# Week 9: Agenda

- Dictionaries and Sets, Efficiency
  - Code Tracing
  - Free Response
- Course admin
  - What’s next?
  - Homework 8
- Recursion
  - Let’s play a game
  - Example

# Code Tracing: Spring 2021 (Exam 2)

```
def f(u):
    if 8 in u:
        print(f'8: {u[8]}')
        del u[8]
    return u

def ct(L):
    s = set(L)
    d = dict()
    for v in L:
        d[v] = d.get(v,v) + min(s)
        s.add(d[v])
    u = f(d)
    print(f's = {s}')
    print(f'd = {d}')
    print(f'u = {u}')

ct([8,4,8,4,2])
```

# Dictionaries: `mostVisits`

Write the function `mostVisits(logbook)` that is given a dictionary mapping days of the week to the list of students who visited CMU-Q on that day, and returns a set that contains the student (or students, if there is a tie) who visited on the most number of days that week. There is one caveat: The log system might register a visit multiple times on the same day, therefore one name might appear multiple times in a list, but it should be counted only once per day.

For example, given the dictionary:

```
{ "Sunday" : [ "Layla", "Peter", "Otto", "Amir" ],
"Monday" : [ "Yusuf", "Layla", "Bernard", "John" ],
"Tuesday" : [ "Yusuf", "Peter", "Otto", "Layla", "Salma", "Otto" ],
"Wednesday" : [ "Otto", "Layla", "Yusuf", "Otto" ] }
```

The function should return the set `{"Layla"}`, since Layla visited the CMU-Q building four days (Otto entered the building more times, but only visited on three different days).

If Layla had not visited the building on Monday, then it would return `{"Layla", "Otto", "Yusuf"}`, since each student would have visited exactly three days.

# Dictionaries: `mostVisits`

- Many,  many, possible solutions
  - Usually, building a dictionary like this would help (a lot)

| Student | Visit Count (number of days) |
|---|---|
| Layla | 4 |
| Peter | 2 |
| Otto | 3 |
| Yusuf | 3 |
| … | … |

  - Then find the maximum value and return the corresponding key (see how we did it in `mostFrequentWord`)

# Fluke Numbers

(15 points) **Free Response: Fluke Numbers** A *fluke number* (coined term) is an integer that has a frequency in the list equal to its value

Write the function `findFlukeNumbers(L))` that is given a list `L` of objects (not necessarily integers). The function should return a set containing all the fluke numbers in the list. Your solution should run in $O(N)$ time.

For example,

```
assert(findFlukeNumbers([1,'a','a',[4], 3, False, 3, 3]) == {1, 3})
assert(findFlukeNumbers([1, 2, 2, 3, 3, 3, 4]) == {1, 2, 3})
assert(findFlukeNumbers([0, False, 'hello']) == set())
```

```
import string
def bigOh(s):  # s is a string,  N = len(s)
    result = ""                                         _____
    for c in s:                                         _____
        for c in string.ascii_lowercase:                _____
            if s.count(c) == result.count(c):           _____
                result += c                             _____
    return c                                            _____
```

---

```
def bigOh(L):  # L is a list, N = len(L)
    d = dict()                                          _____
    for i in L:                                         _____
        d[i] = i                                        _____
    return len(d)                                       _____
```

---

```
def bigOh(L):  # L is a list, N = len(L)
    n = len(L)                                          _____
    for i in range(n**2):                               _____
        L.append(L.count(i))                            _____
```

# Efficiency

What's the maximum length of result?

```
import string
def bigOh(s):  # s is a string,  N = len(s)
    result = ""
    for c in s:
        for c in string.ascii_lowercase:
            if s.count(c) == result.count(c):
                result += c
    return c
```

O(1)
O(n)
O(1)
O(n)
O(1)
O(1)

$\mathcal{O}(n^2)$

```
def bigOh(L):  # L is a list, N = len(L)
    d = dict()
    for i in L:
        d[i] = i
    return len(d)
```

O(1)
O(n)
O(1)
O(1)

$\mathcal{O}(n)$

```
def bigOh(L):  # L is a list, N = len(L)
    n = len(L)
    for i in range(n**2):
        L.append(L.count(i))
```

O(1)
O(n^2)
O(n)

$\mathcal{O}(n^3)$

# What’s next?

| | | | |
|---|---|---|---|
| HW#8 Due | W9 | Recursion | Quiz #8 |
| HW#9 Due | W10 | OOP (Term Project Intro) | Quiz #9 |
| Project Proposals Due | W11 | Exam 2 (**Sunday** Nov 5th), OOP | Term Project Season Starts! |
| | W12 | Searching and Sorting / Hashing | |
| | W13 | … | |
| | W14 | … | |

Start discussing your project ideas with your mentor!

