NUMBERS, APPROXIMATIONS, and BISECTION

(download slides and .py files to follow along)

6.0001 LECTURE 3

Eric Grimson

Last Time

- new data structure – strings
- iteration and loops – while, for
- guess and check algorithms

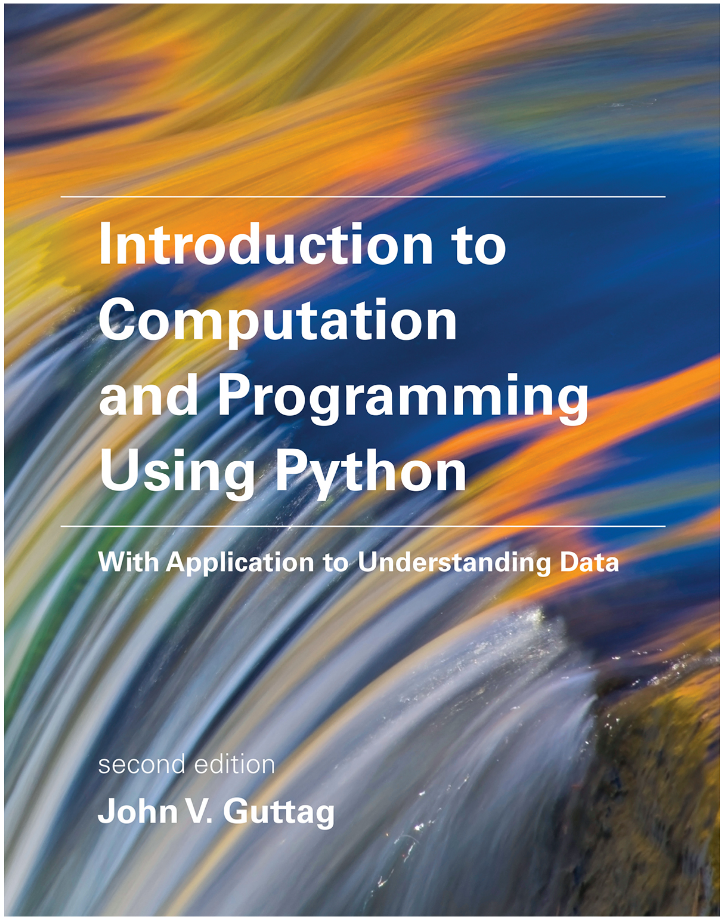
Today

- a short digression:
  - representing numbers
- approximate solutions
- guess & check algorithms using approximations
- bisection methods

Assigned Reading

"I'm sorry you didn't do the assigned reading, but no, I'm not going to say 'spoiler alert' every time something interesting happens."

- Today:
  - Sections 3.3 – 3.5
- Next lecture:
  - Section 4.1 – 4.3



See https://mitpress.mit.edu/books/introduction-computation-and-programming-using-python-second-edition for errata sheet

Numbers in Python

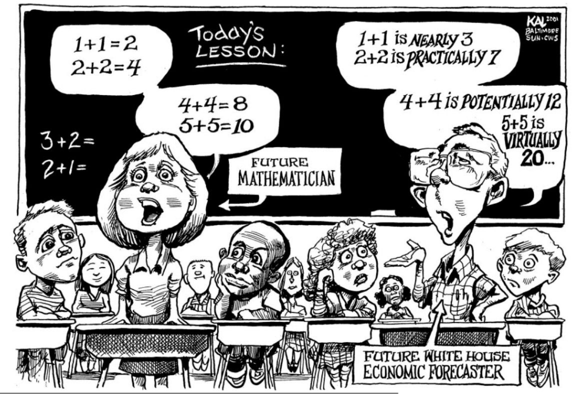
- **int:** integers, (or whole numbers), like the ones you learned about in elementary school
- **float:** ~~reals, (or numbers with digits after the decimal point), like the ones you learned about in middle school~~

A Closer Look at Floats

- Python (and all programming languages) uses "floating point" to **approximate** real numbers
- The term "floating point" refers to way these numbers are stored in computer (more later about this)
- You would hope that approximating real numbers in computations usually shouldn't matter

But "usually shouldn't matter" is another way of saying what?

Does approximation matter?




```
x = 0
for i in range(10):
    x += 0.1
print(x == 1)
print(x, '==', 10*0.1)
```

Note:
x += 0.1
is the same as
x = x + 0.1

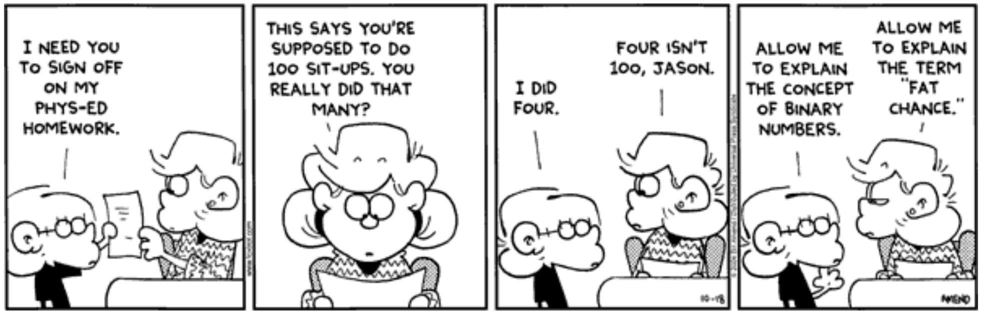
So, approximating real numbers matters!

Is it "close enough for government work"?

Adding 10 instances of 0.1 is not the same as multiplying 10 by 0.1?

