# FLOATS and APPROXIMATION METHODS

(download slides and .py files to follow along)

6.100L Lecture 5

Ana Bell

# OUR MOTIVATION FROM LAST LECTURE

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

0.9999999999999999 == 1.0

# INTEGERS

- Integers have straightforward representations in binary
- The code was simple (and can add a piece to deal with negative numbers)

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

Set a negative flag and handle it

# FRACTIONS

# FRACTIONS

- 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
  - $a*2^{-1} + b*2^{-2} + c*2^{-3}$
- 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
  - $f = 0.5a + 0.25b + 0.125c + 0.0625d + 0.03125e + \ldots$

# WHAT ABOUT FRACTIONS?

- How might we find that representation?
- In decimal form: $3/8 = 0.375 = 3*10^{-1} + 7*10^{-2} + 5*10^{-3}$

- **Recipe idea**: 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 $0.011_2$

# BUT…

- If there is **no integer p such that $x*(2^p)$ 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
- Floating point conversion works:
  - Precisely for numbers like 3/8
  - But not for 1/10
  - **One has a power of 2 that converts to whole number, the other doesn't**

# TRACE THROUGH THIS ON YOUR OWN
# Python Tutor LINK

% grabs the decimal part only
e.g. 1.1%1 gives 0.1

```
x =0.625
```

```
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

```
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, same as prev slide

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

Pad front with 0's, i.e. shift right

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

Insert decimal

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

# 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: ?

0.00011001100110011001100110011...

Infinite!

# THE POINT?

- 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
  - The idea was to try and convert a real number to an int by multiplying the real with some multiple of 2 to get an int
  - Sometimes there is no such power of 2!
  - Have to somehow **approximate the potentially infinite binary sequence** of bits needed to represent them

# FLOATS

# STORING FLOATING POINT NUMBERS #.#

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

- Floating point is a pair of integers
  - Significant digits and 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

  125 is 1111101 then move the decimal point over 2

# USE A FINITE SET OF BITS TO REPRESENT A POTENTIALLY INFINITE SET OF BITS

- The maximum number of significant digits governs the precision with which numbers can be represented
- Most modern computers use **32 bits** to represent significant digits
- If a number is represented with more than 32 bits in binary, the **number will be rounded**
  - Error will be at the 32nd bit
  - **Error will only be on order of $2*10^{-10}$**

$2^{-32}$ is approx. $10^{-10}$
pretty small number, isn't it?

# 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

# 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

# APPROXIMATION METHODS

# LAST LECTURE

- Guess-and-check provides a **simple algorithm** for solving problems
- When set of **potential solutions is enumerable**, exhaustive enumeration guaranteed to work (eventually)
- It's a limiting way to solve problems
  - Increment is **usually an integer but not always**. i.e. we just need some pattern to give us a finite set of enumerable values
  - Can't give us an approximate solution to varying degrees

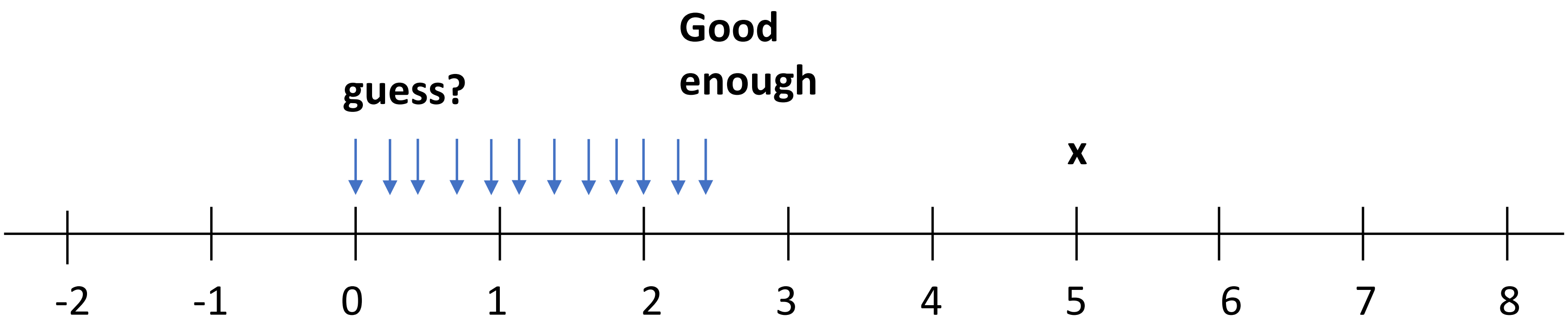
# BETTER than GUESS-and-CHECK

- Want to find an **approximation to an answer**
  - Not just the correct answer, like guess-and-check
  - And not just that we did not find the answer, like guess-and-check

# EFFECT of APPROXIMATION on our ALGORITHMS?

- **Exact** answer may not be **accessible**
- Need to find ways to get **"good enough" answer**
  - Our answer is "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
- Floating point **approximation errors** are important to this method
  - Can't rely on equality!

