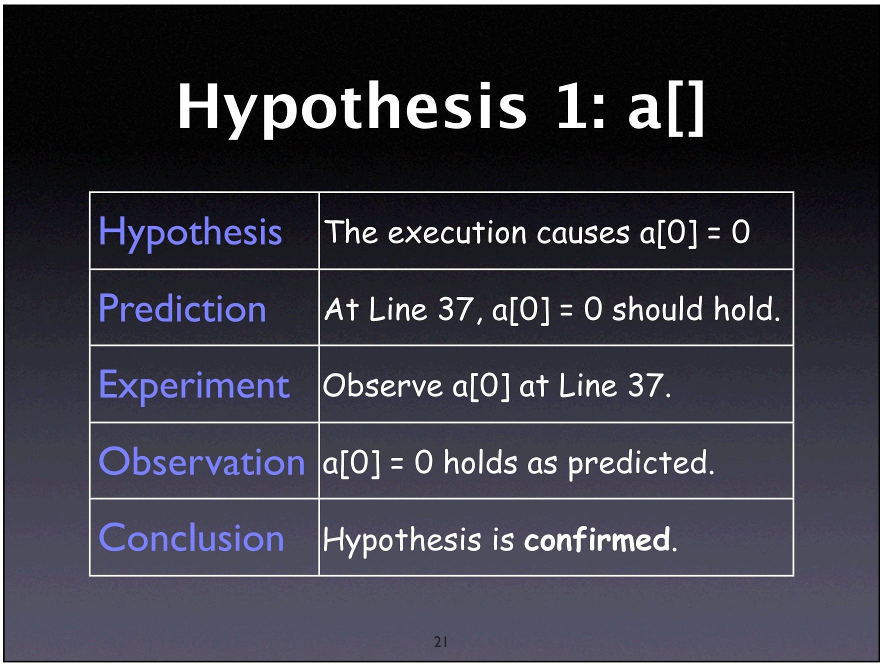
Initial Hypothesis

Hypothesis	"sample 11 14" works.
Prediction	Output is "11 14"
Experiment	Run sample as above.
Observation	Output is "0 11"
Conclusion	Hypothesis is **rejected**.



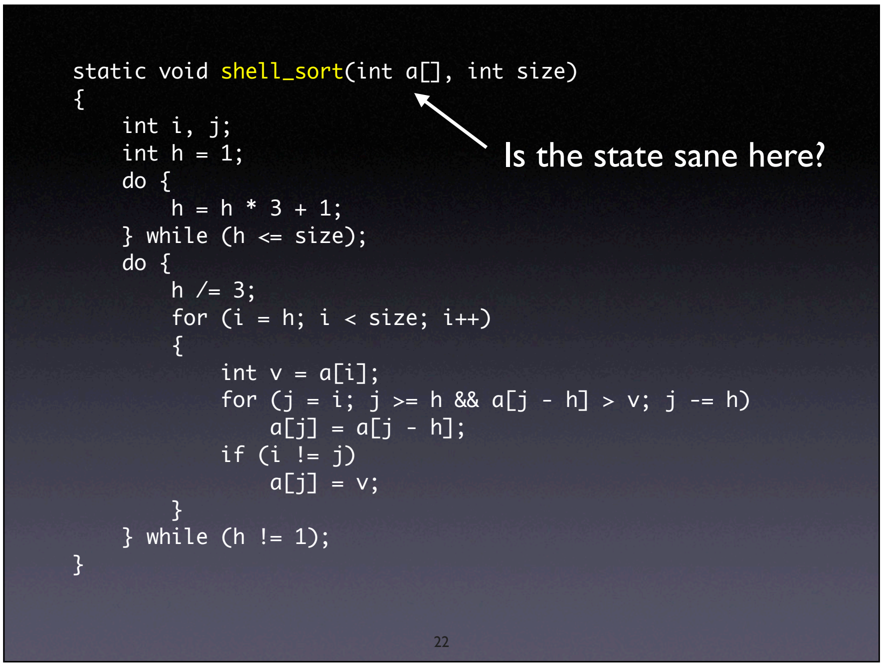
```
int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);

    return 0;
}
```

Does a[0] = 0 hold?



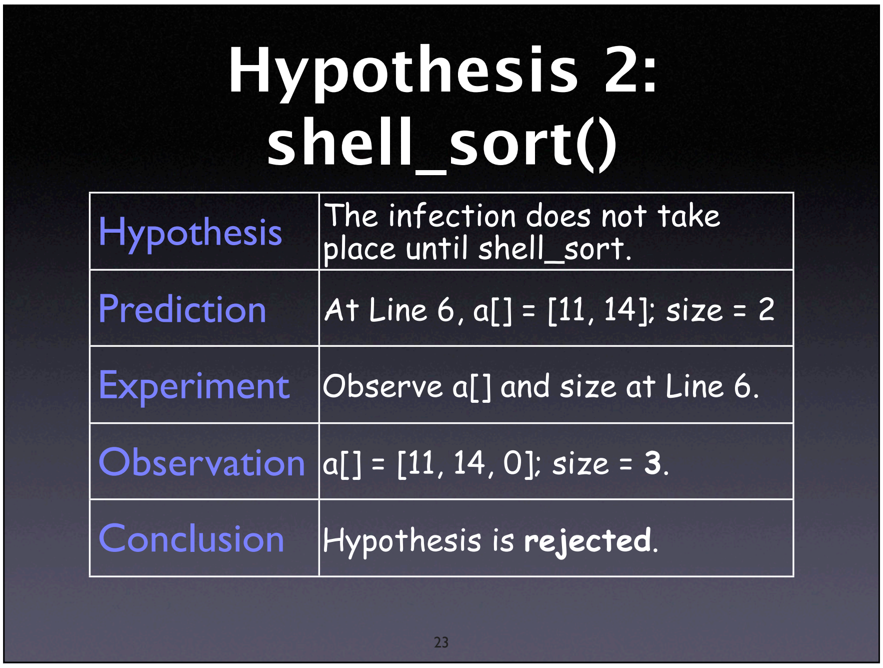
Hypothesis 1: a[]

Hypothesis	The execution causes a[0] = 0
Prediction	At Line 37, a[0] = 0 should hold.
Experiment	Observe a[0] at Line 37.
Observation	a[0] = 0 holds as predicted.
Conclusion	Hypothesis is **confirmed**.