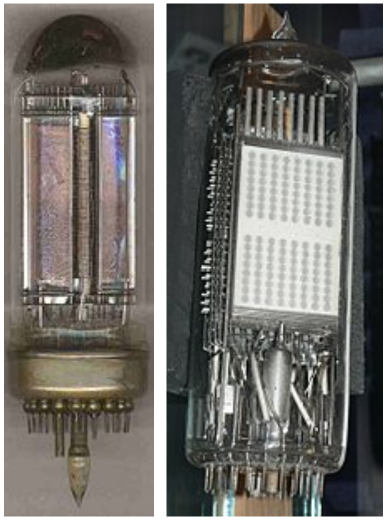
Why?

- Representation of floating point numbers is function of computer hardware, not programming language implementation
- Usual representation: standard called IEEE 754 floating point
- Key things to understand
  - In all modern computers, numbers (and everything else) are represented as a **sequence of bits** (0 or 1). Think of these as binary numbers (i.e., base 2)
  - When **we** write numbers down, we are using a notation designed to express rational numbers using base 10. E.g., 0.1 stands for the rational number 1/10
  - This produces **cognitive dissonance** – and it will influence how we write code

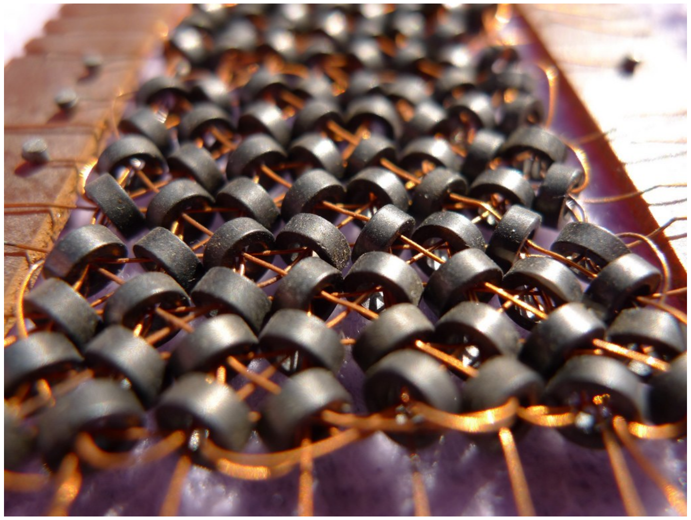


Why Binary?

- Easy to implement in hardware—build components that can be in **one** of **two** states



4096 bit; 256 bit
$2.00 per bit



Core memory

1960 ($0.62/bit) - 1971 ($0.004/bit)

What does a bit of dynamic RAM cost today? $0.000000002/bit

Binary Numbers

- Base 10 representation of an integer
  - sum of powers of 10, scaled by integers from 0 to 9

$1507 = 1*10^3 + 5*10^2 + 0*10^1 + 7*10^0$

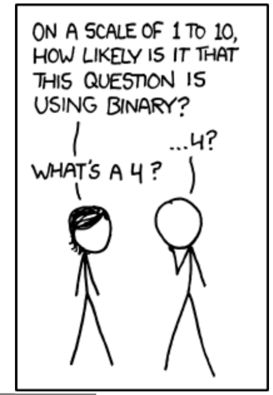
$= 1000 + 500 + 7$

- Binary representation is same idea in base 2
  - sum of powers of 2, scaled by integers from 0 to 1
- $1507_{10} = 1*2^{10} + 1*2^8 + 1*2^7 + 1*2^6 + 1*2^5 + 1*2^1 + 1*2^0$

$= 1024 + 256 + 128 + 64 + 32 + 2 + 1$

$= 10111100011_2$

Converting Decimal Integer to Binary



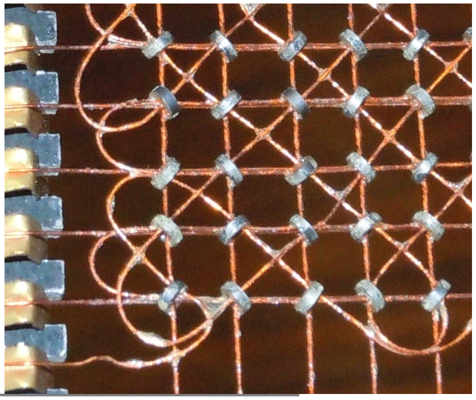
- We input integers in decimal form, computer needs to convert to binary, so it can store/manipulate the numbers
- Consider example of
  - $x = 19_{10} = 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 10011$
- If we take remainder of x relative to 2 (`19%2` in our example) , that gives us the smallest binary digit (bit)
- If we then integer divide x by 2 (`19//2` in our example), all the bits get shifted right
  - $19//2 = 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 = 1001$
- Repeat on the remainder; this gets next bit, and new remainder, and so on
- Let's us convert to binary form

Doing this in Python


```
if num < 0:
    isNeg = True
    num = abs(num)
else:
    isNeg = False
result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num//2
if isNeg:
    result = '-' + result
```

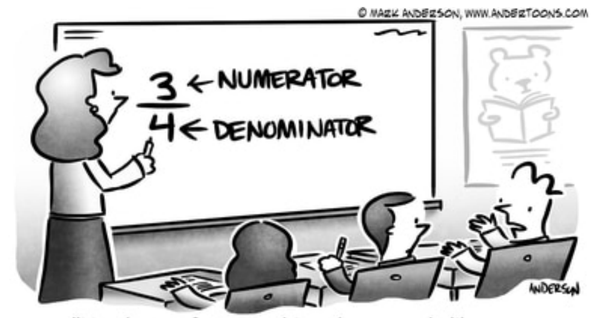
Note: this gives us a result as a string, but idea holds for numbers

Hardware Implementation



- Computer hardware is built around methods that can efficiently store information as 0's or 1's (a voltage is "high" or "low"; or a magnetic spin is "up" or "down") and that can efficiently perform arithmetic operations on such representations
- Fine for integer arithmetic
- But what about numbers with fractional parts (floats)?

