Student Copy
AIEP Scratch 2 Session 4A

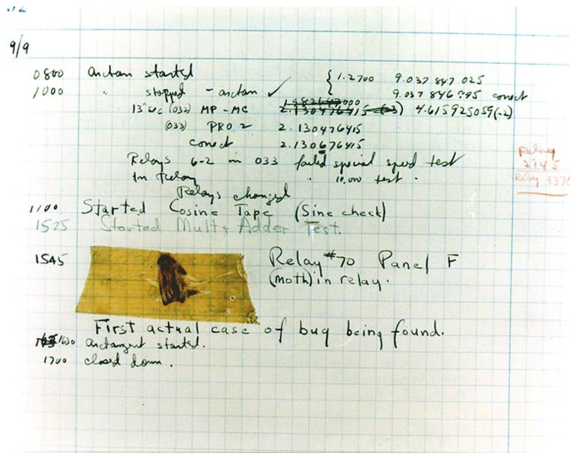
THE ART AND SCIENCE OF DEBUGGING

"Each new user of a new system uncovers a new class of bugs."

~ Brian Kernighan

In computer science, a BUG is an error in a program that causes it to behave contrary to what we would expect or intend.

The widespread usage of this funny-sounding term comes from an amusing story. In 1946, the computer pioneer Grace Murray Hopper and her team discovered an actual bug (technically, a moth) stuck inside their computer, Harvard Mark II, which was causing it to function weirdly. Thus, the first reported computer bug was truly a *bug* — a literal insect!



9/9
0800 Arctan started
1000 " stopped - arctan ✓ {1.2700 9.037 847 025 / 9.037 846 995 correct
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (-3) 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 11,000 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Arctangent started.
1700 closed down.

LEFT: The first computer bug (moth), taped on the Mark II log book

It is currently on display in the Smithsonian National Museum of American History.

RIGHT: Grace Murray Hopper (1906 – 1992)

She pioneered the idea of programming languages that can run on any type of machine, instead of being specific to one type or brand of device.

The act of fixing bugs in a program is called DEBUGGING. In English, the root word "de-" is commonly associated with removing something; in this case, we are trying to remove bugs, the errors or flaws in our code.

DEBUGGING TECHNIQUES

It is perfectly normal to have bugs in our code, and it is perfectly normal, too, to struggle finding and fixing them. Debugging is an essential skill that programmers constantly hone through years of experience. Here are some staple techniques to get us started:

1. Display the relevant variables

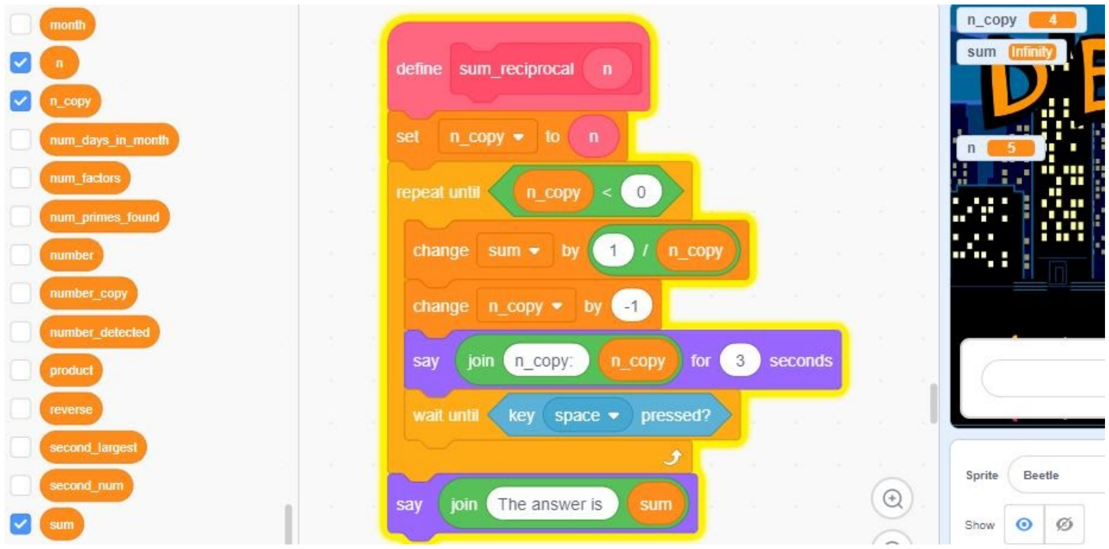
We must take advantage of the built-in variable tracker in Scratch, and use it to observe and study the behavior of our program. Initially, we may have to display all the variables in the relevant My Block; after all, each of them is "suspicious".

