

# NUMBERS, APPROXIMATIONS, and BISECTION

(download slides and .py files to follow along)

---

6.0001 LECTURE 3

Eric Grimson

# Last Time

---

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

# Today

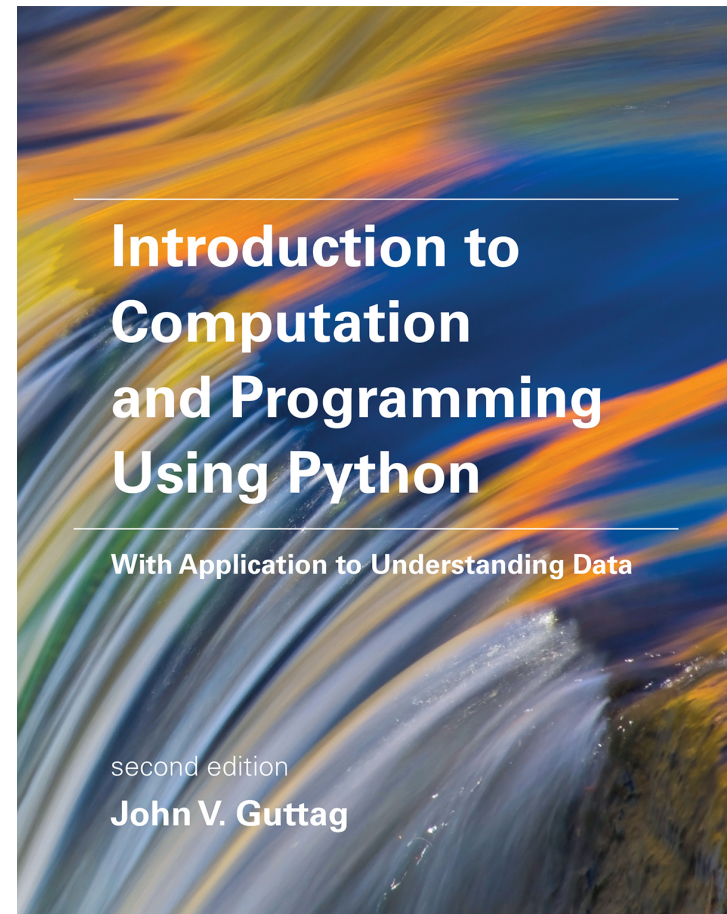
---

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

# Assigned Reading



- Today:
  - Sections 3.3 – 3.5
- Next lecture:
  - Section 4.1 – 4.3
  - Sections 5.1 – 5.5



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



# Numbers in Python



- **int:** integers, like the ones you learned about in elementary school
- **float:** ~~reals, like the ones you learned about in middle school~~



# A Closer Look at Floats

- Python (and every other programming language) uses “floating point” to approximate real numbers
- The term “floating point” refers to way these numbers are stored in computer
- Approximation usually doesn’t matter

Like government contractors, just need to be “close enough”

But “usually doesn’t matter” is another way of saying what?

# Actually It Matters!

---

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

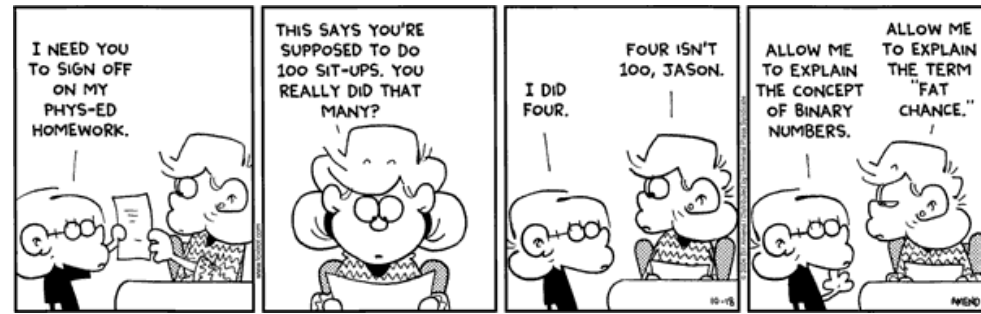
Note:  $x += 0.1$  is the same as  
 $x = x + 0.1$