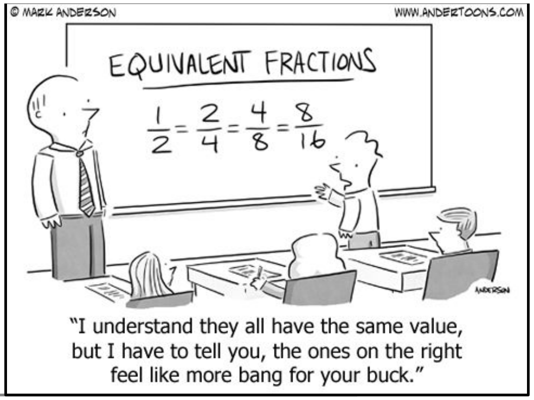
Fractions



"You know, for something that sounds like two killer robots, this is really disappointing."

- What does the decimal fraction 0.abc mean?
  - $a*10^{-1} + b*10^{-2} + c*10^{-3}$
- For binary representation, we use the same idea (where a, b, c are either 0 or 1)
  - $a*2^{-1} + b*2^{-2} + c*2^{-3}$
- Or to put in simpler terms, the binary representation of a decimal fraction f would require finding the values of a, b, c, etc. (all either 0 or 1) such that
  - $f = 0.5a + 0.25b + 0.125c + 0.0625d + 0.03125e + \ldots$

What About Fractions?



- How might we find that representation?
- In decimal form: $3/8 = 0.375 = 3*10^{-1} + 7*10^{-2} + 5*10^{-3}$
- If we can multiply by a power of 2 big enough to turn into a whole number, can convert to binary (using previous method), and then divide by the same power of 2 to restore
- $0.375 * (2**3) = 3_{10}$
- Convert 3 to binary, yielding 11_2
- Divide by 2**3 (shift right three spots) to get 0.011_2

Check: 0.011_2 is $1/4 + 1/8 = 3/8$

```python
x = float(input('Enter a decimal number between 0 and 1: '))
```

```python
p = 0
while ((2**p)*x)%1 != 0:
    print('Remainder = ' + str((2**p)*x - int((2**p)*x)))
    p += 1
```

Find power of 2 to make integer

```python
num = int(x*(2**p))
```

Convert to int

```python
result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num//2
```

Encode as binary number

```python
for i in range(p - len(result)):
    result = '0' + result
```

Pad front with 0's

```python
result = result[0:-p] + '.' + result[-p:]
```

Insert decimal

```python
print('The binary representation of the decimal ' + str(x) + ' is
' + str(result))
```

But …

- Why did display of remainder suddenly jump from simple fraction to number with a bunch of zeroes and a tiny additional amount?
- I am assuming that the representation for the decimal fraction I provided as input is completely accurate and not already an approximation as a result of number being read into Python
- Moreover, if there is no integer p such that x*(2**p) is a whole number, then internal representation is **always** an approximation, and I can't find the exact binary representation
- Hence, while the floating point conversion will work precisely for numbers like 3/8, it will not work for numbers like 1/10
  - The first example has a power of 2 that converts to whole number, the second one doesn't

Why is this a problem?

- What does the decimal representation 0.125 mean?
  - $1*10^{-1} + 2*10^{-2} + 5*10^{-3}$
- Suppose we want to represent it in binary?
  - $1*2^{-3}$ 0.001 (only need a few bits)
- How how about the representation 0.1?
  - In base 10: $1 * 10^{-1}$
  - In base 2: ? 0.00011001100110011001100110011… goes on forever

Any finite number of bits gives us an approximation

And the point is?

"It's not my fault. If a decimal point is so important, they should make it BIGGER!"

- If everything ultimately is represented in terms of bits, we need to think about how to use binary representation to capture numbers
- Integers are straightforward
- But real numbers (things with digits after the decimal point) are a problem:
  - Have to somehow **approximate** the potentially infinite binary sequence of bits needed to represent them
  - **MORE IMPORTANTLY,** have to consider how approximation of numbers will impact algorithm design

Floating Point Numbers

I used to hate math, but then I realized decimals have a point.

- Floating point representation is a **pair of integers**
  - Consists of a set of significant digits and a base 2 exponent
  - (1, 1) → $1*2^{1}$ → 10_2 → 2.0
  - (1, -1) → $1*2^{-1}$ → 0.1_2 → 0.5
  - (125, -2) → $125*2^{-2}$ → 11111.01_2 → 31.25

Called "floating point" because location of decimal can "float" relative to significant digits

- The maximum number of significant digits governs the precision with which numbers can be represented
  - When exceeded, numbers are rounded
- Most modern computers use 32 bits to represent significant digits, so error will only be on order of $\mathbf{2*10^{-10}}$

After all, 10^{-10} is a pretty small number, isn't it?

Because You Can Get Surprising Results

```
x = 0
for i in range(10):
    x += 0.125
print(x == 1.25)
```

True

```
x = 0
for i in range(10):
    x += 0.1
print(x == 1)
```

False

```
print(x, '==', 10*0.1)
```

0.9999999999999999 == 1.0

The Moral of the Story

Never, ever, use == to test floats

- Instead test whether they are within small amount of each other

What gets **printed** isn't always what is in **memory**

Need to be careful in **designing algorithms** that use floats, to account for approximate representations of numbers

Effect of approximation on our algorithms?