Once we have confirmed that some of the variables are behaving the way we intended, we slowly and systematically narrow down oour selection. Before we can remedy a problem, we must first give an accurate and precise "diagnosis."

2. Insert `say` and `wait` blocks

In most other programming languages, there is no built-in variable tracker (unless we are using a debugging tool); therefore, programmers have to rely on strategically placed `print` statements, the equivalent of Scratch `say` blocks.

Nonetheless, this practice is still useful in Scratch debugging, especially when the values of variables change too rapidly (for example, inside a `repeat` block) and the script has to be temporarily interrupted to properly track them. In this case, it may be advisable to insert a `say` block followed by a `wait until` block.



- ☐ month
- ✓ n
- ✓ n_copy
- ☐ num_days_in_month
- ☐ num_factors
- ☐ num_primes_found
- ☐ number
- ☐ number_copy
- ☐ number_detected
- ☐ product
- ☐ reverse
- ☐ second_largest
- ☐ second_num
- ✓ sum

```
define sum_reciprocal (n)
set [n_copy] to (n)
repeat until <(n_copy) < (0)>
    change [sum] by ((1) / (n_copy))
    change [n_copy] by (-1)
    say (join [n_copy: ] (n_copy)) for (3) seconds
    wait until <key [space] pressed?>
end
say (join [The answer is ] (sum))
```

```
n_copy 4
sum Infinity
n 5
```

3. Debug with a friend

This is best explained in the following quote by Brian Kernighan, a computer pioneer renowned for his role in the development of the C programming language:

> Another effective [debugging] technique is to explain your code to someone else. This will often cause you to explain the bug to yourself. Sometimes it takes no more than few sentences, followed by an embarrassed "Never mind, I see what's wrong. Sorry to bother you." This works remarkably well; you can even use non-programmers as listeners.

SEVEN CAPITAL BUGS

While it is impossible to provide an exhaustive classification of all possible bugs, this list of common oversights — as you might have realized through the Debug City activity (refer to page 7) — might be helpful in identifying error "hotspots" in our code:

1. Incorrect/Missing Initialization

Initialization refers to assigning an initial value to a variable. In Scratch, an uninitialized variable is assigned a value from the previous run of the program[1]. Hence, a good practice is to just initialize all the variables in our code.

Note that the initial value depends on the purpose of the program. In Denug City's `largest_number`, the bug is in the initial value of `largest`. It should have been initialized to -1 or any other smaller value. *Why?* The table below shows the usual initial values for common tasks:

Task	Usual Initial Value
Getting the sum	0
Getting the product	1
Getting the maximum value	Very small number ($-\infty$)
Getting the minimum value	Very large number ($+\infty$)

[1] Programming languages vary in their behaviors. In Python, an uninitialized numerical variable is assigned the value 0. In C and C++, a garbage value is (dangerously) assigned.

2. Integer vs. Floating-Point Arithmetic

The division block in Scratch returns a floating-point (also known as decimal) number[2]. If we want an integer instead — that is, we discard the remainder or the decimal portion — we have to take the floor function of the quotient (provided that the quotient is positive).

We have already seen how sneaky and destructive this bug is in decimal-to-binary conversion. In Debug City's `product_digits`, `number_copy` must be set to the floor of `number_copy / 10` since our goal is to remove the last digit, that is, we want our answer to be an integer (not a floating-point number) .

