BISECTION SEARCH

(download slides and .py files to follow along)

6.100L Lecture 6

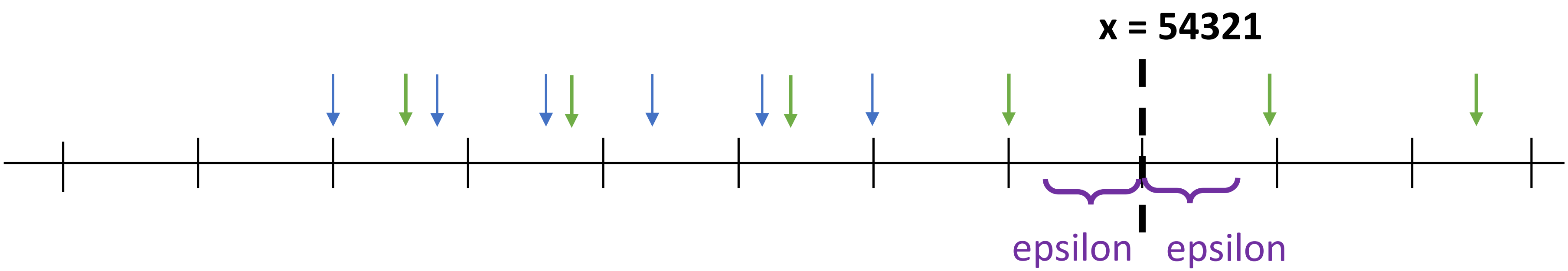
Ana Bell

LAST LECTURE

- 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** :(
- Guess-and-check enumerates ints one at a time as a solution to a problem
- **Approximation methods** enumerate using a float increment. Checking a solution is not possible. Checking whether a solution yields a **value within epsilon** is possible!

RECAP: SQUARE ROOT FINDING: STOPPING CONDITION with a BIG INCREMENT (0.01)

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



RECAP of APPROXIMATION METHOD TO FIND A “close enough” SQUARE ROOT

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

Stop when you are within epsilon, i.e. close enough

Stop when you go past the last reasonable guess

Exited the loop because further guesses are unreasonable

Exited the loop because guess**2 got close enough to x

BISECTION SEARCH

CHANCE to WIN BIG BUCKS

- Suppose I attach a hundred dollar bill to a particular page in the text book, 448 pages long
- If you can guess page in 8 or fewer guesses, you get big bucks
- If you fail, you get an F
- Would you want to play?

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

BISECTION SEARCH

- Apply it to **problems with an inherent order** to the range of possible answers
- Suppose we know answer lies within some interval
  - Guess **midpoint** of interval
  - If not the answer, 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 them from N to N-1 on each step
  - Bisection search reduces them 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
- **Algorithm comparison: guess-and-check vs. bisection search**
  - Checking answer on-by-one iteratively is linear in the number of possible guesses
  - Checking answer by guessing the halfway point is logarithmic on the number of possible guesses
  - Log algorithm is much **more efficient**

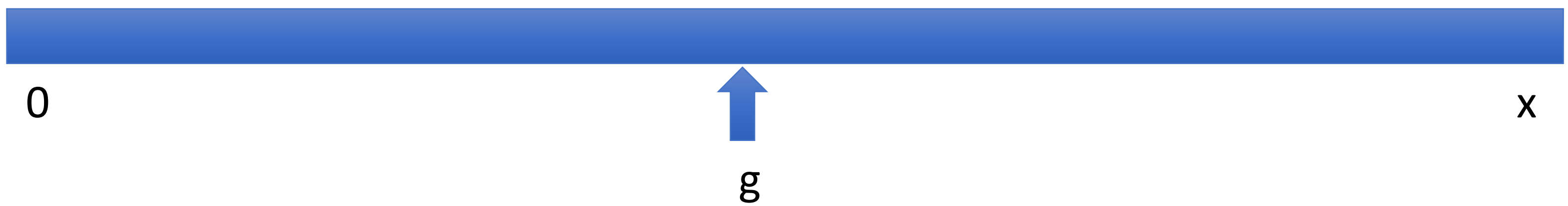
We will see discussion of relative costs of different algorithms in a few weeks

AN ANALOGY

- Suppose we forced you to sit in **alphabetical order** in class, 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