- Exact answer may not be accessible
- Need to find ways to decide when we have a “good enough” answer – is it close enough to ideal answer?
- Need ways to deal with fact that exhaustive enumeration can’t test every possible value, since set of possible guesses to check is in principle infinite

Finding Roots

- Last lecture we looked at using guess & check methods to find the **roots of perfect squares**
- Suppose we want to find the square root of any positive integer, or any positive number
  - Answer may no longer be an integer, so need to change how we generate guesses
  - Answer may not be found exactly, so need to change how decide when we are done
- Back to original question: What does it mean to find the square root of x?
  - Find an r such that r*r = x ?
  - If x is not a perfect square, then not possible in general to find an exact r that satisfies this relationship, but may find an r so that |r*r –x | is small

Find the root of a perfect food (truffle)

Approximation

- Find an answer that is "good enough"
  - E.g., find a r such that r*r is within a given (small) distance of x
  - By tradition, use epsilon for distance, so given x we want to find r such that $|r^2\text{-x}|<\varepsilon$

- Algorithm
  - Start with guess known to be too small – call it g
  - Increment by some small value – call it a – to give a new guess g
  - Check if `g**2` is close enough to `x` (within ε)
  - Continue until get answer close enough to actual answer

- Essentially, we are looking at all values `g + k*a` for positive integer values of `k` – so similar to exhaustive enumeration
  - May not find exact answer because of choice of k, initial value of g

Approximation Algorithms

- In this case, we have two parameters to set – epsilon (how close are we to answer?) and increment (how much to increase our guess?)
- Performance will vary based on these values
  - In speed
  - In accuracy
- Decreasing increment size → slower program, but more likely to get closer to real answer
- Increasing epsilon → less accurate answer, but faster program

Implementation

```
x = 36
epsilon = 0.01
numGuesses = 0
ans = 0.0
increment = 0.0001
while abs(ans**2 - x) >= epsilon:
    ans += increment
    numGuesses += 1
print('numGuesses =', numGuesses)
if abs(ans**2 - x) >= epsilon:
    print('Failed on square root of', x)
else:
    print(ans, 'is close to square root of', x)
```

Why do we think that loop will terminate?

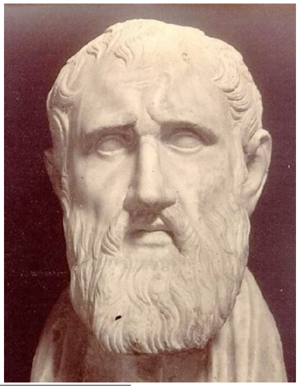
Note: ans += increment is same as ans = ans + increment

Reasoning About Loop Termination

- Define a decrementing function
  - Function maps variable(s) in program to a number
  - Show that value of function starts out >= 0
  - Show that value is decreased each time loop body is executed
  - Show that loop is exited when value is <= 0

Is "decreased each time" strong enough to guarantee termination?

Reasoning About Loop Termination



Zeno of Elea: 495BC – 425 BC

- Zeno's paradox
  - Achilles gives a tortoise a head start in a race
  - Both run at different constant speeds
  - By time Achilles reaches tortoise's starting point, tortoise has moved further distance
  - Repeat argument
  - Thus, Achilles can never catch tortoise

Decrementing function should be decreased each time in a way that guarantees that it reaches 0 in a finite number of steps

Approximation Algorithms

```
x = 36
epsilon = 0.01
numGuesses = 0
ans = 0.0
increment = 0.0001
while abs(ans**2 - x) >= epsilon:
    ans += increment
    numGuesses += 1
print('numGuesses =', numGuesses)
if abs(ans**2 - x) >= epsilon:
    print('Failed on square root of', x)
else:
    print(ans, 'is close to square root of', x)
```

Function: map ans to
abs(ans**2 - x) – epsilon

Initial value > 0?

Yes: x - epsilon

Decremented by positive amount each iteration?

Yes: from x-ans**2
to x – (ans+increment)**2

Approximation Algorithms

```
x = 36
epsilon = 0.01
numGuesses = 0
ans = 0.0
increment = 0.0001
while abs(ans**2 - x) >= epsilon:
    ans += increment
    numGuesses += 1
print('numGuesses =', numGuesses)
if abs(ans**2 - x) >= epsilon:
    print('Failed on square root of', x)
else:
    print(ans, 'is close to square root of', x)
```

Hint: this test is not monotonic in ans

Does decrementing function always eventually make this true?

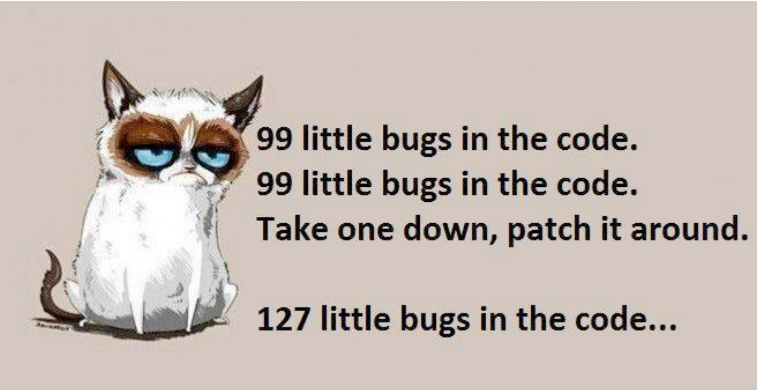
Will this test ever return True?

We should run it, and check

Some Observations

- Didn't find 6
- Took about 60,000 guesses
- Let's try:
  - 24
  - 2
  - 12345
  - 54321

Let’s Debug It

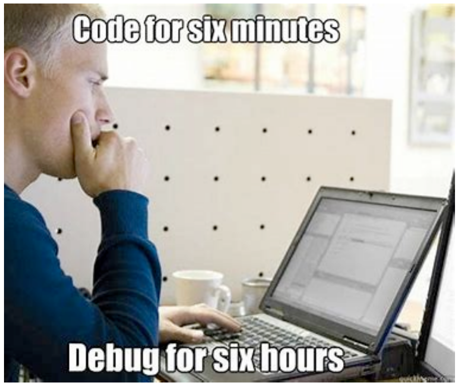



```
x = 54321
epsilon = 0.01
numGuesses = 0
ans = 0.0
increment = 0.0001
while abs(ans**2 - x) >= epsilon:
    ans += increment
    numGuesses += 1
    if numGuesses%100000 == 0:
        print('Current guess =', ans)
        print('Current guess**2 - x =', abs(ans*ans - x))
print('numGuesses =', numGuesses)
if abs(ans**2 - x) >= epsilon:
    print('Failed on square root of', x)
else:
    print(ans, 'is close to square root of', x)
```