# Why?

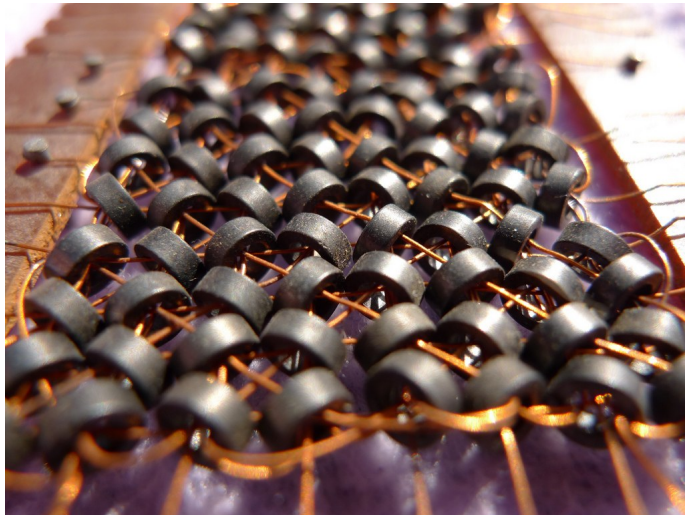
---

- 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



Core  
memory

1960 (\$0.62/bit) - 1975 (\$0.008/bit)

What does a bit of dynamic RAM cost today?

About \$8.25 per Gigabyte! Or \$0.000000001/bit

# Binary Numbers

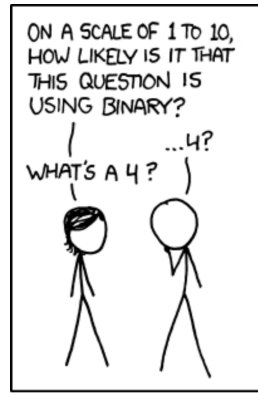
- Base 10 representation of an integer
  - sum of powers of 10, scaled by integers from 0 to 9

$$\begin{aligned}1507 &= 1*10^3 + 5*10^2 + 0*10^1 + 7*10^0 \\ &= 1000 + 500 + 7\end{aligned}$$

- Binary representation is same idea in base 2
  - sum of powers of 2, scaled by integers from 0 to 1

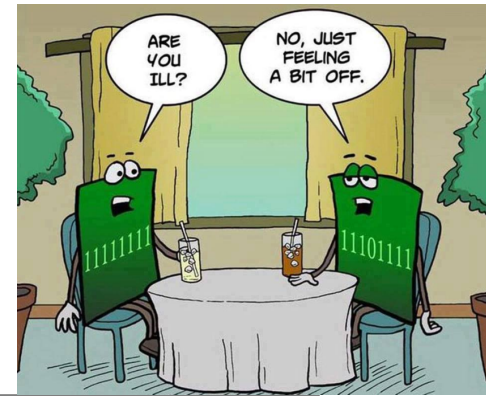
$$\begin{aligned}1507_{10} &= 1*2^{10} + 1*2^8 + 1*2^7 + 1*2^6 + 1*2^5 + 1*2^1 + 1*2^0 \\ &= 1024 + 256 + 128 + 64 + 32 + 2 + 1 \\ &= 10111100011_2\end{aligned}$$

# Converting Decimal Integer to Binary



- We input integers in decimal, computer needs to convert to binary
- 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 ( $x \% 2$ ), that gives us the last binary bit
- If we then integer divide  $x$  by 2 ( $x // 2$ ), all the bits get shifted right
  - $x // 2 = 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 = 1001$
- Keep doing successive divisions; now remainder gets next bit, and so on
- Let's us convert to binary form

