Lecture 5: Debugging, Iteration, and Recursion

Math 98, Fall 2023

Reminders and Agenda

- Using the MATLAB debugger
  - Breakpoints
  - Step and Run
  - Review the MATLAB Documentation
- Exercise on finding bugs
- Debugging and Programming Best Practices
  - Sections 2.7-2.8 of http://www.sfu.ca/~wcs/ForGrads/ensc180spring2016f.pdf
- Iteration vs. Recursion
- Exercises

Demo: Using the MATLAB Debugger

- Breakpoints
- Step and Run
- Review the MATLAB Documentation

bisectionbuggy.m

Consider this implementation of bisection

```
function p = bisectionbuggy(f, a, b, tol)
    while 1
        p = (a+b)/2;
        if p - a < tol
            break;
        end
        if f(b)*f(p) > 0
            a = p;
        else
            b = p;
        end
    end
end
```

bisectionbuggy.m: Wrong Output

There's clearly something wrong with this....

```
>> p = bisectionbuggy(@(x) x, -1, 2, 1e-4)
p =
   1.999908447265625
```

Let's see if we can smoke out the bug with the debugger.

`bisectionbuggy.m`: The Solution

Solution: Either change `f(a)*f(p) > 0` or `f(b)*f(p) < 0` or switch `a = p` and `b = p`

Incremental Development

When you start writing scripts that are more than a few lines, you might find yourself spending more and more time debugging. The more code you write before you start debugging, the harder it is to find the problem.

Incremental development is a way of programming that tries to minimize the pain of debugging.

Incremental Development: Three Steps

The fundamental steps of incremental debugging are:

1. Always start with a working program. If you have an example from a book or a program you wrote that is similar to what you are working on, start with that. Otherwise, start with something you know is correct, like $x = 5$. Run the program and confirm that you are running the program you think you are running. This step is important, because in most environments there are lots of little things that can trip you up when you start a new project. Get them out of the way so you can focus on programming.
2. Make one small, testable change at a time. A "testable" change is one that displays something on the screen (or has some other effect) that you can check. Ideally, you should know what the correct answer is, or be able to check it by performing another computation.
3. Run the program and see if the change worked. If so, go back to Step 2. If not, you will have to do some debugging, but if the change you made was small, it shouldn't take long to find the problem.

Unit Testing

In large software projects, **unit testing** is the process of testing software components in isolation before putting them together.

The programs we have seen so far are not big enough to need unit testing, but the same principle applies when you are working with a new function or a new language feature for the first time. You should test it in isolation before you put it into your program.

Unit Testing: Example

For example, suppose you know that x is the sine of some angle and you want to find the angle. You find the MATLAB function `asin`, and you are pretty sure it computes the inverse sine function. Pretty sure is not good enough; you want to be very sure.
Since we know $\sin(0) = 0$, we could try:

```
>> asin(0)
ans =
    0
```

which is correct. We also know that sin of 90° is 1, so if we try `asin(1)` we expect the answer 90, right?

```
>> asin(1)
ans =
    1.5708
```

What's going on here?

Unit Testing: Example (cont.)

Oops. We forgot that the trig functions in MATLAB work in radians, not degrees. So the correct answer is $\frac{\pi}{2}$, which we can confirm by dividing through by π:

```
>> asin(1)/pi
ans =
    0.5000
```

With this kind of unit testing, you are not really checking for errors in MATLAB, you are checking your understanding. If you make an error because you are confused about how MATLAB works, it might take a long time to find, because when you look at the code, it looks right.

```
The worst bugs aren't in your code; they are in your head
```

Debugging in Four Acts

- **Reading**: Examine your code, read it back to yourself, and check that it means what you meant to say.
- **Running**: Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding.
- **Ruminating**: Take some time to think! What kind of error is it: syntax, runtime, logical? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you?re seeing? What did you change last, before the problem appeared?
- **Retreating**: At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works, and that you understand. Then you can starting rebuilding.

Iteration: Motivations

Many tasks in life are boring or tedious because they require doing the same basic actions over and over again – `iterating` – in slightly different contexts.

- So let's get the computer to do this!
- `for` loops and `while` loops.