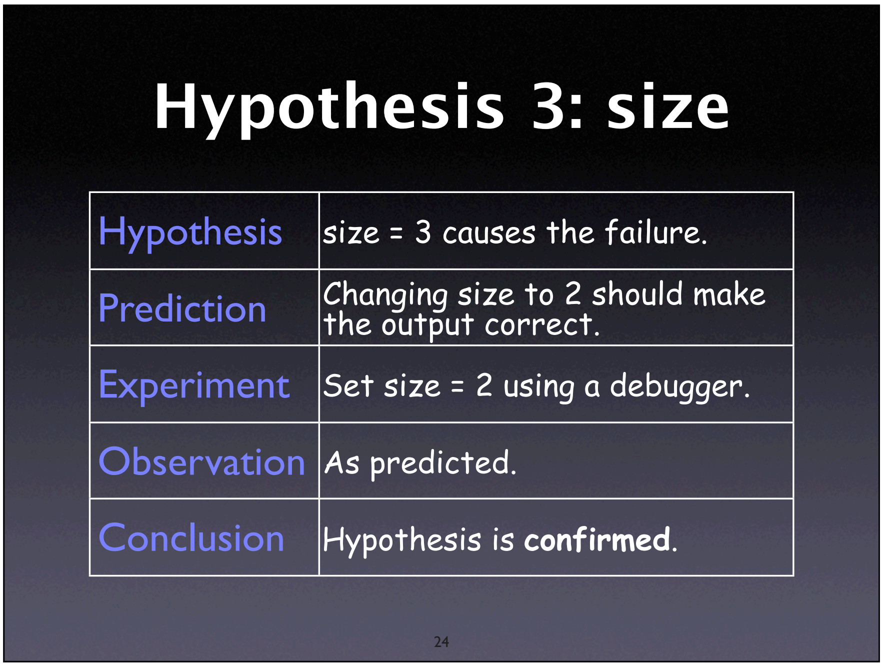
```
static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}
```

Is the state sane here?



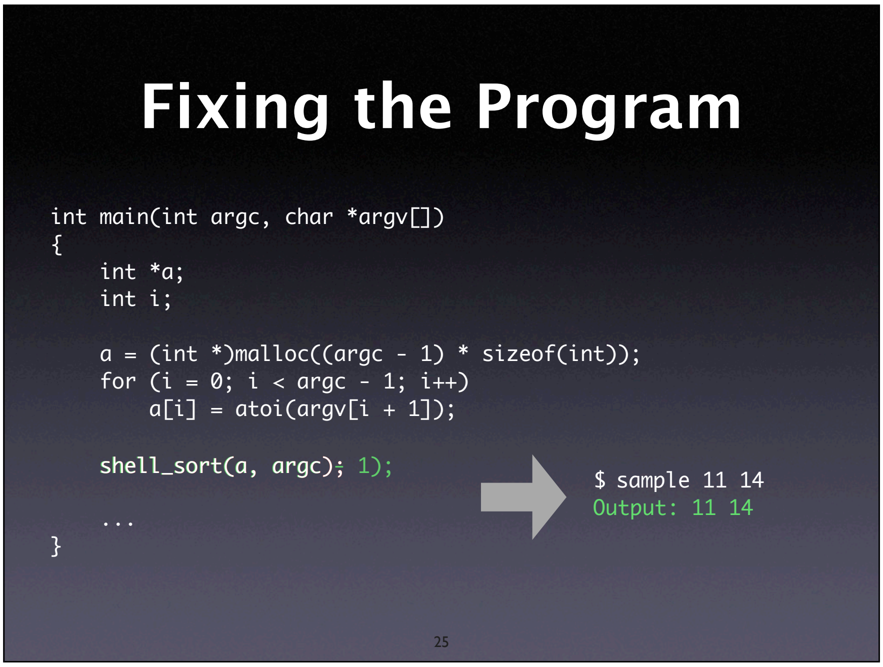
Hypothesis 2: shell_sort()

Hypothesis	The infection does not take place until shell_sort.
Prediction	At Line 6, a[] = [11, 14]; size = 2
Experiment	Observe a[] and size at Line 6.
Observation	a[] = [11, 14, 0]; size = **3**.
Conclusion	Hypothesis is **rejected**.



Hypothesis 3: size

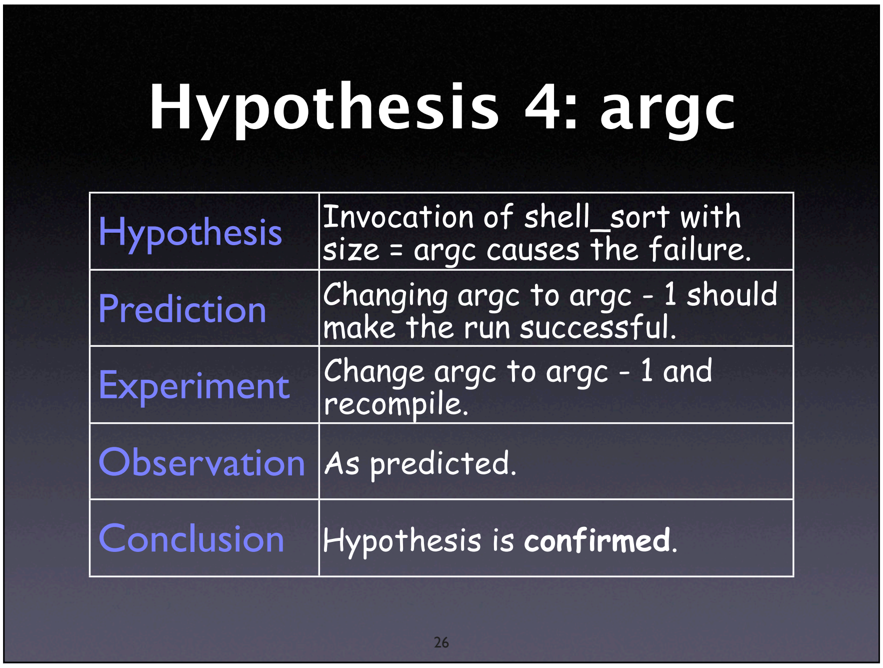
Hypothesis	size = 3 causes the failure.
Prediction	Changing size to 2 should make the output correct.
Experiment	Set size = 2 using a debugger.
Observation	As predicted.
Conclusion	Hypothesis is **confirmed**.