# Doing this in Python



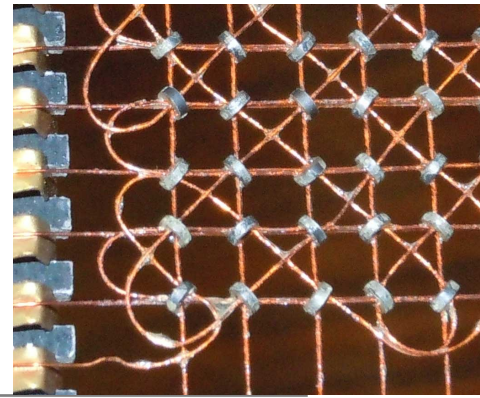
```
if num < 0:
    isNeg = True
    num = abs(num)
else:
    isNeg = False

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

if isNeg:
    result = '-' + result
```

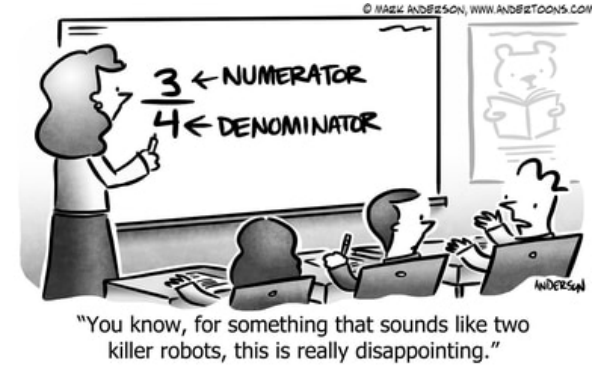


# Hardware Implementation



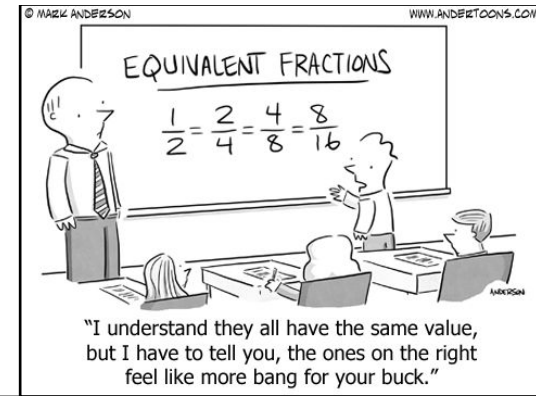
- Computer hardware is built around methods that can efficiently store information as 0's or 1's (a voltage is “high” or “low”; or a magnetic spin is “up” or “down”) and can efficiently perform arithmetic operations on such representations
- Fine for integer arithmetic
- But what about numbers with fractional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

# Fractions



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

# What About Fractions?



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

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

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

Find power  
of to make  
integer

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

Convert to  
int

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

Encode as  
binary  
number

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

Pad front  
with 0's

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

Insert  
decimal

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

# But ...



- If there is no integer  $p$  such that  $x \cdot (2^p)$  is a whole number, then internal representation is **always** an approximation
- **And** 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
- Hence the floating point conversion will work precisely for numbers like  $3/8$  but not for  $1/10$  – one has a power of 2 that converts to whole number, other 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**
- How how about the decimal representation 0.1
  - In base 10:  $1 * 10^{-1}$
  - In base 2: ? **0.0001100110011001100110011...**

**Any finite number of bits gives us an approximation**

# And the point is?



- 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


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

# Floating Point Numbers

- Floating point is a pair of integers
  - Significant digits and base 2 exponent
  - $(1, 1) \rightarrow 1 * 2^1 \rightarrow 10_2 \rightarrow 2.0$
  - $(1, -1) \rightarrow 1 * 2^{-1} \rightarrow 0.1_2 \rightarrow 0.5$
  - $(125, -2) \rightarrow 125 * 2^{-2} \rightarrow 11111.01_2 \rightarrow 31.25$
- 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  $2 * 10^{-10}$

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





...and i should care,  
why?