# APPROXIMATE sqrt(x)



# FINDING ROOTS

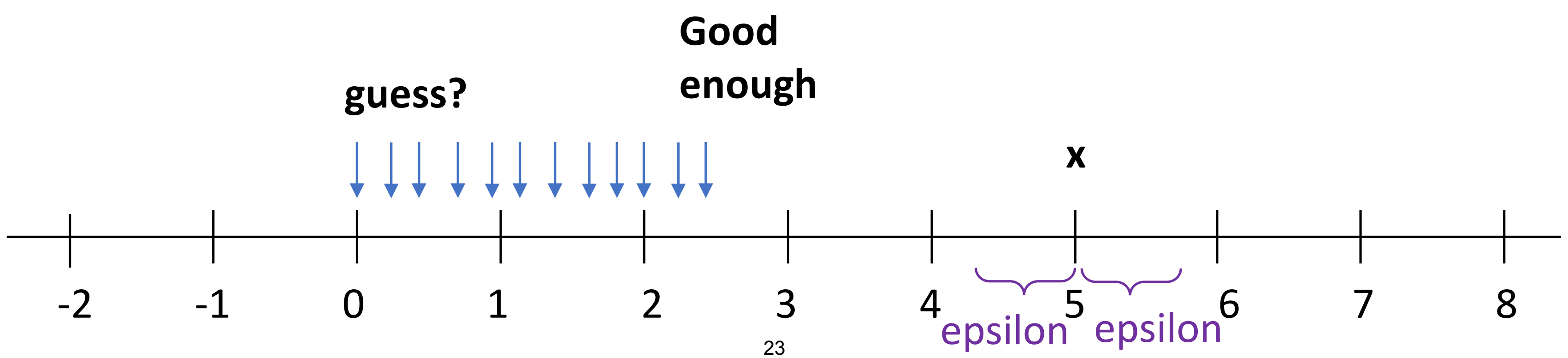
- Last lecture we looked at using exhaustive enumeration/guess and check methods to find the **roots of perfect squares**
- Suppose we want to find the square root of any positive integer, or any positive number
- Question: What does it mean to find the square root of x?
  - Find an r such that r*r = 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**

# 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
  - Use epsilon: given x we want to find $r$ such that $|r^2$-x$|<\varepsilon$
- Algorithm
  - Start with guess **known to be too small** – call it `g`
  - Increment by a small value – call it `a` – to give a new guess `g`
  - Check if `g**2` is close enough to `x` (within $\varepsilon$)
  - 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

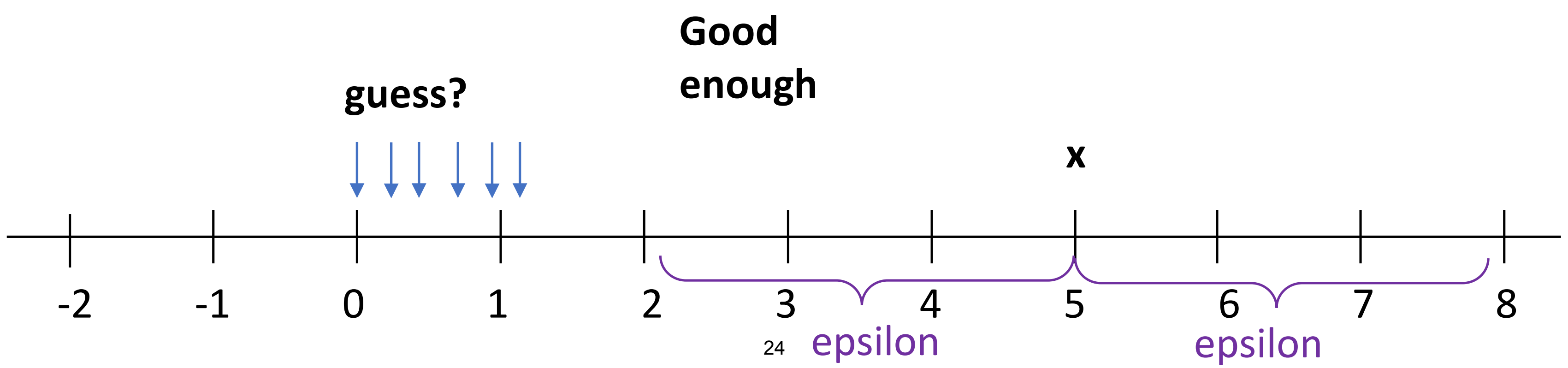
# APPROXIMATION ALGORITHM

- In this case, we have **two parameters** to set
  - **epsilon** (how close are we to answer?)
  - **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 good answer (and vice versa)