Fixing the Program

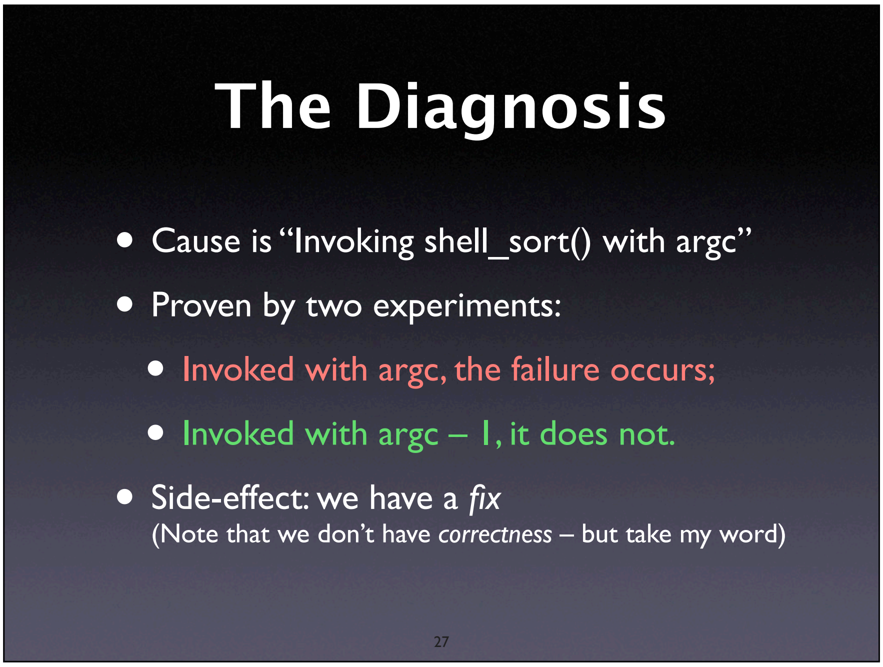
```
int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc)~~;~~ 1);

    ...
}
```

```
$ sample 11 14
Output: 11 14
```



Hypothesis 4: argc

Hypothesis	Invocation of shell_sort with size = argc causes the failure.
Prediction	Changing argc to argc - 1 should make the run successful.
Experiment	Change argc to argc - 1 and recompile.
Observation	As predicted.
Conclusion	Hypothesis is **confirmed**.



The Diagnosis

- Cause is "Invoking shell_sort() with argc"
- Proven by two experiments:
  - Invoked with argc, the failure occurs;
  - Invoked with argc – 1, it does not.
- Side-effect: we have a *fix*
  (Note that we don't have *correctness* – but take my word)

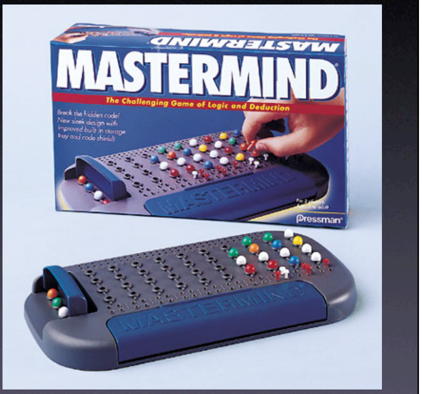
Explicit Debugging

- Being *explicit* is important to understand the problem.
- Just *stating* the problem can already solve it.

http://www.varsityclub.harvard.edu/Logos/teddy.gif

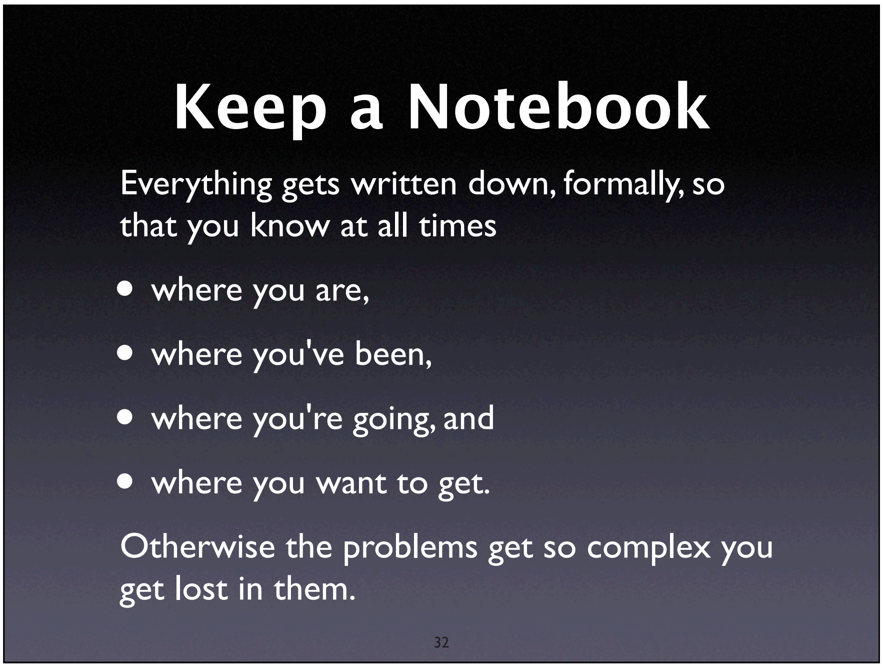
Keeping Track

- In a Mastermind game, *all* hypotheses and observations are explicit.
- Makes playing the game much easier.