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** 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

# Effect of approximation on our algorithms?

---

- Exact answer may not be accessible
- Need to find ways to get “good enough” answer
  - Including ways to decide when answer is “good enough” or “close enough” to ideal answer
- Need ways to deal with fact that exhaustive enumeration can’t test every possible value, since set of possible answers is in principle infinite

# Finding Roots

- Last lecture we looked at using exhaustive enumeration/guess and 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
- 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; and exhaustive search is infinite



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  $|r^2 - x| < \epsilon$
- 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  $g*g$  is close enough to  $x$  (within  $\epsilon$ )
  - Continue until get answer close enough to actual answer
- Looking at all possible **values  $g + k*a$**  for integer values of  **$k$**  – so similar to exhaustive enumeration
  - But cannot test all possibilities as infinite

# 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  $\rightarrow$  slower program, but more likely to get good answer
- Increasing epsilon  $\rightarrow$  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*



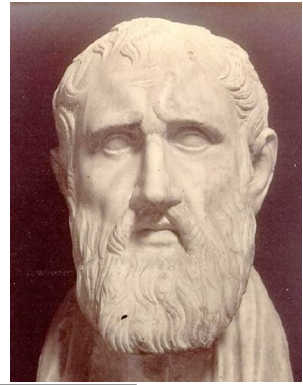
# Reasoning About Loop Termination

- Define a decremending function
  - Function maps variable(s) in program to a number
  - Show that value of function starts out  $\geq 0$
  - Show that value is decreased each time loop body is executed
  - Show that loop is exited when value is  $\leq 0$



Is “decreased each time” strong enough to guarantee termination?

# Reasoning About Loop Termination



- 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  
 $\text{abs}(\text{ans}^2 - x) - \text{epsilon}$

Do we eventually exit?

Initial value > 0?

Yes:  $x - \text{epsilon}$

Decrement by positive amount  
each iteration?

Yes: from  $x - \text{ans}^2$   
to  $x - (\text{ans} + \text{increment})^2$

# 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)
```

Does decrementing function  
always eventually make this  
true?

Will 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



99 little bugs in the code.  
99 little bugs in the code.  
Take one down, patch it around.  
  
127 little bugs in the code...

