Numerical Computation

Thinking like a computer

In order to write an algorithm, you must think like a computer

- break down the problem into small steps
- identify repeating aspects that are easy to code with loops
- identify what information is essential to have ahead of time
- think about other information that you can DERIVE

Squaring a number with addition

- Algorithm for squaring with addition
  - add a number to itself, number times

PSEUDO CODE

Initialize variable to store the answer

Loop number times

add a number to itself

- Hidden step
  - initialize a variable to store the answer as it develops

- Essential information for a function
  - The number that will be squared!
  - This will be the parameter of the function

```python
# Squaring a number by addition

def square(number: int) -> int:
  """Square a number by addition."""

  count = 0
  answer = 0

  while count < number:
    answer += number
    count += 1

  return answer
```

PSEUDO CODE

Initialize variable to store the answer

Loop number times

add a number to itself

Figuring out the square root of a number (poorly)

Essential information: The number that should be square-rooted

- This will be a parameter to a function

Logical Steps:

- guess a random number as the solution
- confirm or deny by squaring it
- repeat until solution is found

Pseudo Code:

- loop while solution has not been found
  - create random guess
  - square random guess
  - if random guess squared IS the original number
    - return random guess!
  - Otherwise start process again

guess and check

Bad example!

```python
# Guess and Check

import random

def squareroot(number: int) -> int:
  """Guess integer roots and check."""

  while True:

    guess = random.randint(0, number)

    if guess**2 == number:
      return guess
```

- loop while solution has not been found
  - create random guess
  - square random guess
  - if random guess squared IS the original number
    - return random guess!
  - Otherwise start process again

guess and check

Organized guessing…?

Imagine: What is the square root of 225?

The solution should be somewhere between 1 and 225

[1,2,3,4,5,6,7,8,9,10,11,……,24,25,26,27,.....225]

Exhaustive enumeration

Figuring out the square root of a number

Essential information: The number that should be square-rooted

- This will be a parameter to a function

Logical Steps:

- check every number in a range, in order, exhaustively
- confirm or deny by squaring it
- repeat until solution is found

Pseudo Code:

- loop through the range
  - Consider item in the range
  - square the item
  - if item squared IS the original number
    - return it!
  - Otherwise move on to next item in the range

Exhaustive enumeration

```python
# Exhaustive Enumeration for perfect squares

def squareroot(number: int) -> int:
  """Exaustively check integer roots."""

  for possible_answer in range(number):

    if possible_answer**2 == number:
      return possible_answer

  return -1
```

- loop through the range
  - Consider item in the range
  - square the item
  - if item squared IS the original number
    - return it!
  - Otherwise move on to next item in the range

Exhaustive enumeration

What about non-perfect squares?

```python
# Exhaustive Enumeration for non-perfect squares

def squareroot_ee(number: int) -> float:
  """Exhaustively check all possible roots for numbers >= 1."""

  if number < 1:
    return -1

  epsilon = 0.01 # margin of error, could be a parameter
  step_size = epsilon**2
  possible_answer = 0

  while abs(number - possible_answer**2) > epsilon:
    possible_answer += step_size

  return possible_answer
```

Exhaustive enumeration

```
while abs(number - possible_answer**2) > epsilon:
```

- epsilon is a margin of error
- as you saw in the first lab, floating point numbers are not perfect
- It will never be possible to find the exact answer, so a margin of error is needed!

Figuring out the square root of a number

Essential information: The number that should be square-rooted

- This will be a parameter to a function

Logical Steps:

- check middle number in a range
- confirm or deny by squaring it
- cut the range in half intelligently
- repeat until solution is found

Pseudo Code:

- loop while solution has not been found
  - consider middle item
  - square item
  - if item squared IS the original number
    - return it!
  - Otherwise
    - if the item squared was too large
      - adjust the search range to be the lower half of the range
    - otherwise
      - adjust the search range to be the upper half of the range
  - repeat

Bisection Search

Bisection Search to find square root of 144

min = 0, max = 144, midpoint = 72

72*72 is too large!

min = 0, max = 72, midpoint = 36

36*36 is too large!

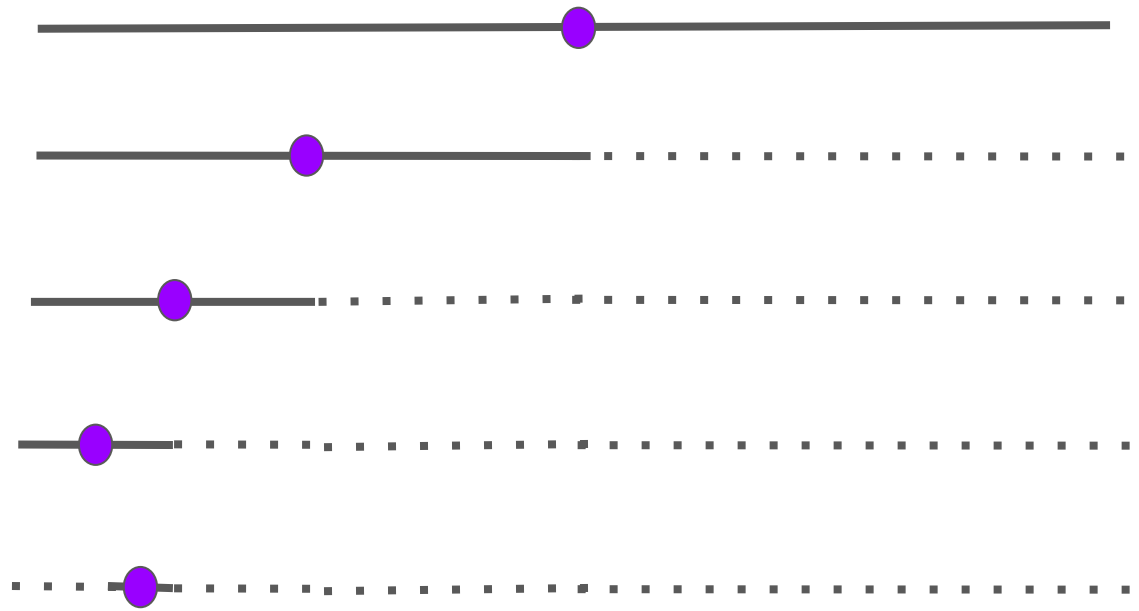
min = 0, max = 36, midpoint = 18

18*18 is too large!

min = 0, max = 18, midpoint = 9

9*9 is too small!

min = 9, max = 18, midpoint = 13.5 etc.



Search space is cut in half every time! Amazing, compared to exhaustive enumeration

```python
# Bisection Search for non-perfect squares

def squareroot_bs(number: int) -> float:
  """Perform bisection search to find root."""

  if number < 1:
    return -1

  epsilon = 0.01 # margin of error, could be a parameter
  lower_bound = 0
  upper_bound = number
  midpoint = (lower_bound+upper_bound)/2

  while abs(number - midpoint**2) > epsilon:

    if midpoint**2 > number:
      upper_bound = midpoint
    else:
      lower_bound = midpoint

    midpoint = (lower_bound+upper_bound)/2

  return midpoint
```

- loop while solution has not been found
  - consider middle item
  - square item
  - if item squared IS the original number
    - return it!
  - Otherwise
    - if the item squared was too large
      - adjust the search range to be the lower half of the range
    - otherwise
      - adjust the search range to be the upper half of the range
  - repeat

Bisection Search