Implicit Debugging

- Remember your last debugging session: Did you write down hypotheses and observations?
- Not being explicit forces you to keep all hypotheses and outcomes *in memory*
- Like playing Mastermind in memory

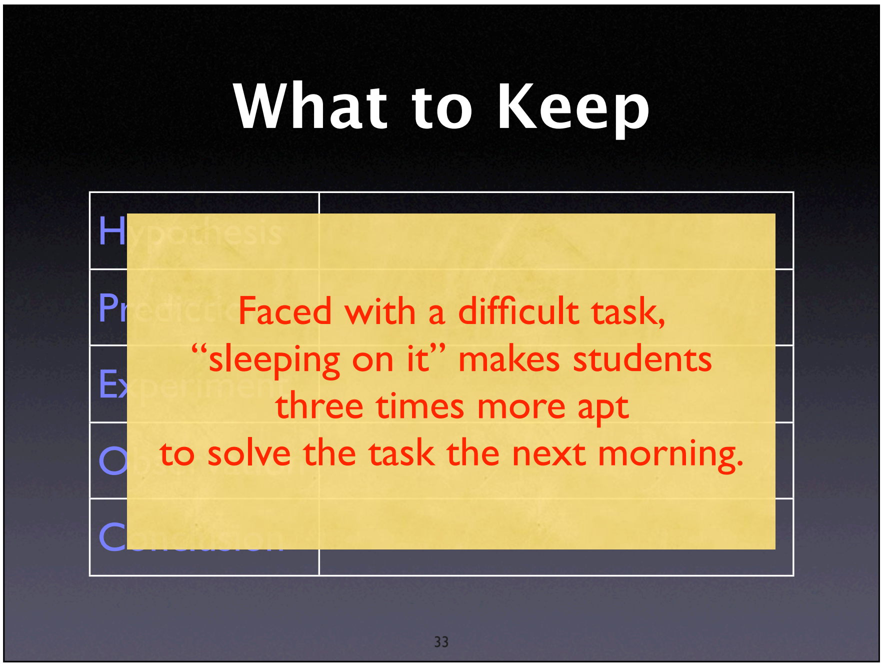


Keep a Notebook

Everything gets written down, formally, so that you know at all times

- where you are,
- where you've been,
- where you're going, and
- where you want to get.

Otherwise the problems get so complex you get lost in them.

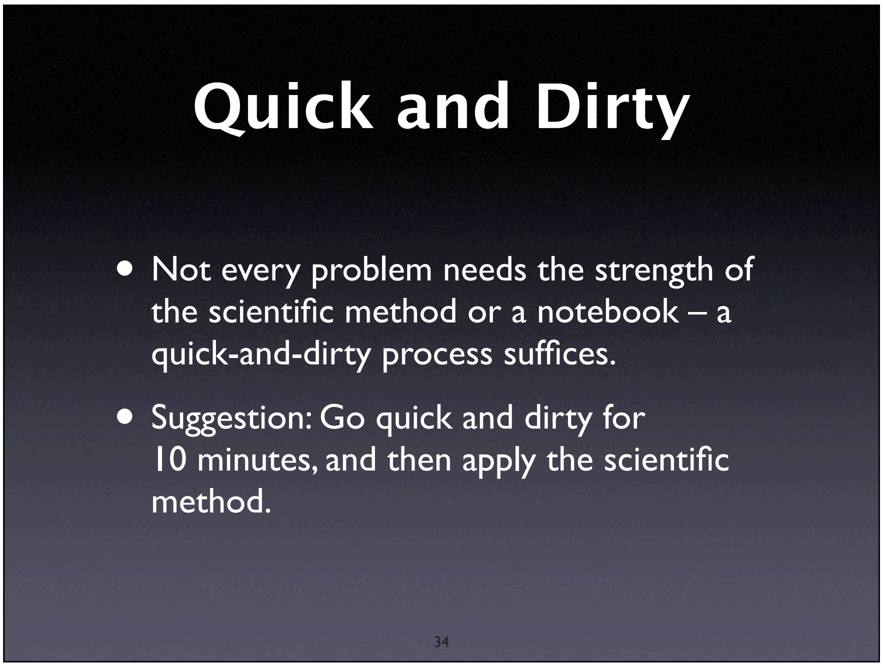


What to Keep

Hypothesis	
Pr	
Ex	
O	
Conclusion	

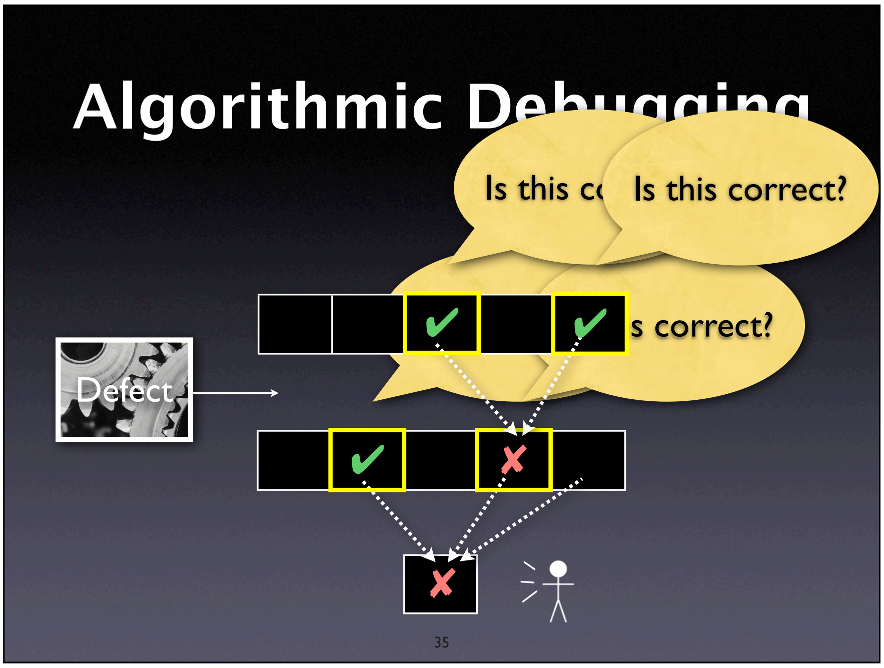
Faced with a difficult task,
“sleeping on it” makes students
three times more apt
to solve the task the next morning.

```
@Article{wagner/etal/2004/
nature,
  author =      {Ullrich
Wagner and Steffen Gais
and Hilde Haider and Rolf
Verleger and Jan Born},
  title =   {Sleep inspires
insight},
  journal =     {Nature},
  year =    2004,
  volume =  427,
  pages =   {325--355}
}
```