```
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)
```



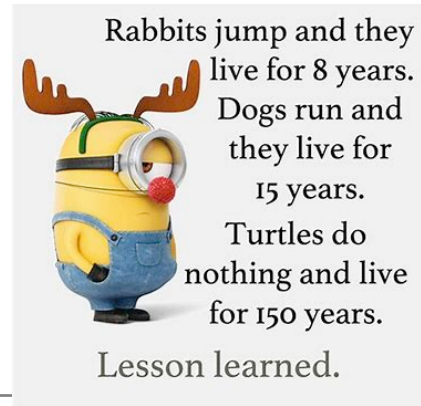
# Some Observations



- 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 get 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
- Hint: the book is 447 pages long
- Would you want to play?
- 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 56*

*Your chances are about 1 in 3*



# Bisection Search

---

- We are given a problem where there is an inherent order to the range of possible answers, and thus range of answers forms a coherent interval
- Suppose 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  $N/2$

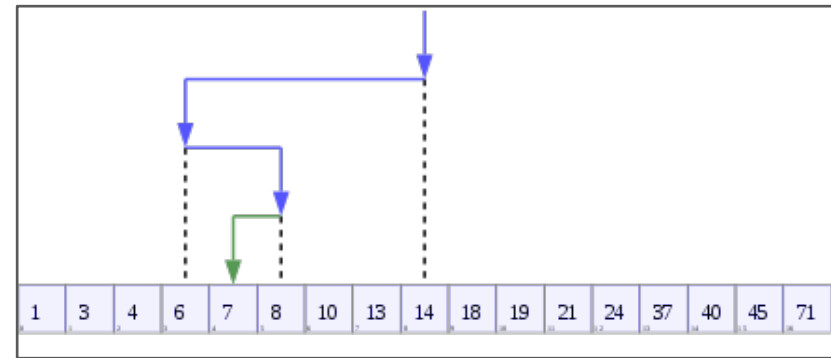
# Log Growth Is Better



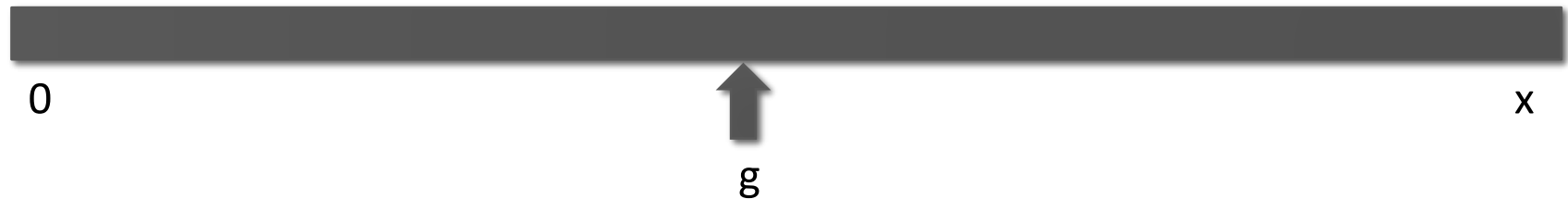
- Process cuts set of things to check in half at each stage
  - Characteristic of a logarithmic growth
- We can replace the algorithm that is linear in the number of possible guesses with one is that logarithmic on the number of possible guesses
  - This should be much more efficient

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

# Bisection Search

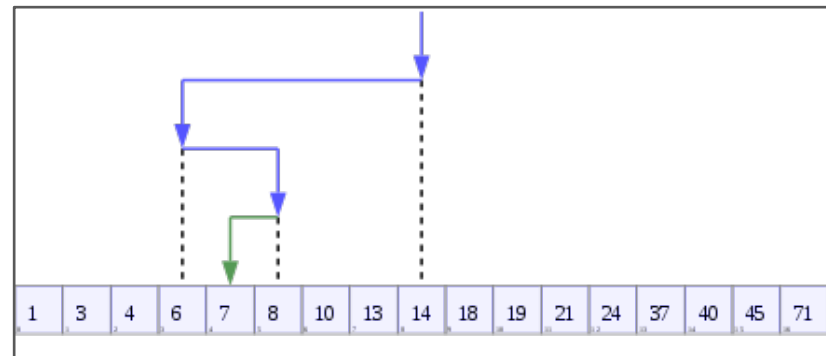


- Suppose we know that the answer lies between 0 and  $x$
- Rather than exhaustively trying things starting at 0, suppose instead we pick a number in the middle of this range

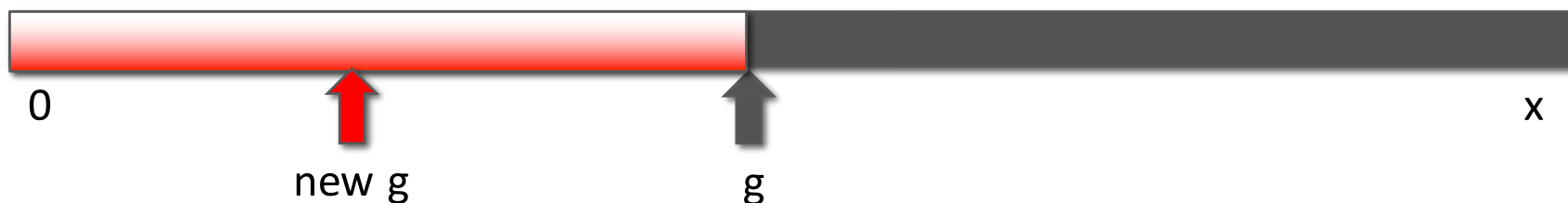


- If we are lucky, this answer is close enough

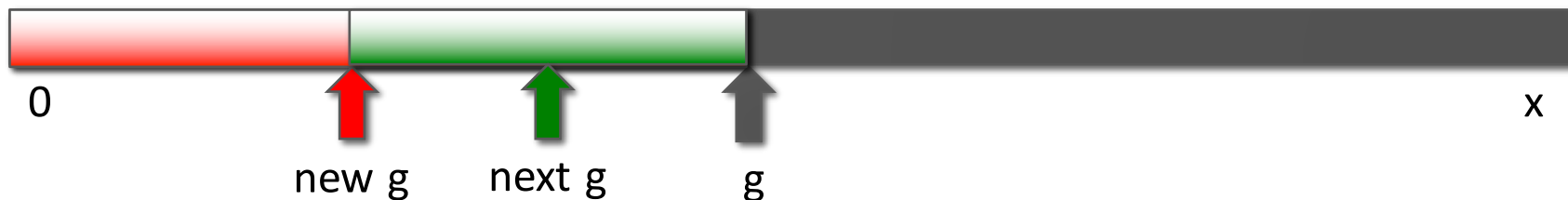
# Bisection Search



- 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



$$\sqrt{-4}=2$$

It's all fun and games  
until someone loses an *i*.

# 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))
```