# APPROXIMATION ALGORITHM

- In this case, we have **two parameters** to set
  - **epsilon** (how close are we to answer?)
  - **increment** (how much to increase our guess?)
- Performance will vary based on these values
  - In speed
  - In accuracy
- **Increasing epsilon** → less accurate answer, but faster program (and vice versa)



# BIG IDEA

## Approximation is like guess-and-check except…

1) We increment by some small amount
2) We stop when close enough (exact is not possible)

# IMPLEMENTATION

```
x = 36
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.0001
```

```
while abs(guess**2 - x) >= epsilon:
    guess += increment
    num_guesses += 1
```

```
print('num_guesses =', num_guesses)
```

```
print(guess, 'is close to square root of', x)
```

Will this loop always terminate?

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

# OBSERVATIONS with DIFFERENT VALUES for x

- For x = 36
  - Didn’t find 6
  - Took about 60,000 guesses
- Let’s try:
  - 24
  - 2
  - 12345
  - 54321

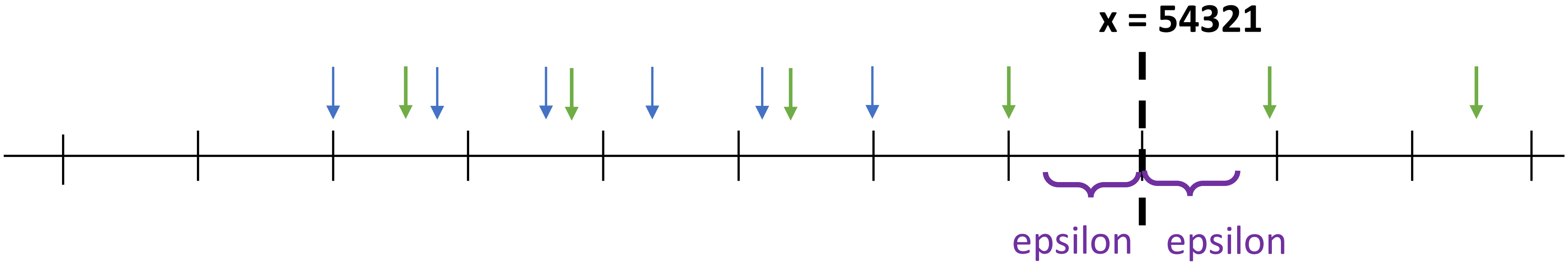
```python
x = 54321
epsilon = 0.01
numGuesses = 0
guess = 0.0
increment = 0.0001

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

Debugging print statements every 100000 times through the loop, showing guess and how far away from epsilon we are

# WE OVERSHOT the EPSILON!

- Blue arrow is the `guess`
- Green arrow is `guess**2`



# 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 FIX IT

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

Same condition as guess-and-check, stop when you go past the last reasonable guess

Exited b/c guess**2 > x

Exited b/c guess**2 is within eps

# BIG IDEA

It’s possible to overshoot the epsilon, so you need another end condition

# SOME OBSERVATIONS

- Now it stops, but **reports failure**, because it has over-shot the answer
- Let's try resetting increment to 0.00001
  - Smaller increment means **more values will be checked**
  - Program will be slower

# BIG IDEA

Be careful when comparing floats.

# LESSONS LEARNED in APPROXIMATION

- Can't use == to check an exit condition
- 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**?
  - **YES!** We will see it next lecture….

# SUMMARY

- Floating point numbers introduce challenges!
- They **can't be represented in memory exactly**
  - Operations on floats introduce tiny errors
  - Multiple operations on floats **magnify errors** :(
- Approximation methods use floats
  - Like guess-and-check except that
    (1) We use a float as an **increment**
    (2) We stop when we are **close enough**
  - **Never use == to compare floats** in the stopping condition
  - Be careful about **overshooting** the close-enough stopping condition