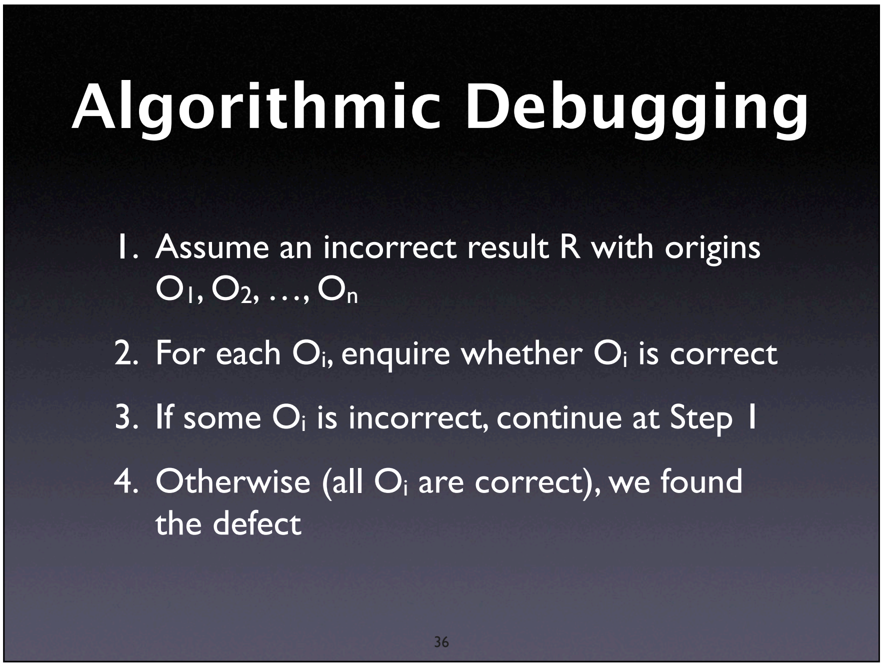


Quick and Dirty

- Not every problem needs the strength of the scientific method or a notebook – a quick-and-dirty process suffices.
- Suggestion: Go quick and dirty for 10 minutes, and then apply the scientific method.

