Y4 CEP - Introduction to Computational Thinking and Data Science

Lesson 0

Assumptions .

- proficient with basics of Python:
  - conditional statements
  - loops
- familiar with writing object – oriented programs in Python 3
  - class
  - attributes
  - methods
  - getter
  - setter

Expectations

- less about learning to program, more about dipping your toes into data science
- using mathematics and statistics together with programming to better understand “things”

How you will be graded

- Homework / in-class Tasks (30%)
  - Programming Tasks

- FSR (30%)
  - Individual Finding/Sharing/Reflecting
    - peer evaluation for sharing component (both evaluated)

- ~Weekly Quiz (40%)
  - pen and paper / may include some programming questions
  - short answer , True/False, MCQ , code writing
  - ~30 min before start of lesson

What is expected of you . . .

- revisit the PowerPoint Slides
- test the codes in the readings and slides
- work on the tasks (if any)
- delve deeper on your own

Topics to discuss . . .

- some simple numerical problems
- complexity, searching & sorting
- optimization problems
- graph-theoretic models
- stochastic thinking
- random walks
- Monte Carlo simulation
- confidence intervals
- sampling and standard error
- experimental data
- introduction to machine learning
- clustering
- classification — in total about 15~16 lessons
- FSR — about 2~3 lessons

Some Simple Numerical Problems

Lesson 1

finding cube root

GUESS-AND-CHECK

- the process below also called **exhaustive enumeration**

- given a problem…
- you are able to **guess a value** for solution
- you are able to **check if the solution is correct**
- keep guessing until find solution or guessed all values

GUESS-AND-CHECK – cube root

```
cube = 8

for guess in range(cube+1):

    if guess**3 == cube:

        print("Cube root of", cube, "is", guess)
```

GUESS-AND-CHECK – cube root

```
cube = 8
for guess in range(abs(cube)+1):
    if guess**3 >= abs(cube):
        break
if guess**3 != abs(cube):
    print(cube, 'is not a perfect cube')
else:
    if cube < 0:
        guess = -guess
    print('Cube root of '+str(cube)+' is '+str(guess))
```

APPROXIMATE SOLUTIONS

- **good enough** solution
- start with a guess and increment by some **small value**
- keep guessing if `|guess`^{3}`-cube| >= epsilon` for some **small epsilon**

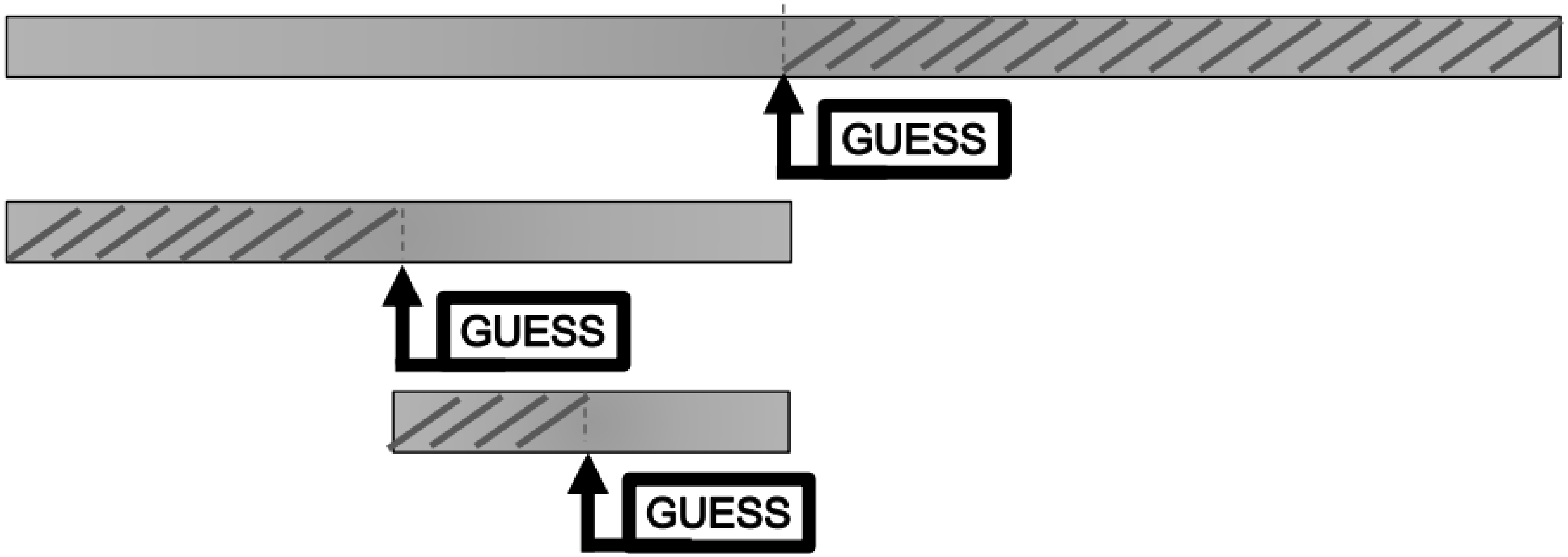
- decreasing increment size → slower program
- increasing epsilon → less accurate answer

APPROXIMATE SOLUTION – cube root

```
cube = 27
epsilon = 0.01
guess = 0.0
increment = 0.0001
num_guesses = 0
while abs(guess**3 - cube) >= epsilon  and guess <= cube :
    guess += increment
    num_guesses += 1
print('num_guesses =', num_guesses)
if abs(guess**3 - cube) >= epsilon:
    print('Failed on cube root of', cube)
else:
    print(guess, 'is close to the cube root of', cube)
```

BISECTION SEARCH

- half interval each iteration
- new guess is halfway in between
- to illustrate, let's play a game!



BISECTION SEARCH – cube root

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

BISECTION SEARCH CONVERGENCE

- search space
  - first guess: $N/2$
  - second guess: $N/4$
  - kth guess: $N/2^k$
- guess converges on the order of $\log_2 N$ steps
- bisection search works when value of function varies monotonically with input
- code as shown only works for positive cubes > 1 – why?
- challenges → modify to work with negative cubes!
  → modify to work with $x < 1$!

$x < 1$

- if $x < 1$, search space is 0 to x but cube root is greater than x and less than 1
- modify the code to choose the search space depending on value of x

SOME OBSERVATIONS

- Bisection search radically reduces computation time – being smart about generating guesses is important
- Should work well on problems with “ordering” property – value of function being solved varies monotonically with input value
    - Here function is g**2; which grows as g 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 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

NEWTON-RAPHSON

- 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, a better guess is

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

NEWTON-RAPHSON

- This gives us 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)

Practice/Exercise

Write a program that asks the user to enter an integer and prints two integers, root and pwr, such that 1 < pwr < 6 and root**pwr is equal to the integer entered by the user. If no such pair of integers exists, it should print a message to that effect.

Write a program that prints the sum of the prime numbers greater than 2 and less than 1000. Hint: you probably want to have a loop that iterates over the odd integers between 3 and 999.

Practice/Exercise

The empire state building is 102 stories high. A man wanted to know the highest floor from which he could drop an egg without the egg breaking. He proposed to drop an egg from the top floor. If it broke, he would go down a floor and try it again. He would do this until the egg did not break. At worst, this method requires 102 eggs. Implement a method that at worst uses seven eggs.

Add some code to the implementation of Newton-Raphson that keeps track of the number of iterations used to find the root. Use that code as part of a program that compares the efficiency of Newton-Raphson and bisection search.