*For what values of x does this work?*

# Bisection Search – Cube Root

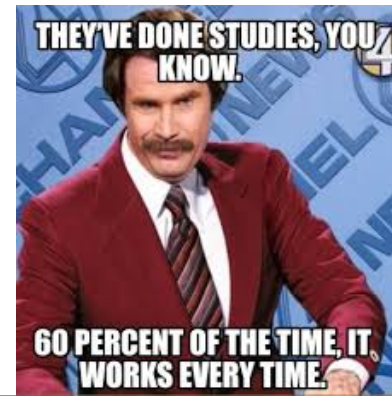
---

```
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)
```

# 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
    - first guess:  $N/2$
    - second guess:  $N/4$
    - $k^{\text{th}}$  guess:  $N/2^k$
  - guess converges on the order of  $\log_2 N$  steps



# 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  $x$

```

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

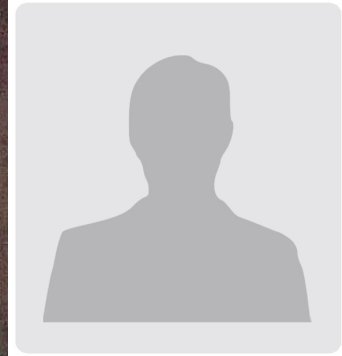
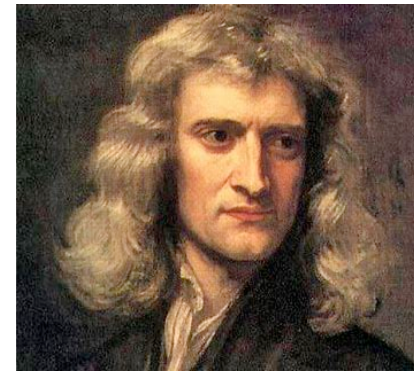
while abs(ans**2 - x) >= epsilon:
    print('low = ' + str(low) + ' high = ' + str(high)\
          + ' ans = ' + str(ans))
    numGuesses += 1
    if ans**2 < x:
        low = ans
    else:
        high = ans
    ans = (high + low)/2.0
print('numGuesses = ' + str(numGuesses))
print(str(ans) + ' is close 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
