Numerical Computation

Guttag Chapter 3

Prof. Graber

Goals

- Understand and run multiple algorithms for finding a square root
  - Random Guessing
  - Exhaustive Enumeration
  - Bisection Search

Square Root Definition

- The square root of a given number can be multiplied by itself to get the number.
- $\sqrt{49} = 7$
- $7 * 7 = 49$
- How can a computer find the $\sqrt{}$?

Random Guessing Algorithms

Guess and Check

Logical Steps:

- choose the number you want to $\sqrt{}$
- guess a random number as the solution
- confirm or deny by squaring it
- repeat until solution is found

. . .

Pseudo Code:

```
# choose a number to take the sqrt of
# 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

. . .

```python
import random

def squareroot_gc(number: int) -> int:
  """Guess integer roots and check."""
  while True:
    guess = random.randint(0, number)
    if guess**2 == number:
      return guess
```

. . .

Any concerns with this code?

. . .

- might never end
- only works with integer guesses for perfect squares!
- copy and try it below for a perfect square
- copy and try it below for a non-perfect square

Guess and Check (with a limit)

. . .

```python
import random

def squareroot_gcl(number: int) -> int:
  """Guess integer roots and check."""
  num_guesses_allowed = 100
```

```python
    num_guesses_sofar = 0
    while num_guesses_sofar < num_guesses_allowed:
      guess = random.randint(0, number)
      if guess**2 == number:
        return guess
      num_guesses_sofar += 1
    return -1
```

. . .

- copy and try it below!

Random Guessing Summary

- Simple algorithm
- Might never find the right answer

Exhaustive Enumeration Algorithms

Exhaustive Enumeration

. . .

Logical Steps:

- choose the number you want to $\sqrt{}$
- don't do random guessing and checking, do it in an organized way
- check every number in a range, in order, exhaustively
- confirm or deny by squaring it
- repeat until solution is found

. . .

Pseudo Code:

```
# choose a number to take the sqrt of
# loop through a range
    # Consider index in the range
    # square the index
    # if index squared IS the original number
        # return it!
    # Otherwise move on to next item in the range
```

Exhaustive Enumeration (for loop)

. . .

```python
# Exhaustive Enumeration for perfect squares

def squareroot_eep(number: int) -> int:
  """Exaustively check integer roots."""
  for possible_answer in range(number):
    if possible_answer**2 == number:
      return possible_answer
  return -1
```

. . .

Any concerns with this code?

. . .

- try it with 12345 * 12345
- try it with 144.1 * 144.1
- the code only works when there is an integer solution

Exhaustive Enumeration (while loop)

. . .

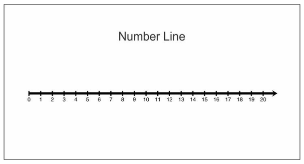
```python
# Exhaustive Enumeration for perfect squares

def squareroot_eep_while(number: int) -> int:
  """Exaustively check integer roots."""
  possible_answer = 0
  while possible_answer**2 <= number:
    if possible_answer**2 == number:
        return possible_answer
    possible_answer += 1
  return -1 # no answer found
```

. . .

Try it!

Exhaustive Enumeration

Exhaustive enumeration is like moving along a number line.



. . .

But number lines with integers are only useful for finding the square roots of perfect squares.

Exhaustive Enumeration (non-integer)

- Let's use a much **finer** number line to find non-integer solutions.
- We will also use the concept of an allowed **margin of error,** ϵ for solutions that are "good enough".
- For example, what is the sqrt of 26?
- 5.1 * 5.1 = 26.01, so depending on ϵ, we might accept or reject the solution of 5.1

Exhaustive Enumeration (non-integer)

. . .

Pseudo Code:

```
# choose a number to take the sqrt of
# define an epsilon (allowed margin of error)
# define a tiny step size
# initialize the possible answer
# while loop so long as possible_answer**2 is too small, allowing for the margin of error
    # increase possible answer by the tiny step size
```

Exhaustive Enumeration (non-integer)

. . .