Some Observations

- Decrementing function eventually starts incrementing
  - So didn't exit loop as expected
- We have over-shot the mark
  - I.e., we jumped from a value too far away but too small to one too far away but too large
- We didn't account for this possibility when writing the loop
- Let's fix that

Let's Debug It



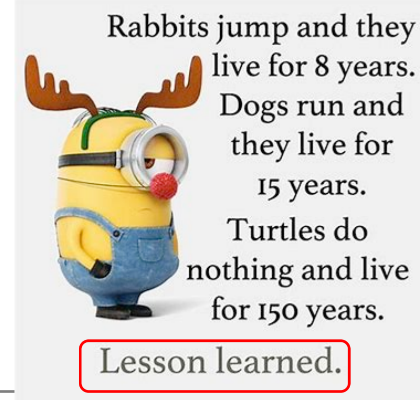

```
x = 54321
epsilon = 0.01
numGuesses = 0
ans = 0.0
increment = 0.0001
while abs(ans**2 - x) >= epsilon and ans**2 <= x:
    ans += increment
    numGuesses += 1
    if numGuesses%50000 == 0:
        print('Current guess =', ans)
        print('Current guess**2 - x =',
                abs(ans*ans - x))
print('numGuesses =', numGuesses)
if abs(ans**2 - x) >= epsilon:
    print('Failed on square root of', x)
else:
    print(ans, 'is close to square root of', x)
```

Some Observations

- Decrementing function eventually starts incrementing
- We have over-shot the mark
- We didn't account for this possibility when writing the loop
- Let's fix that
- Now it stops, but reports failure, because it has over-shot the answer
- Let's try resetting increment to 0.00001

How many iterations of loop?

Lessons Learned in Approximation Algorithms



- Need to be careful that looping mechanism doesn't jump over exit test and loop forever
- Tradeoff exists between efficiency of algorithm and accuracy of result
- Need to think about how close an answer we want when setting parameters of algorithm
- To find a good answer, this method can be painfully slow
  - Is there a faster way that still gets good answers?

Five Minute Break

Chance to Win Bucks

- Suppose I attach a hundred dollar bill to a particular page in the text book
- If you can guess page in 8 or fewer guesses, you get the "Benjamin"
- If you fail, you lose a late day
- Would you want to play?
  - Hint: the book is 447 pages long

Your chances are about 1 in 56

- Now suppose on each guess I told you whether you were correct, or too low or too high
- Would you want to play in this case?

Your chances are about 1 in 3

$$447 \xrightarrow{1} 223 \xrightarrow{2} 111 \xrightarrow{3} 55 \xrightarrow{4} 27 \xrightarrow{5} 13 \xrightarrow{6} 6 \xrightarrow{7} 3$$

Bisection Search

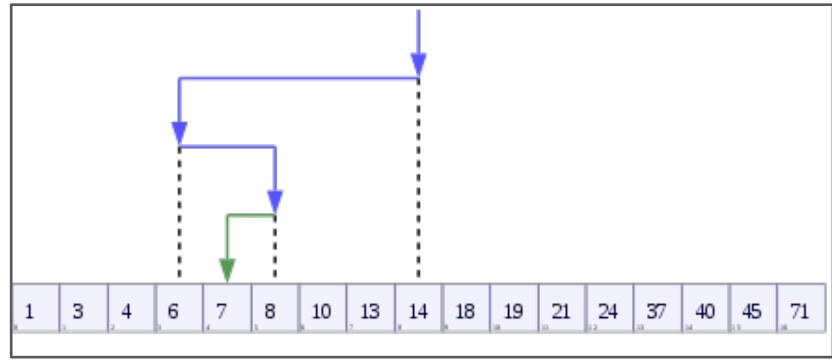
- Suppose we are given a problem where there is an inherent order to the range of possible answers, and that there is a maximum and minimum possible so that the range of possible answers forms a coherent interval
- Thus we know answer lies within some interval
  - Guess midpoint of interval
  - If not answer, then check if answer is greater than or less than midpoint
  - Change interval
  - Repeat
- Process cuts set of things to check in half at each stage
  - Exhaustive search reduces set of possible answers from N to N-1 on each step; bisection search reduces from N to N/2

Log Growth Is Better

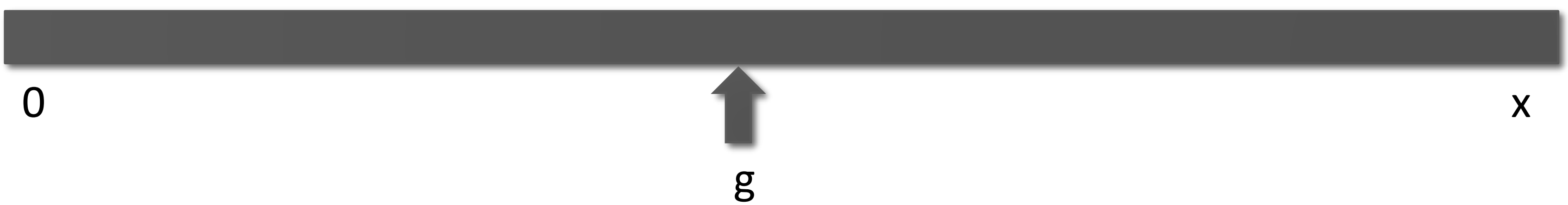
- Process cuts set of things to check in half at each stage
  - Characteristic of a logarithmic growth
- We can replace the algorithm that is linear in the number of possible guesses with one is that logarithmic on the number of possible guesses
  - This should be much more efficient