Iteration: `for` loops and `while` loops

A statement to repeat a section of code a specified number of times.

```
for countVariable = 1 : numberOfIterations
    % do something here
    % this part will run
    % (numberOfIterations) times
end
```

A statement to repeat a section of code *until* some condition is satisfied.

```
while [EXPRESSION is true]
    % repeat this part until
    % (EXPRESSION) is false
    % be sure to modify (EXPRESSION) in this loop
end
```

Fixed Point Iteration: Example

Let's say we're interested in this fixed iteration

$$\varphi(x) = \sqrt{1+x} \qquad x_0 = 3$$

After 10 iterations.

```
>> x = 3;
>> x = sqrt(1+x)
x =
     2
.........
>> x = sqrt(1+x)
x =
    1.618064196086926
>> x = sqrt(1+x)
x =
    1.618043323303466
```

Fixed Point Iteration: For Loop

I claim this converges to $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618033988749895$. This is the golden ratio, one of the most famous numbers in mathematics.

I probably should have done the above calculation with a for loop.

```
>> x = 3;
for k = 1:10
    x = sqrt(1+x);
end
x
x =
    1.618043323303466
```

Fixed Point Iteration: While Loop

Let's do this with a while loop until it "converges", until the computer can't tell the difference anymore.

```
>> x = 3;
while x~= sqrt(1+x)
    x = sqrt(1+x);
end
x
x =
    1.618033988749895
>> x == (1+sqrt(5))/2
ans =
    logical
    1
```

Infinite Loops

Careful with infinite loops!

```
>> N = 0;
while N > -1
    N = N + 1;
end
```

Put maximum iteration limits and breaks in your loops to guard for this.

Factorial as an Iteration

How do we compute the factorial of a number?

$$n! = \begin{cases} 1 & n == 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

A `for` loop will do nicely.

```
function nfac = myFactorial(n)
    nfac = 1;
    for i = 1:n
        nfac = nfac * i;
    end
end
```

Factorial as a Recursion

How do we compute the factorial of a number?

$$n! = \begin{cases} 1 & n == 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

We can also take advantage of the recursive definition, and define our function recursively:

```
function nfac = myFactorial(n)
    if n == 0
        nfac = 1;
    else
        nfac = n*myFactorial(n-1);
    end
end
```