- Works on problems with “ordering” property – value of function being solved varies monotonically with input value
  - Here function is  $\text{ans}^2$ ; which grows as  $\text{ans}$  grows



# Newton-Raphson

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

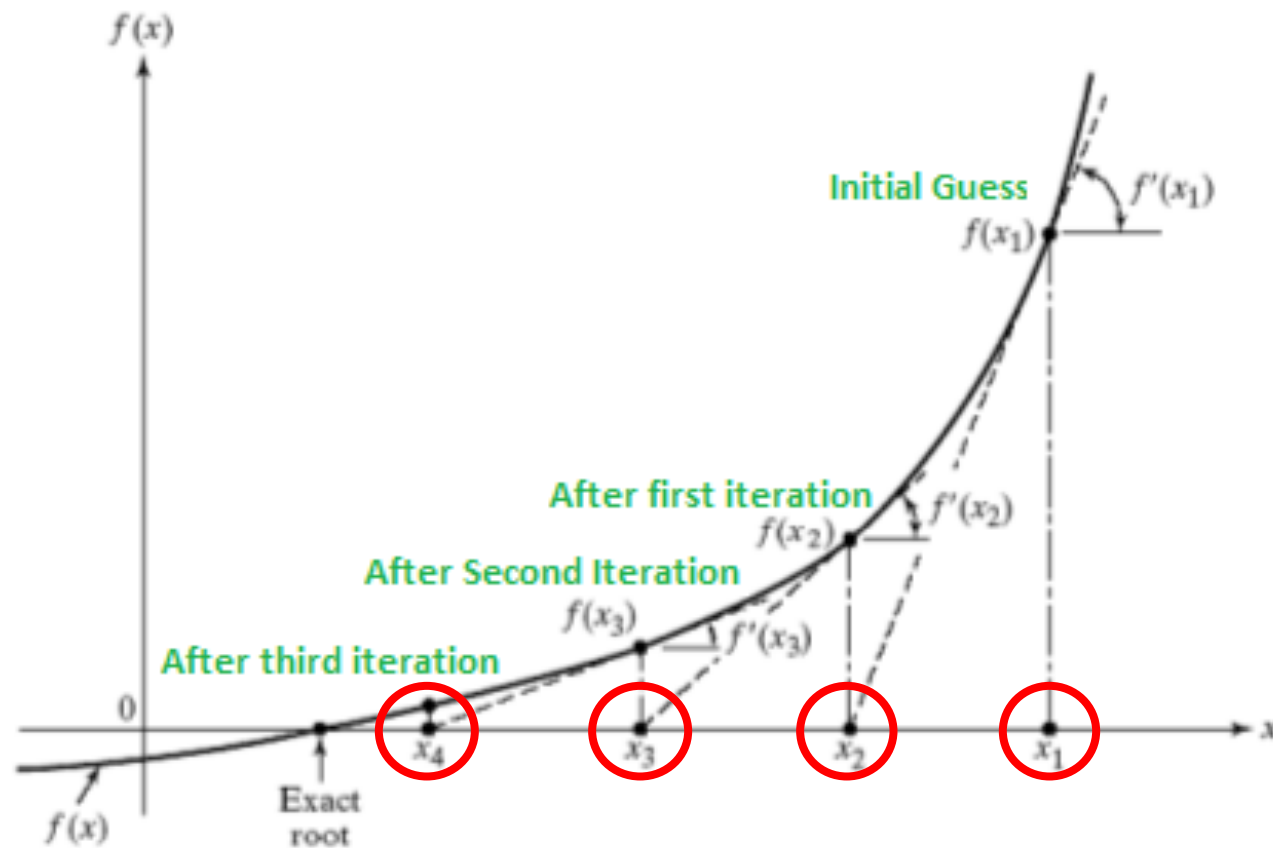
$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

- Want to find  $r$  such that  $p(r) = 0$
- For example, to find the square root of 24, find the root of  $p(x) = x^2 - 24$
- Newton and Raphson showed that if  $g$  is an approximation to the root, then

$$g - p(g)/p'(g)$$

is a better approximation; where  $p'$  is derivative of  $p$

# Intuition for Newton-Raphson





# 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
```

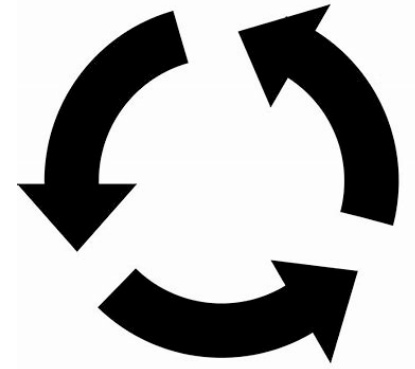
```
while abs(guess*guess - y) >= epsilon:
```

```
    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