We will see discussion of relative costs of different algorithms in a few weeks

Bisection Search

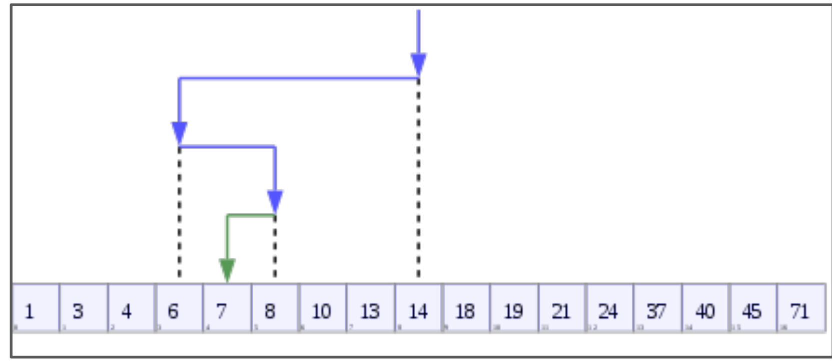


- Suppose we are looking for square root of x, so we know that the answer lies between 0 and x
- Rather than exhaustively trying things starting at 0, suppose instead we pick a number in the middle of this range

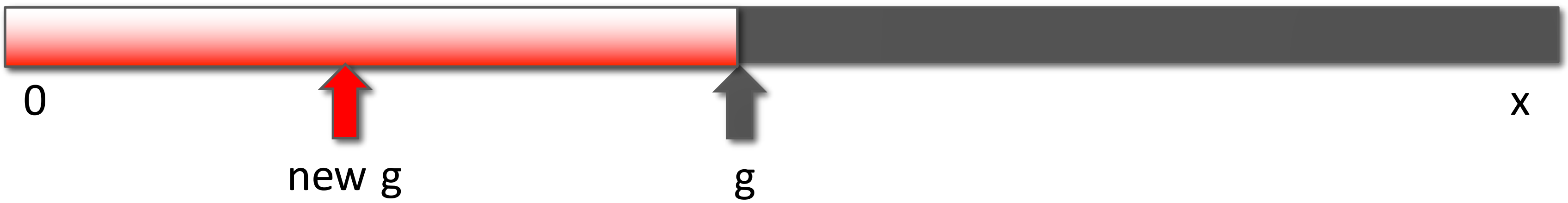


- If we are lucky, this answer is close enough

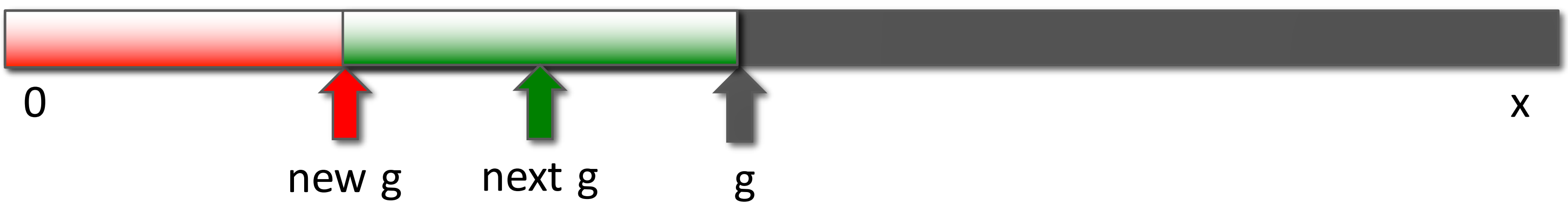
Bisection Search



- If not close enough, is guess too big or too small?
- If g**2 > x, then know g is too big; so now search



- And if, for example, this new g is such that g**2 < x, then know too small; so now search



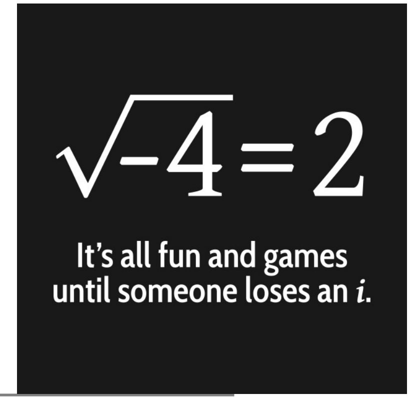
- At each stage, reduce range of values to search by half

Replace algorithm that is linear in the number of possible guesses with one is that logarithmic in the number of possible guesses

An analogy

- Suppose we forced you to sit in alphabetical order in 26-100, from front left corner to back right corner
- To find a particular student, I could ask the person in the middle of the hall their name
- Based on the response, I can either dismiss the back half or the front half of the entire hall
- And I repeat that process until I find the person I am seeking

Fast Square Root



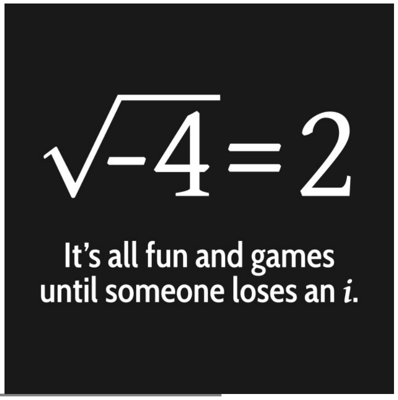

```
x = 54321
epsilon = 0.01
numGuesses = 0
low = 0.0
high = x
ans = (high + low)/2
while abs(ans**2 - x) >= epsilon:
    print('low = ' + str(low) + ' high = ' + str(high)\
          + ' ans = ' + str(ans))
    numGuesses += 1
    if ans**2 < x:
        low = ans
    else:
        high = ans
    ans = (high + low)/2.0
print('numGuesses = ' + str(numGuesses))
print(str(ans) + ' is close to square root of ' + str(x))
```

Log Growth Is Better!