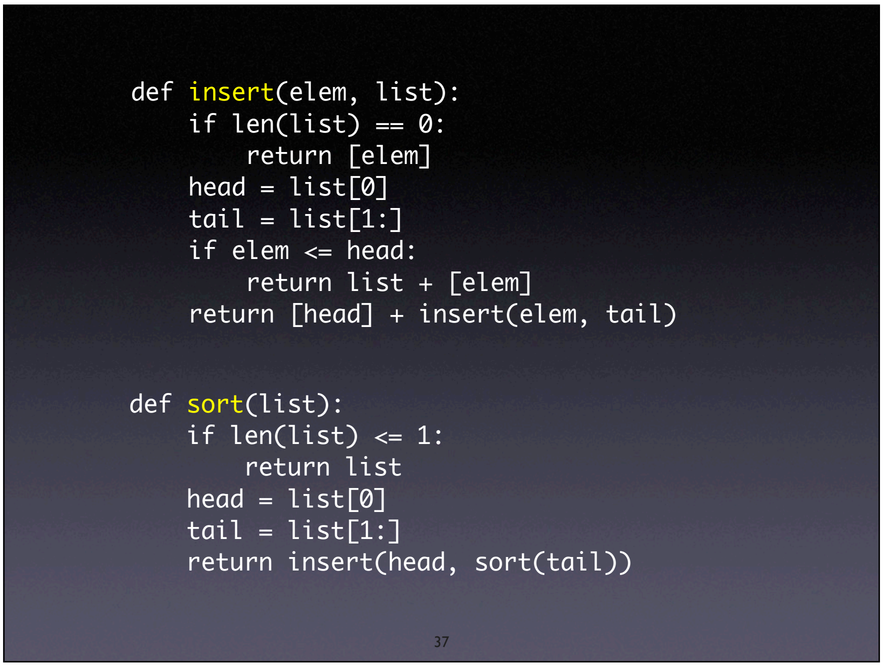


Algorithmic Debugging

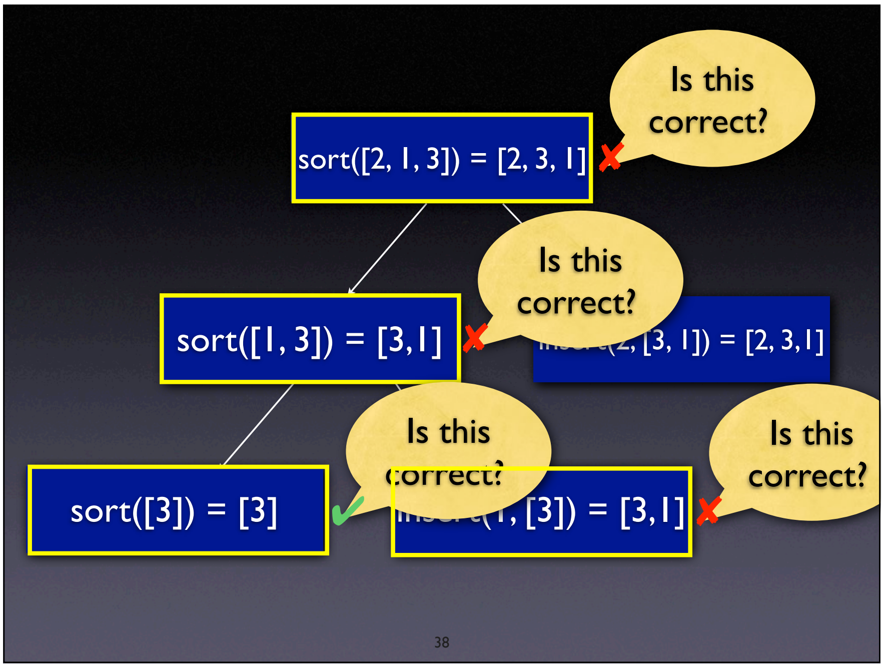


Algorithmic Debugging

1. Assume an incorrect result R with origins $O_1, O_2, \ldots, O_n$
2. For each O_i, enquire whether O_i is correct
3. If some O_i is incorrect, continue at Step 1
4. Otherwise (all O_i are correct), we found the defect



```
def insert(elem, list):
    if len(list) == 0:
        return [elem]
    head = list[0]
    tail = list[1:]
    if elem <= head:
        return list + [elem]
    return [head] + insert(elem, tail)


def sort(list):
    if len(list) <= 1:
        return list
    head = list[0]
    tail = list[1:]
    return insert(head, sort(tail))
```



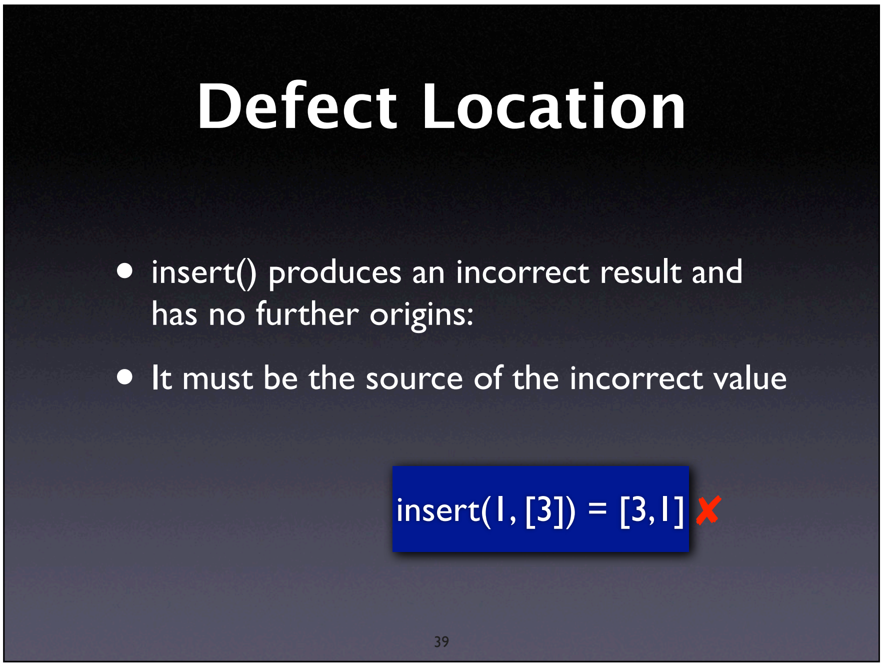
sort([2, 1, 3]) = [2, 3, 1] ✗

sort([1, 3]) = [3,1] ✗

insert(2, [3, 1]) = [2, 3, 1]