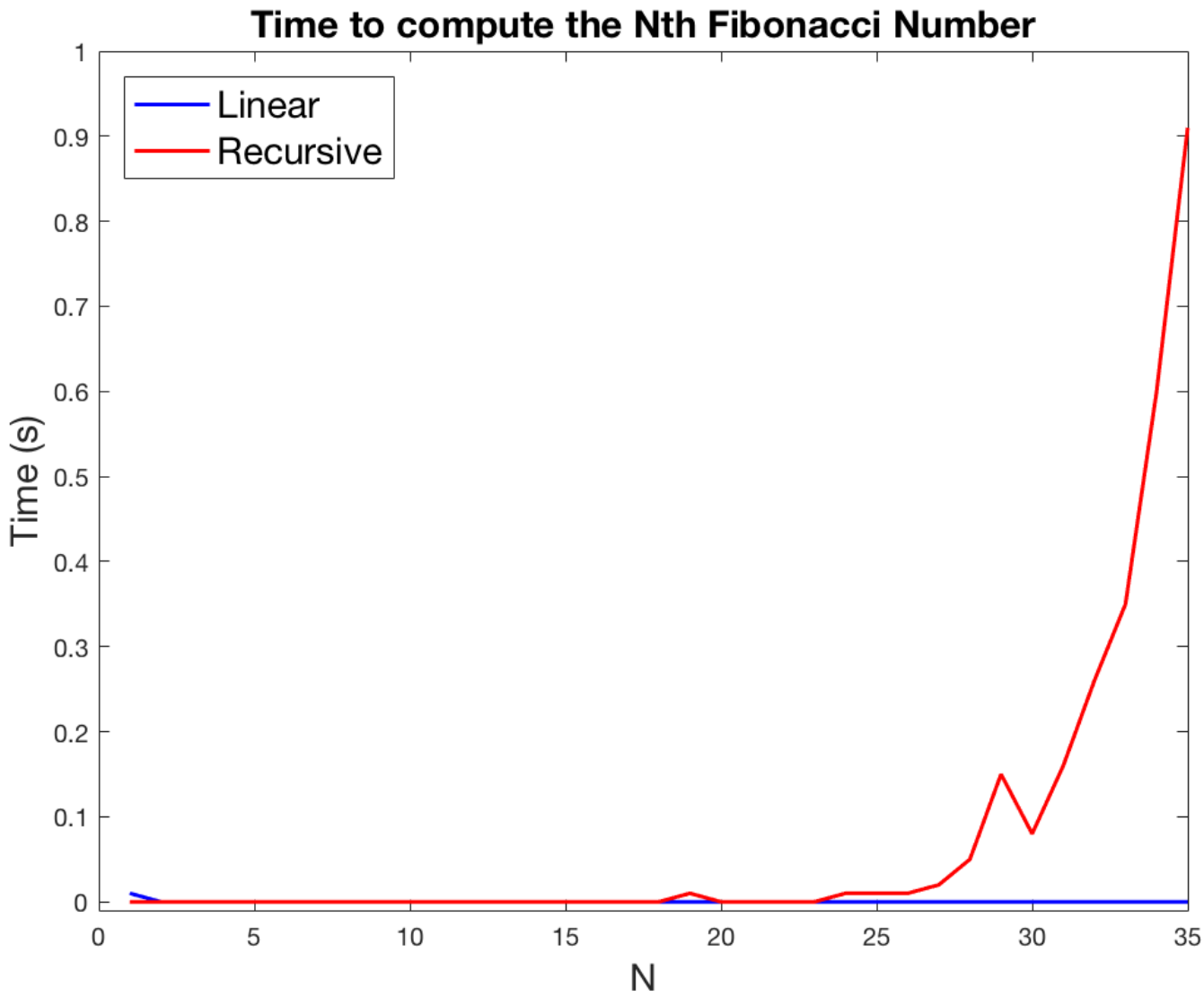
Exercise: Fibonacci Numbers

Define the Fibonacci numbers as

$$f(n) = \begin{cases} 0 & n == 0 \\ 1 & n == 1 \\ f(n-1) + f(n-2) & n >= 2 \end{cases}$$

Write a recursive function `fiboRec.m` to compute $f(n)$, then write a non-recursive function `fiboLin.m` (using a `for` loop) to do the same. The non-recursive function should compute all numbers $f(0), f(1), \ldots, f(n)$.

Fibonacci Numbers: Compute Times



Fibonacci Numbers: Compute Times

The problem: our recursive definition did lots of unnecessary computation by not using previously computed values.

```
>> fiboRec(4)
Computing f(4)
Computing f(2)
Computing f(0)
Computing f(1)
Computing f(3)
Computing f(1)
Computing f(2)
Computing f(0)
Computing f(1)
ans =
      3
```

Iteration Exercise: `nested_sqrt.m`

Write a function

```
function a = nested_sqrt(n)
```

that takes an integer n and returns the nth term in the following sequence:

$$a_1 = 1, a_2 = \sqrt{1+2}, a_3 = \sqrt{1+2\sqrt{1+3}}, a_4 = \sqrt{1+2\sqrt{1+3\sqrt{1+4}}}, \ldots$$

Guess the limiting value of the sequence $a = \lim_{n\to\infty} a_n$ and make a plot of $\ln(|a_n - a|)$ vs. n. Also plot the line $y = 3 - (\ln 2)n$.
What sequence β_n would you guess is appropriate for $a_n - a = O(\beta_n)$?

Recursion Exercise: `qsort.m`

How do we sort a list of numbers v?
There are many ways, but `quickSort` offers a simple recursive implementation.

1. Pick an element $x \in v$ to be the **pivot** element. (say, the first one).
2. Divide the rest of the list in two: those **smaller** than x and those **larger** than x.
3. `output = [quickSort(Smaller), x, quickSort(Larger)]`

A few questions we need to answer when working out the details:

- What are the base cases that we need to handle?
- What if some numbers are equal to x?

Implement

```
function w = qsort(v)
```