```python
1  # Exhaustive Enumeration for non-perfect squares
2
3  def squareroot_ee(number: int) -> float:
4    """Exhaustively check all possible non-integer roots."""
5    epsilon = 0.01 # margin of error
6    step_size = epsilon**2
7    possible_answer = 0
8    while possible_answer**2 < number + epsilon:
9      if possible_answer**2 > number - epsilon:
10         return possible_answer # good!
11     possible_answer += step_size
12   return possible_answer # not so good!
13
14 print(squareroot_ee(26))
```

```
5.098100000001457
```

. . .

Why is line 12 marked as "not so good" after the while loop?

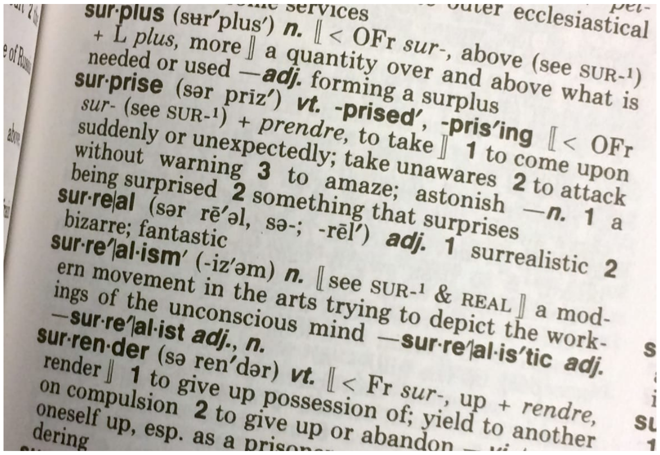
Exhaustive Enumeration Summary

- Possible solutions are checked in order
- Step sizes can be integer or non-integer
- Many steps could be required!

Bisection Search Algorithms

Bisection Search

. . .



services ... outer ecclesiastical

sur·plus (sur'plus') ***n.*** ⟦< OFr *sur-*, above (see SUR-1) + L *plus*, more⟧ a quantity over and above what is needed or used —***adj.*** forming a surplus

sur·prise (sər prīz') ***vt.*** **-prised', -pris'ing** ⟦< OFr *sur-* (see SUR-1) + *prendre*, to take⟧ **1** to come upon suddenly or unexpectedly; take unawares **2** to attack without warning **3** to amaze; astonish —***n.*** **1** a being surprised **2** something that surprises

sur·re|al (sər rē'əl, sə-; -rēl') ***adj.*** **1** surrealistic **2** bizarre; fantastic

sur·re'|al·ism' (-iz'əm) ***n.*** ⟦see SUR-1 & REAL⟧ a modern movement in the arts trying to depict the workings of the unconscious mind —**sur·re'|al·is'tic** ***adj.*** —**sur·re'|al·ist** ***adj., n.***

sur·ren·der (sə ren'dər) ***vt.*** ⟦< Fr *sur-*, up + *rendre*, render⟧ **1** to give up possession of; yield to another on compulsion **2** to give up or abandon ... oneself up, esp. as a prisoner ... dering

. . .

How do you use this?

Bisection Search

- Bisection search is like searching through a Merriam Webster paper dictionary for a specific word.
- You zero in on the word by going **forward or backward** multiple times
- For square roots, you zero in on the solution by going **forward or backward** multiple times
- There is no number line in bisection search for square roots
- There is no step size because the search does not proceed linearly

Bisection Search

. . .

Logical Steps:

- choose the number you want to $\sqrt{}$
- define a search range with an upper and lower bound
- check middle number in a range
- confirm or deny by squaring it
- eliminate half of the search range intelligently
- repeat until a “good enough” solution is found

Bisection Search

. . .

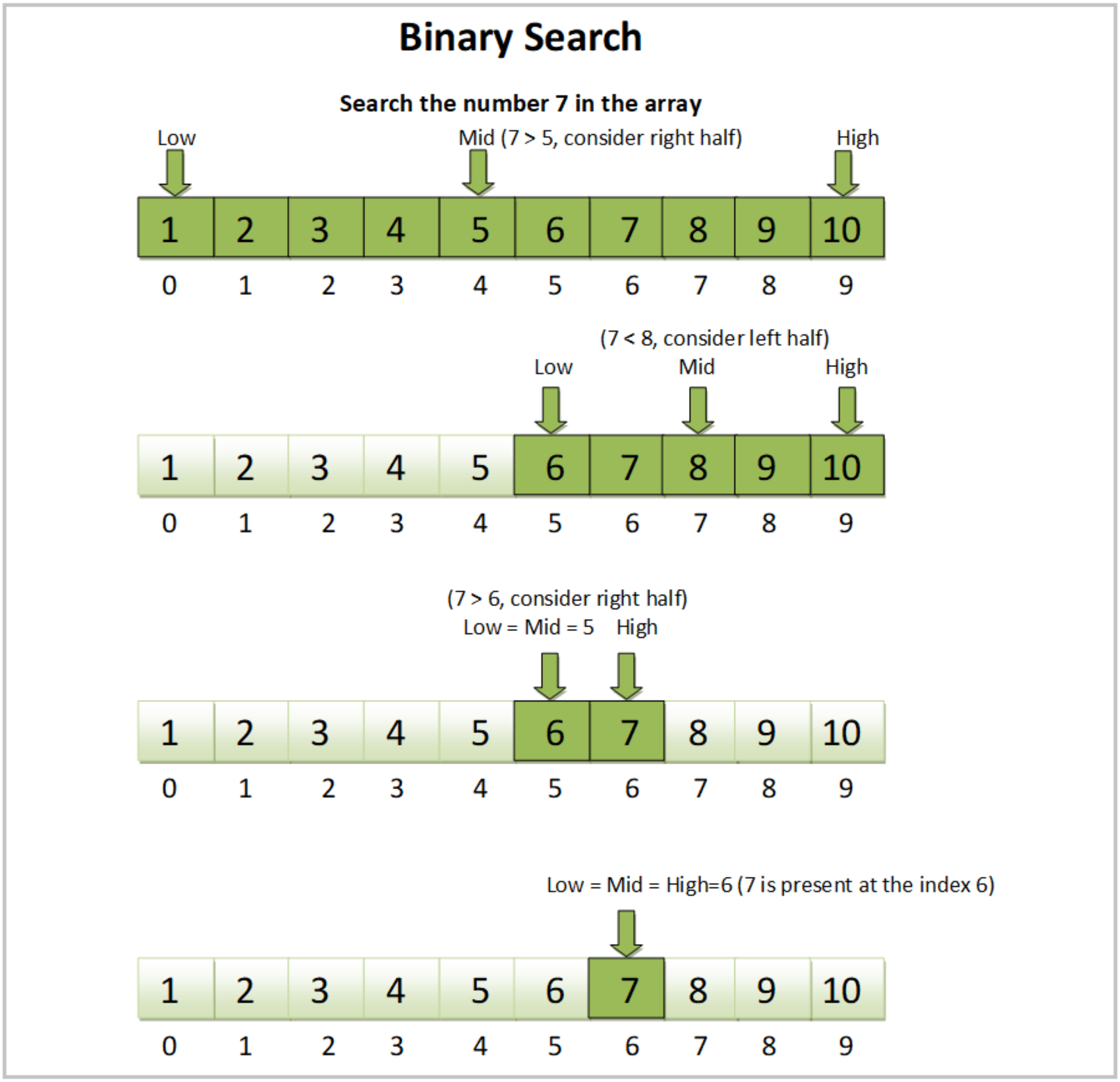
Pseudo Code:

```
# choose a number to take the sqrt of
# define an epsilon (allowed margin of error)
# initialize the search space starting and ending value
# initialize the possible answer to the middle of the search space
# while loop so long as possible_answer**2 is not _within_ the margin of error (±)
    # if the possible_answer ** 2 was too large
        # adjust the search space to be the lower half of the space
    # Or, if the possible_answer ** 2 was too small
        # adjust the search space to be the upper half of the space
    # compute the middle of the new search space and assign that to possible answer
# assume possible_answer is within the margin of error!
```

. . .

Note how the start, middle, and end of the search space are used repeatedly

Bisection Search



Bisection Search (number > 1)

. . .

```
# Bisection Search for non-perfect squares

def squareroot_bs(number: int) -> float:
```

```python
    """Perform bisection search to find root."""
    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

print(squareroot_bs(144.3))
```

```
12.012155914306641
```

. . .

- this algorithm does not work for numbers less than 1, try it!

Bisection Search (number < 1)

. . .

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

def squareroot_bs(number: int) -> float:
    """Perform bisection search to find root."""
    epsilon = 0.01 # margin of error, could be a parameter
    lower_bound = 0
    if number < 1:
      upper_bound = 1
    else:
      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

print(squareroot_bs(0.5))
```

```
0.703125
```

. . .

What changed?

Bisection Search Summary

- Possible solutions are checked in the middle of a search space
- The search space quickly gets smaller by eliminating half of the space on every iteration
- There is no guiding step size

Closing Thoughts

Understanding the Computer

- simple algorithms like random guessing are usually less efficient
- computer can never get the exact answer for non perfect squares
- numerical strategies often require approximation (like using a margin of error)

Challenge

- Try to add code that can count how many times the loops repeat for the various methods of computing a square root.
- Which algorithm runs with the fewest iterations of the loop?