sort([3]) = [3] ✓

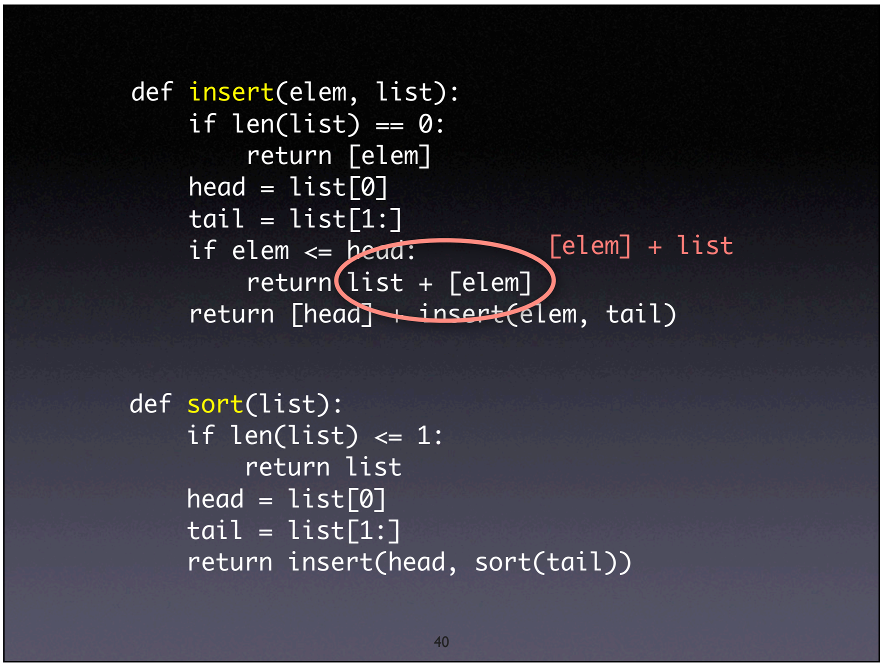
insert(1, [3]) = [3,1] ✗



Defect Location

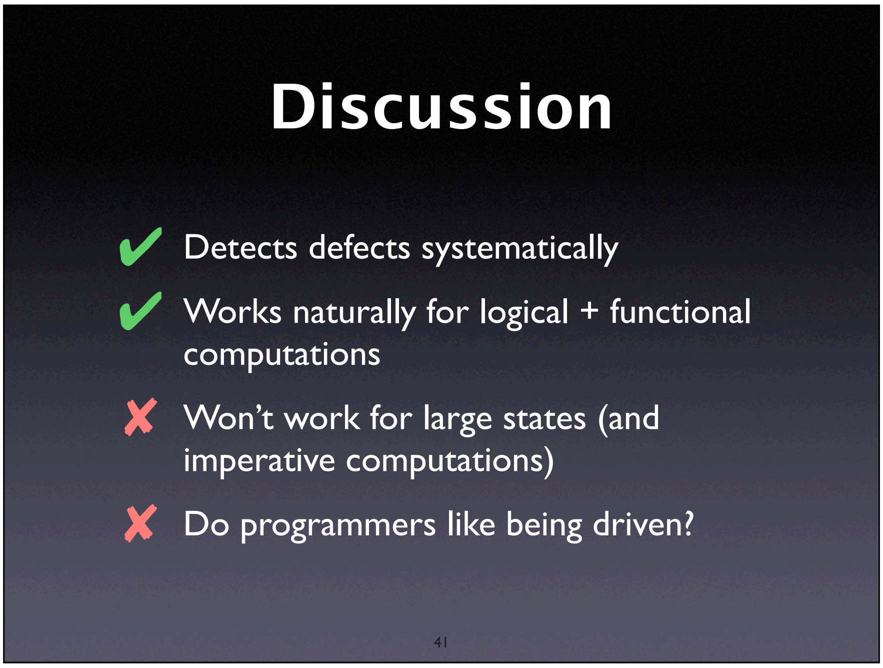
- insert() produces an incorrect result and has no further origins:
- It must be the source of the incorrect value

insert(1, [3]) = [3,1] ✗



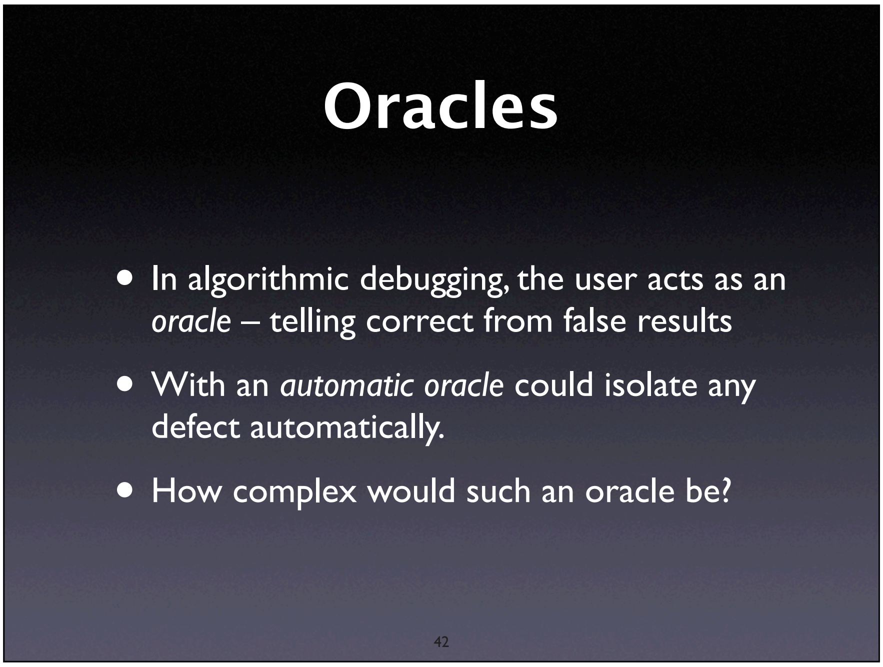
```
def insert(elem, list):
    if len(list) == 0:
        return [elem]
    head = list[0]
    tail = list[1:]
    if elem <= head:
        return list + [elem]
    return [head] + insert(elem, tail)


def sort(list):
    if len(list) <= 1:
        return list
    head = list[0]
    tail = list[1:]
    return insert(head, sort(tail))
```

[elem] + list



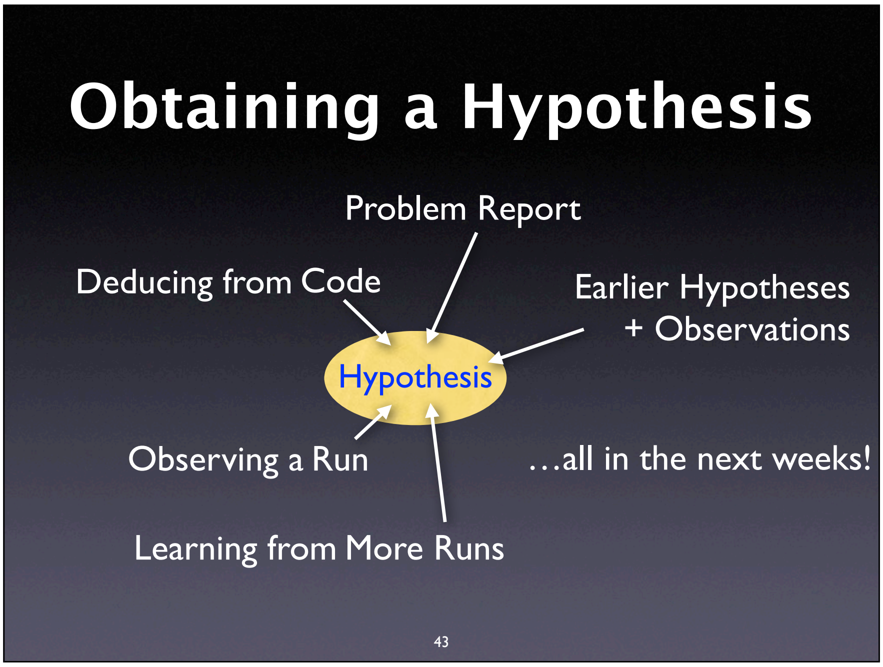
Discussion

- ✓ Detects defects systematically
- ✓ Works naturally for logical + functional computations
- ✗ Won't work for large states (and imperative computations)
- ✗ Do programmers like being driven?



Oracles

- In algorithmic debugging, the user acts as an *oracle* – telling correct from false results
- With an *automatic oracle* could isolate any defect automatically.
- How complex would such an oracle be?