3. Inaccurate Boolean Operators

Boolean operators (named after the English mathematician George Boole) refer to logical connectives:

a) `not` - Flips the truth value of an expression
b) `and` - Returns true only if both expressions are true
c) `or` - Returns false only if both expressions are false

On the surface, the distinction between `and` and `or` is quite clear; after all, it reflects their usage in English. However, programming is an exercise in cold, hard logic, and a simple swap between `and` and `or` — which happens even to the more experienced programmers — changes the semantics of the code completely.

This is seen several times in Debug City. For instance, in `tribonacci`, the first condition should have been written as `if n < 3 or n = 3`. There is no number that is less than 3 *and* equal to 3 at the same time. If this bug were not fixed, the entire block of statements inside the `if` clause will never get executed[3]!

4. Improper `repeat` Block Condition

Knowing when to terminate the execution of a `repeat` block can be quite tricky. While it might be tempting to just "tweak and experiment" on the condition in Scratch until it gives us the correct output, this can be a dangerous habit. A good practice is to actually trace the execution on paper to ensure correctness.

[2] Again, programming languages vary in their behaviors. Python and Scratch share a similar behavior. On the other hand, in C and C++, dividing two integers results in an integer: 5 / 2 = 2, not 2.5.
[3] In computer science jargon, we call this unreachable code. Usually, it is a bug.

An example is featured in Debug City's `factors`. The condition should have been `candidate factor > number`. If this bug were not fixed, then the number itself will not be counted as its own factor.

A subtler bug can be found in `sum_reciprocal`. You might have noticed that the sprite says INFINITY, which clearly does not make sense. The error is in the terminating condition `n_copy < 0`. It should have been `n_copy < 1`. The point is that `n_copy` should not be allowed to be equal to 0, as the reciprocal of 0 is undefined[4] — signaled by the INFINITY result displayed.

5. Jumbled Sequence of Blocks

In almost all programming languages, the flow of code execution is from top to bottom. Ergo, the order in which the blocks are arranged is important, especially if their logic is dependent on each other. In real life, we can think of jumbled blocks as similar to placing on shoes before wearing socks.

For example, in `tribonacci`, the proper sequence is updating `tribonacci`, then `first_num`, followed by `second_num`, and, finally, `third_num`. Try to resequence the `set` blocks, and explain why reshuffling will cause the program to collapse.

6. Inexact/Incomplete Logic

Sometimes, the bug lies deep in the logic of our code. We might have missed some test cases, or, probably, our entire idea really does not work. Beyond being a cerebral activity, coding also helps in cultivating humility. We should be willing to revise our algorithm repeatedly — or even rebuild it from scratch, if necessary.

In Debug City's `swap_num`, a temporary variable `temp` has to be created to store the old value of `first_num`; it is this `temp` that gets assigned to `second_num`.

Challenge: There is a way to swap two values without a temporary variable (using carefully sequenced addition and subtraction blocks). Try to figure it out!

In `second_largest_number`, an entire case is missed: What if `number` is less than `largest` but greater than `second_largest`? The key to catching these missing cases is patient and "playful" testing with random values.

[4] In C, C++, Java, and most other programming languages, this will cause your program to crash.

7. Typographical Errors

A quarter of the bugs that I encountered can be ascribed to typos! This is not much of a problem in Scratch since we first have to "declare" (that is, create) a variable before using it. However, in most dynamically typed languages like Python, where variables can be used on the fly, this is a real source of headache.

Nevertheless, we still have to be careful about typos in Scratch. In Debug City's `invalid_input`, specifically in the My Block `words (no numbers)`, two such errors are hidden: `Yos` and `Yes   `. Note that `Yes   ` (emphasis on the three spaces after `Yes`) is different from a simple `Yes` (no space after it).

The moral of the story: strings are compared character by character[5].

AN ART AND A SCIENCE

Like programming, debugging is both an art and a science:

a) Art — We want the fix to be as concise and as elegant as possible. A haphazardly written fix frequently ends up being unmaintainable; worse, it might be the root of other bugs in the future. Why go through the trouble of writing a separate `if-else` block if it can be fixed with a single artistic tweak to the original condition?