- Brute force search for root of 54321 took over 23M guesses
- With bisection search, reduced to 30 guesses!
- We'll spend more time on this later, but we say the brute force method is **linear** in size of problem, because number to steps grows linearly as we increase problem size
- Bisection search is **logarithmic** in size of problem, because number of steps grows logarithmically with problem size
  - search space (if finding root of x, with step of size a, then N = x/a)
    - first guess: N/2
    - second guess: N/4
    - k^{th} guess: $N/2^k$
  - done when $N/2^k$ is 1; or in roughly $k = \log_2 N$ steps

Bisection Search: Cube Root

```
cube = 27
epsilon = 0.01
numGuesses = 0
low = 0
high = cube
ans = (high + low)/2.0
while abs(ans**3 - cube) >= epsilon:
    if ans**3 < cube :
        low = ans
    else:
        high = ans
    ans = (high + low)/2.0
    numGuesses += 1
print('numGuesses =', numGuesses)
print(ans, 'is close to the cube root of', cube)
```

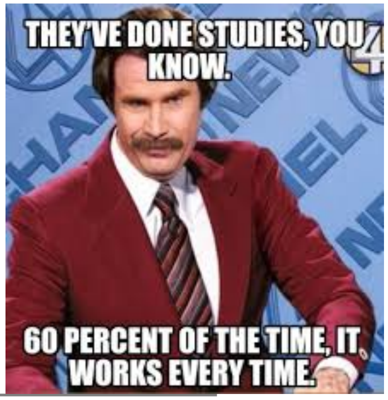


Fast Square Root

```
x = 0.5
epsilon = 0.01
numGuesses = 0
low = 0.0
high = x
ans = (high + low)/2
while abs(ans**2 - x) >= epsilon:
    print('low = ' + str(low) + ' high = ' + str(high)\
          + ' ans = ' + str(ans))
    numGuesses += 1
    if ans**2 < x:
        low = ans
    else:
        high = ans
    ans = (high + low)/2.0
print('numGuesses = ' + str(numGuesses))
print(str(ans) + ' is close to square root of ' + str(x))
```

For what values of x does this work?

Does it always work?



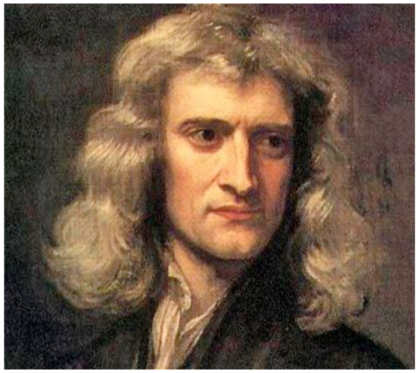
- Try running code for x such that 0 < x < 1
- If x < 1, we are searching from 0 to x but know square root is greater than x and less than 1
- Modify the code to choose the search space depending on value of x
- As we will see in a later lecture, careful thought about test cases is important in ensuring that algorithm performs as expected on all legal inputs

```python
x = 0.5
epsilon = 0.01
numGuesses = 0
if x >= 1:
    low = 1.0
    high = x
else:
    low = x
    high = 1.0
ans = (high + low)/2

while abs(ans**2 - x) >= epsilon:
    print('low = ' + str(low) + ' high = ' + str(high)\
          + ' ans = ' + str(ans))
    numGuesses += 1
    if ans**2 < x:
        low = ans
    else:
        high = ans
    ans = (high + low)/2.0
print('numGuesses = ' + str(numGuesses))
print(str(ans) + ' is close to square root of ' + str(x))
```

Some Observations

- Bisection search radically reduces computation time – being smart about generating guesses is important
- Search space gets smaller quickly at the beginning and then more slowly (in absolute terms, but not as a fraction of search space) later
  - Can see tradeoff between accuracy and speed
- Works on problems with "ordering" property – value of function being solved varies monotonically with input value
  - Here function is `ans**2`; which grows as `ans` grows
- Can we do this even more efficiently?