BISECTION SEARCH for SQUARE ROOT

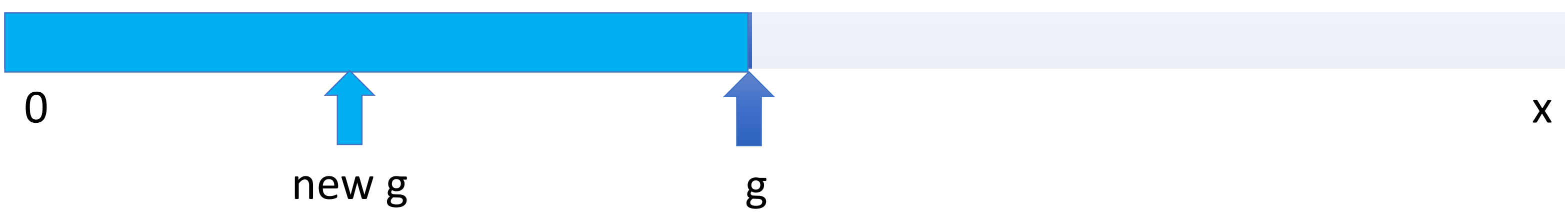
- Suppose we know that the **answer lies between 0 and x**
- Rather than exhaustively trying things starting at 0, suppose instead we **pick a number in the middle** of this range



- If we are lucky, this answer is close enough

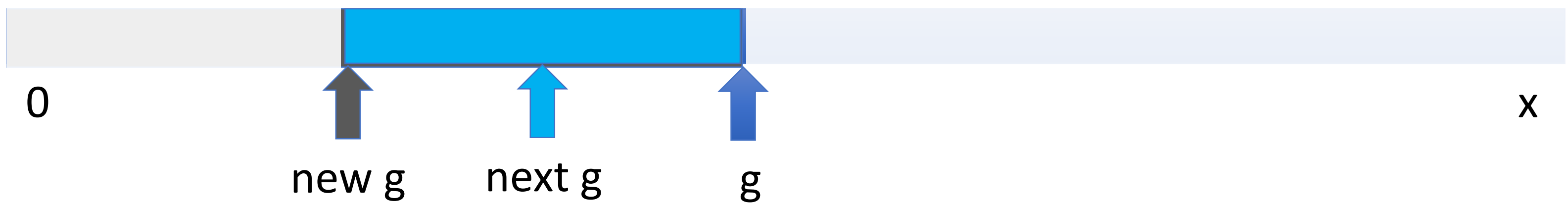
BISECTION SEARCH for SQUARE ROOT

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



BISECTION SEARCH for SQUARE ROOT

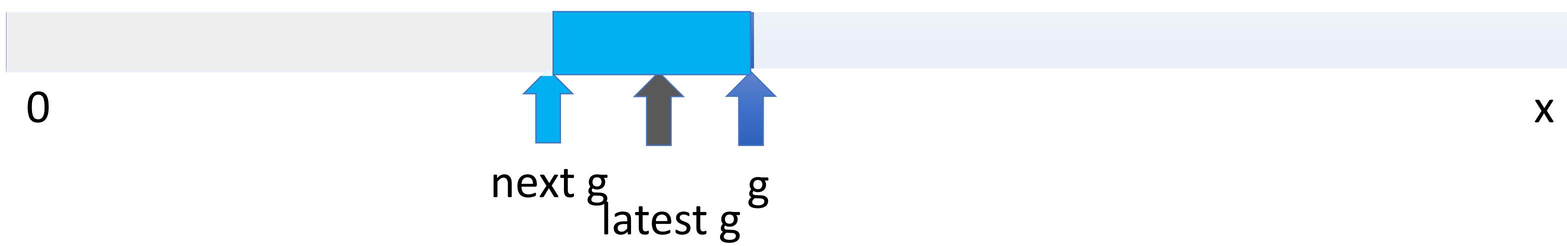
- And if, for example, this new g is such that g**2 < x, then know too small; so now search



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

BISECTION SEARCH for SQUARE ROOT

- And if, for example, this next 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**

BIG IDEA

Bisection search takes advantage of properties of the problem.

1) The search space has an order

2) We can tell whether the guess was too low or too high

YOU TRY IT!

- You are guessing a 4 digit pin code. The only feedback the phone tells you is whether your guess is correct or not. Can you use bisection search to quickly and correctly guess the code?

YOU TRY IT!

- You are playing an EXTREME guessing game to guess a number EXACTLY. A friend has a decimal number between 0 and 10 (to any precision) in mind. The feedback on your guess is whether it is correct, too high, or too low. Can you use bisection search to quickly and correctly guess the number?