Obtaining a Hypothesis

Problem Report

Deducing from Code

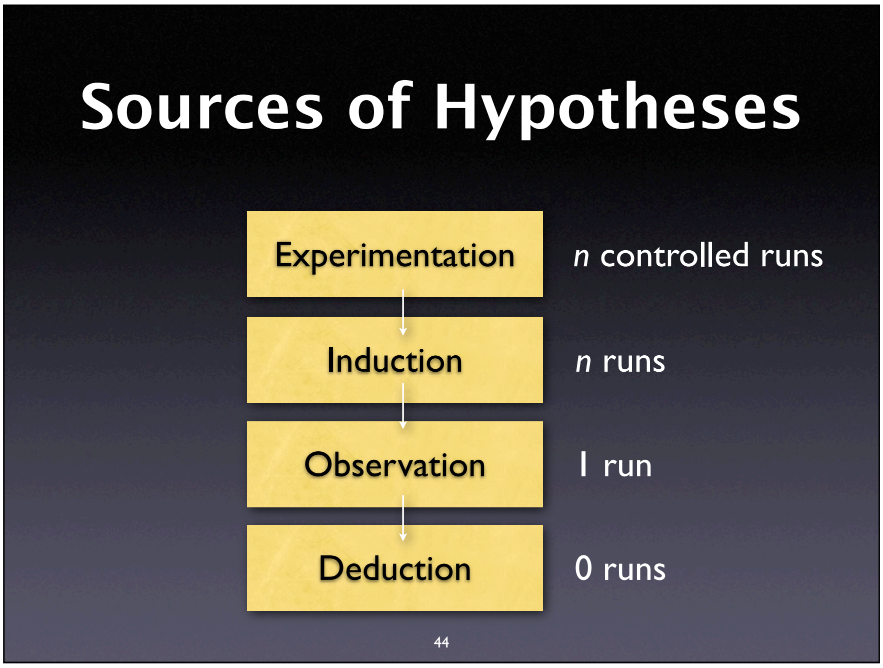
Earlier Hypotheses
+ Observations

Hypothesis

Observing a Run

…all in the next weeks!

Learning from More Runs



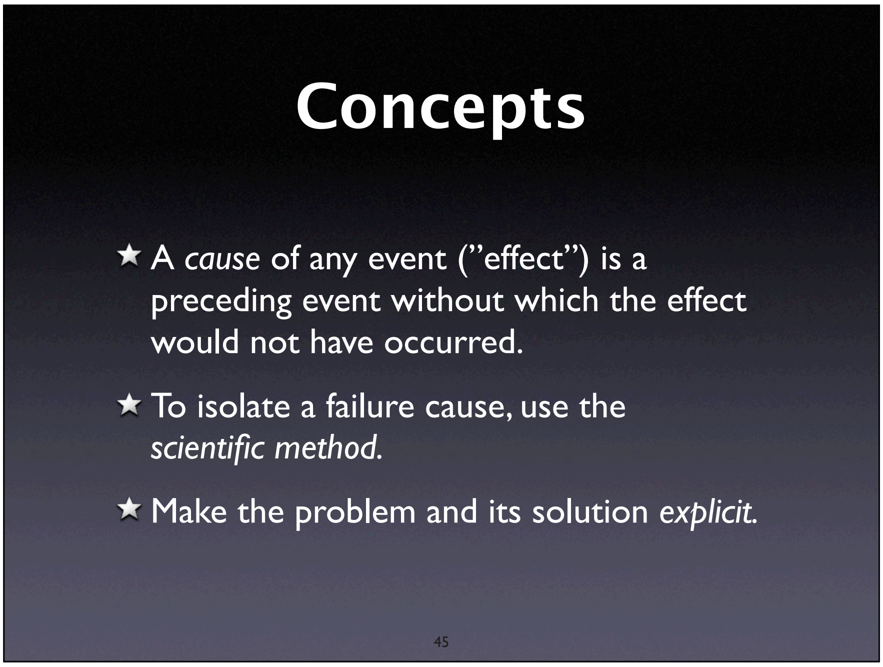
Sources of Hypotheses

Experimentation — *n* controlled runs

Induction — *n* runs

Observation — 1 run

Deduction — 0 runs

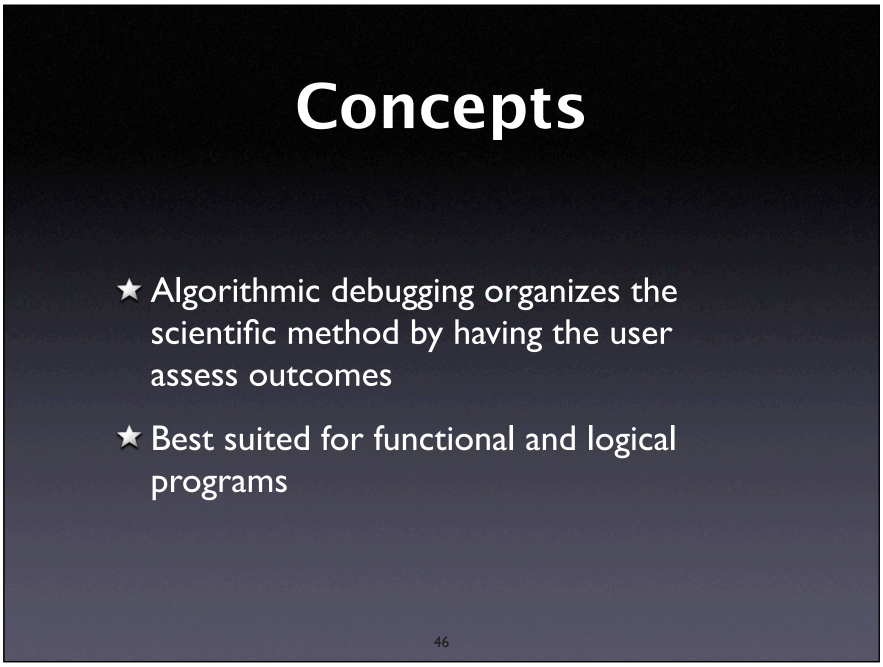


Concepts

★ A *cause* of any event (”effect”) is a preceding event without which the effect would not have occurred.

★ To isolate a failure cause, use the *scientific method.*

★ Make the problem and its solution *explicit.*



Concepts

- ★ Algorithmic debugging organizes the scientific method by having the user assess outcomes
- ★ Best suited for functional and logical programs