Newton-Raphson

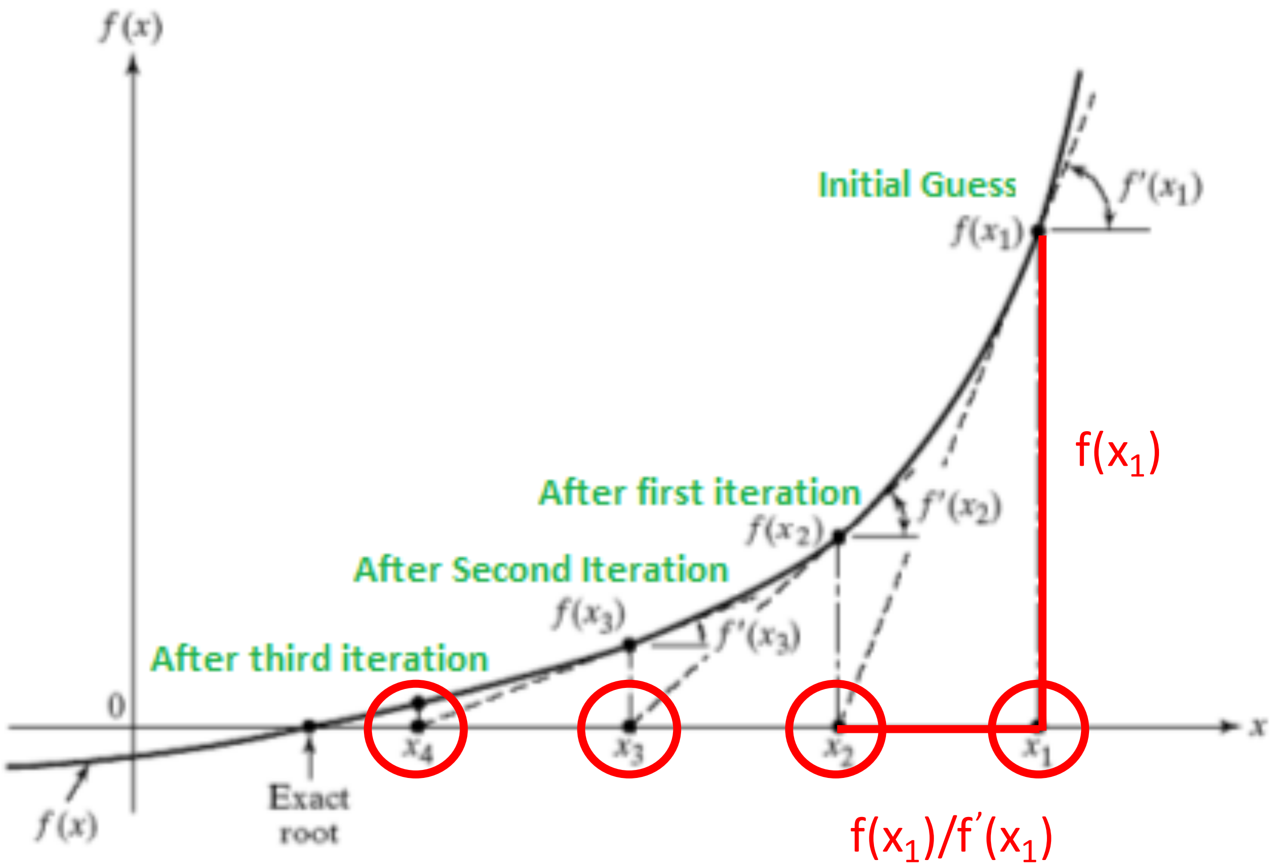


- General approximation algorithm to find roots of a polynomial in one variable

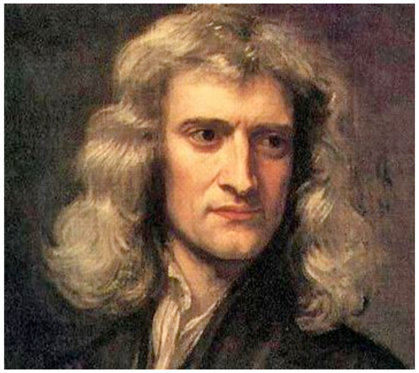
$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

- Want to find r such that f(r) = 0
- For example, to find the square root of 24, find the root of $f(x) = x^2 - 24$
- Newton developed method in 1685; method created sequence of polynomials, whose final solution is the desired root
- Raphson developed a much cleaner method that found successive approximations to the root, published in 1690
- We really use Raphson's method, but Newton (being much more famous) also gets credit (often method is just referred to as Newton's method, even though we use Raphson's version)

Intuition for Newton-Raphson



Newton-Raphson



- General approximation algorithm to find roots of a polynomial in one variable

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

- Raphson showed that if g is an approximation to the root, then

$$g - f(g)/f'(g)$$

is a better approximation; where f' is derivative of f

Newton-Raphson Root Finder

- Simple case: $cx^2 + k$
- First derivative: $2cx$
- So if polynomial is $x^2 - k$, then derivative is $2x$
- Newton-Raphson says given a guess g for root of k, a better guess is

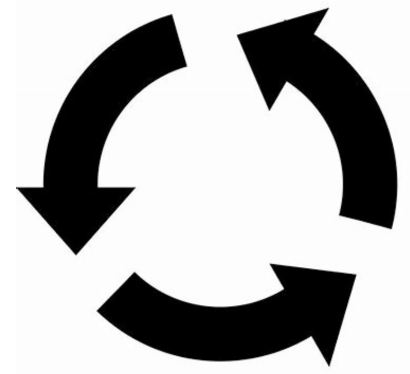
$$g - (g^2 - k)/2g$$

Newton-Raphson Root Finder

- Another way of generating guesses, which we can check; very efficient

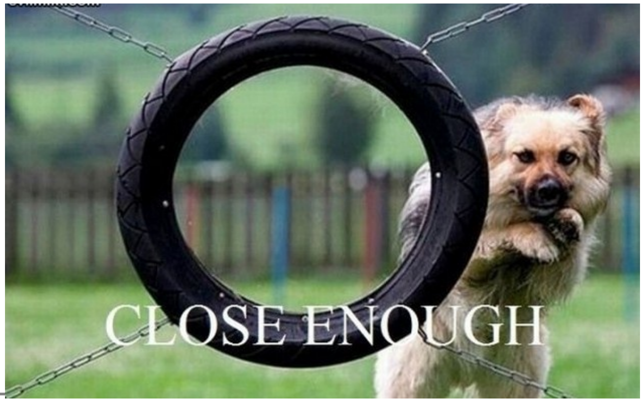
```
epsilon = 0.01
y = 24.0
guess = y/2.0
numGuesses = 0

while abs(guess*guess - y) >= epsilon:
    numGuesses += 1
    guess = guess - (((guess**2) - y)/(2*guess))
print('numGuesses = ' + str(numGuesses))
print('Square root of ' + str(y) + ' is about ' + str(guess))
```

Iterative Algorithms



- Guess and check methods build on reusing same code
  - Use a looping construct to generate guesses, then check and continue
- Generating guesses
  - Exhaustive enumeration
  - Bisection search
  - Newton-Raphson (for root finding)

Summary



- For many problems, cannot find exact answer; need to seek “good enough” answer using approximations
- When testing floating point numbers (e.g., as part of an approximate answer), important to understand how computer represents these in binary, and why we use “close enough” and not “==”
- Bisection search is a great way to reduce a linear algorithm to a logarithmic one