SLOW SQUARE ROOT USING APPROXIMATION METHODS

```
x = 54321
epsilon = 0.01
num_guesses = 0
guess = 0.0
increment = 0.00001
while abs(guess**2 - x) >= epsilon and guess**2 <= x:
    guess += increment
    num_guesses += 1
print('num_guesses =', num_guesses)
if abs(guess**2 - x) >= epsilon:
    print('Failed on square root of', x)
else:
    print(guess, 'is close to square root of', x)
```

FAST SQUARE ROOT

```
x = 54321
epsilon = 0.01
num_guesses = 0

while abs(guess**2 - x) >= epsilon:

    num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to square root of', x)
```

Initialize some stuff

What is repeated each time?

FAST SQUARE ROOT

```
x = 54321
epsilon = 0.01
num_guesses = 0
low = 0
high = x
guess = (high + low)/2.0
while abs(guess**2 - x) >= epsilon:



    num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to square root of', x)
```

Initialize endpoints and guess

What is repeated each time?

FAST SQUARE ROOT

```
x = 54321
epsilon = 0.01
num_guesses = 0
low = 0
high = x
guess = (high + low)/2.0
while abs(guess**2 - x) >= epsilon:
    if guess**2 < x :

    else:



    num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to square root of', x)
```

Check if guess squared is too low or too high compared to x

FAST SQUARE ROOT

```
x = 54321
epsilon = 0.01
num_guesses = 0
low = 0
high = x
guess = (high + low)/2.0
while abs(guess**2 - x) >= epsilon:
    if guess**2 < x :
        low = guess
    else:


    num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to square root of', x)
```

If guess was too low, reset the low endpoint to the guess

FAST SQUARE ROOT

```
x = 54321
epsilon = 0.01
num_guesses = 0
low = 0
high = x
guess = (high + low)/2.0
while abs(guess**2 - x) >= epsilon:
    if guess**2 < x :
        low = guess
    else:
        high = guess

    num_guesses += 1
print('num_guesses =', num_guesses)
print(guess, 'is close to square root of', x)
```

If guess was too high, reset the high endpoint to the guess

FAST SQUARE ROOT
Python Tutor LINK

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

Make a new guess using the new endpoints

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

WHY?

- $N/2^k = 1$ Since at this point we have one guess left to check **this tells us n in terms of k**
- $N = 2^k$ Solve this for k
- $k = \log(N)$ Tells us **k in terms of N**

It takes us k steps to guess using bisection search

==

It takes us log(N) steps to guess using bisection search

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

You Try It: BISECTION SEARCH – SQUARE ROOT with 0 < x < 1

```
x = 0.5
epsilon = 0.01
```

Choose the appropriate endpoints

```
guess = (high + low)/2

while abs(guess**2 - x) >= epsilon:
    if guess**2 < x:
        low = guess
    else:
        high = guess
    guess = (high + low)/2.0

print(f'{str(guess)} is close to square root of {str(x)}')
```

BISECTION SEARCH – SQUARE ROOT for ALL x VALUES

```
x = 0.5
epsilon = 0.01

if x >= 1:
    low = 1.0
    high = x
else:
    low = x
    high = 1.0
guess = (high + low)/2

while abs(guess**2 - x) >= epsilon:
    if guess**2 < x:
        low = guess
    else:
        high = guess
    guess = (high + low)/2.0

print(f'{str(guess)} 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**

YOU TRY IT!

- Write code to do bisection search to find the cube root of positive cubes within some epsilon. Start with:

```
cube = 27
epsilon = 0.01
low = 0
high = cube
```

NEWTON-RAPHSON

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

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

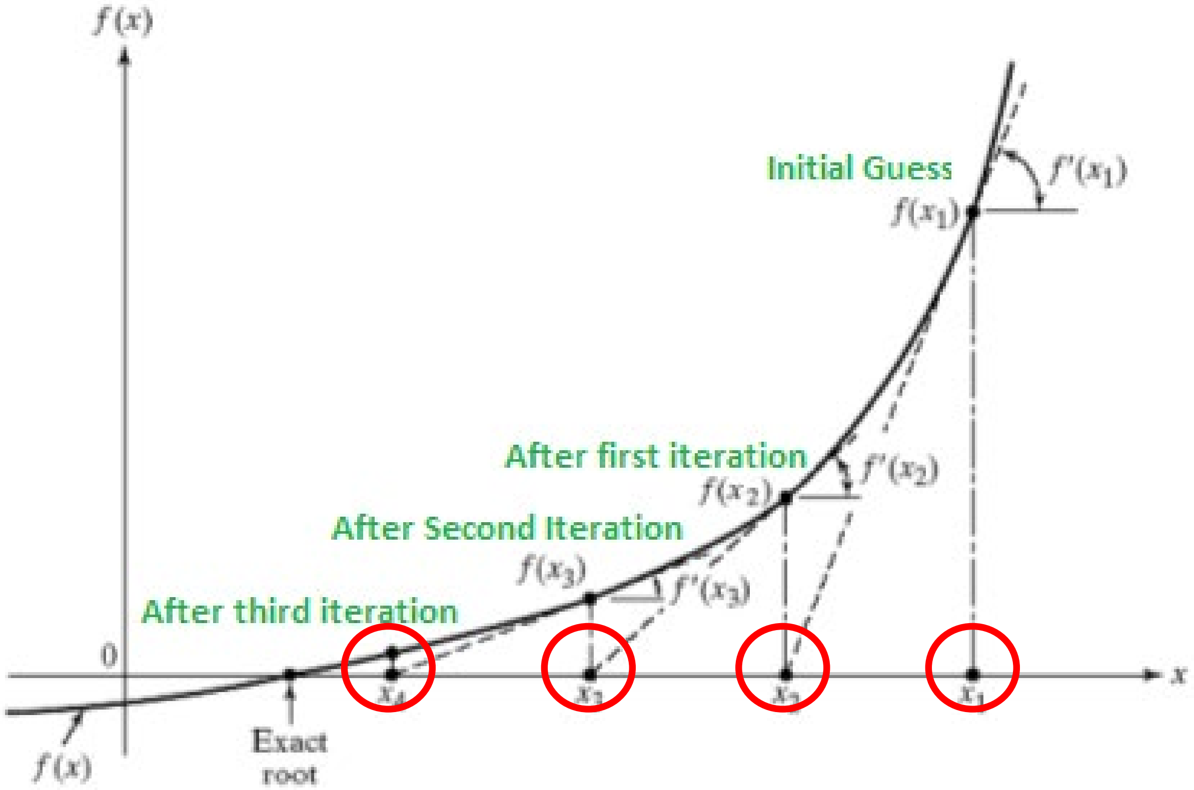
- Newton and Raphson showed that if g is an approximation to the root, then

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

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

- Try to use this idea for **finding the square root of x**
  - Want to find r such that $p(r) = 0$
  - For example, to find the square root of 24, find the root of $p(x) = x^2 - 24$

INTUITION - LINK



NEWTON-RAPHSON ROOT FINDER

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

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

- This eventually **finds an approximation to the square root of k!**

NEWTON-RAPHSON ROOT FINDER

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

```
epsilon = 0.01
k = 24.0
guess = k/2.0
num_guesses = 0

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

$f(x) = x^2 - 24$

f(guess)

f'(guess)

ITERATIVE ALGORITHMS

- Guess and check methods build on **reusing same code**
  - Use a looping construct
  - Generate guesses (important difference in algorithms)
  - Check and continue
- **Generating guesses**
  - Exhaustive enumeration
  - Approximation algorithm
  - Bisection search
  - Newton-Raphson (for root finding)

SUMMARY

- For many problems, **cannot find exact answer**
- Need to seek a **"good enough" answer** using approximations
- When testing floating point numbers
  - It's important to understand how the computer represents these in binary
  - Understand why we use "close enough" and not "=="
- Bisection search works is FAST but for problems with:
  - Two **endpoints**
  - An **ordering** to the values
  - **Feedback** on guesses (too low, too high, correct, etc.)
- Newton-Raphson is a smart way to find roots of a polynomial

DECOMPOSITION and ABSTRACTION

LEARNING to CREATE CODE

- So far have covered **basic language mechanisms** – primitives, complex expressions, branching, iteration
- In principle, **you know all you need to know** to accomplish anything that can be done by computation
- But in fact, we've taught you **nothing** about two of the most important concepts in programming…

DECOMPOSITION and ABSTRACTION

- **Decomposition**
- How to divide a program into **self-contained parts** that can be combined to solve the current problem

DECOMPOSITION and ABSTRACTION

- **Abstraction**
- How to ignore **unnecessary detail**

DECOMPOSITION and ABSTRACTION

- Decomposition:
  - Ideally **parts can be reused** by other programs
  - Self-contained means parts should **complete computation using only inputs provided** to them and "basic" operations

```
a = 3.14*2.2*2.2
```

```
pi = 3.14
r = 2.2
area = pi*r**2
```

- Abstraction:
  - Used to separate **what** something does, from **how** it actually does it
  - Creating parts and abstracting away details allows us to write complex code while **suppressing details**, so that we are not overwhelmed by that complexity

```
# calculate the area of a circle
```

BIG IDEA

Make code easy to

create
modify
maintain
understand

6.100L Introduction to Computer Science and Programming Using Python
Fall 2022