A common mistake among novice programmers is attempting to fix a line of code that is already correct. Just because we do not immediately understand what a line of code is doing does not mean that we should rewrite it entirely!

b) Science — Detecting and fixing bugs follow the scientific method. First, we have to get an idea of what the code is supposed to do versus what it is actually doing. Then, we try to trace each line of code, inserting some helpful `say` blocks along the way and identifying which among the variables are assigned incorrect value.

More importantly, we pinpoint the exact cause of the unintended behavior. The checklist in this handout might help. Finally, once we have written our fix, we repeatedly try to "break" the program to ensure that the problem has been solved!

[5] String comparison in Scratch is not case-sensitive. This means that `yes` and `YES` are treated as the same. However, in most other programming languages, capital and small letters are treated differently. Therefore, it is better to be a bit more disciplined and be aware of this quirk as early as now.

With all these said, there is no such thing as a bug-free program. This is one of the reasons why programs, from small apps that we can play on our phones to large-scale operating systems (like Windows and macOS), release "patches" and updates periodically.

Even Scratch has bugs! Go to https://en.scratch-wiki.info/wiki/Bugs_and_Glitches_(forum) to learn more.

ACTIVITY: DEBUG CITY

Load the Scratch file Debug City_For Students.sb3. Each of the 11 letter sprites reveals a code snippet that needs to be debugged, as explained in the table below[6]:

	Expected Behavior	My Blocks Involved
D	**Swaps the numbers assigned to two variables.** If `first_num` is 4 and `second_num` is 5, swapping them will result in `first_num` taking the value 5 and `second_num` taking the value 4.	`swap_num @,@`
E	**Get the positive factors of a number.** The positive factors of 2021 are 1, 43, 47, and 2021. The positive factor of 1 is just 1 (be careful about this).	`factors @`
B	**Get the n^{th} prime number.** The first 7 prime numbers are 2, 3, 5, 7, 11, 13, and 17.	`@th prime` `is_prime @`
U	**Get the product of the digits of a number.** The product of the digits of 456 is 120.	`product_digits @`
G	**Check if a given number is a palindrome.** The numbers 1234321 and 2332 are palindromes. On the contrary, 1234 is not a palindrome.	`is_palindrome @`

[6] The @ in the names of the My Blocks is a shorthand for the parameters (the variables enclosed in pink oblongs).

C	**Get the n^{th} Tribonacci number.** If the first three Tribonacci numbers are 4, 5, and 6, the 6th Tribonacci number is 47.	`tribonacci @,@,@,@`

I	**Catch invalid input (keep asking for input until a valid input is given by the user).**
	If the choice is 1 (integers only), then 56 and -56 are valid inputs. On the contrary, 3.14 is invalid.
	If the choice is 2 (numbers only, including decimals), then 3.14 is valid. On the contrary, `aiep2021` is invalid.
	If the choice is 3 (words only, no numbers), then `aiep` is valid. On the contrary, `aiep2021` is invalid.
		`menu @`
		`integers only`
		`numbers (with decimals)`
		`words (no numbers)`

T	**Get the sum of the reciprocals of 1 up to *n*.** The value of $\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{10}$ is 2.9289682539682538.	`sum_reciprocal @`

Y	**Catch invalid dates MM-DD-YYYY (keep asking for input until a valid date is given by the user).**
	`02-29-2000`, `02-28-1000`, and `12-31-9999` are valid dates. On the contrary, `02-29-1700`, `2-29-2020`, and `01-32-9999` are invalid dates. Be careful about leap years.
		`check_month @`
		`check_day @`
		`check_year @`
		`is_leap_year @`
		`number_days @`

G	**Get the largest number (without using lists, restricted to nonnegative numbers only).** Given the sequence of input 1 5 0 100 99 0 -1, the largest number is 100.	-
O	**Get the second largest number (without using lists, restricted to nonnegative numbers only).** Given the sequence of input 1 5 0 100 99 0 -1, the second largest number is 99.	-

`-- return 0; --`