# Game time! Pretend to be a Python function

- You will be a Python function
- When you are called, you come to the front, next in line, and receive the parameters in a green paper
- When you return, you provide the returning value (there's always a return value), in a red paper
- When you return, come back to the *workers' corner*

# Let’s try

- Warmup:

```python
def myMysteryFunction(s):
    value = 0
    if len(s) > 0 and s[0] in "aeiou":
        value = 1
    return value
```

- One volunteer
  - Contributes to your participation grade:

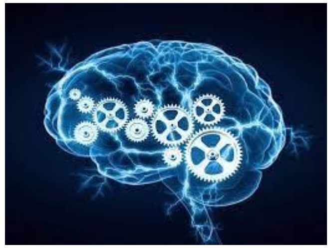
# Another (more complex) function

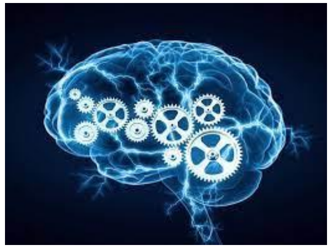
```
def myMysteryFunction(s):
    value = 0
    for c in s:
        if c in "aeiou":
            value += 1
    return value
```

# Let’s try

- This is the function:

```python
def myMysteryFunction(s):
    value = 0
    for c in s:
        if c in "aeiou":
            value += 1
    return value
```

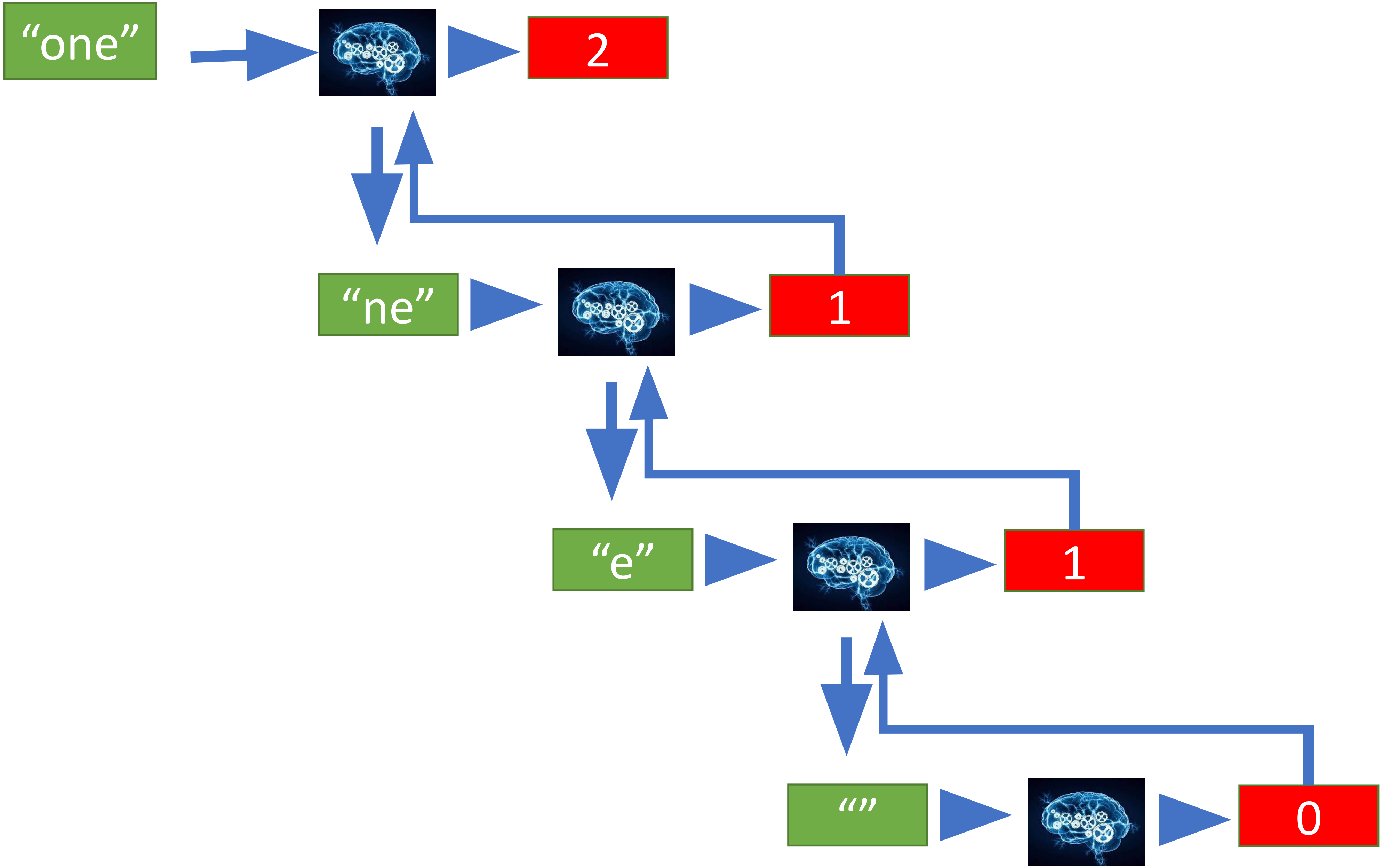
“one” →  → 2

“onetwelve” →  → 4

# Let’s now try this:

```python
def myRecursiveFunction(s):
    if len(s) == 0:
        return 0
    else:
        value = myRecursiveFunction(s[1:])
        if s[0] in "aeiou":
            value += 1
        return value
```

What do you notice?

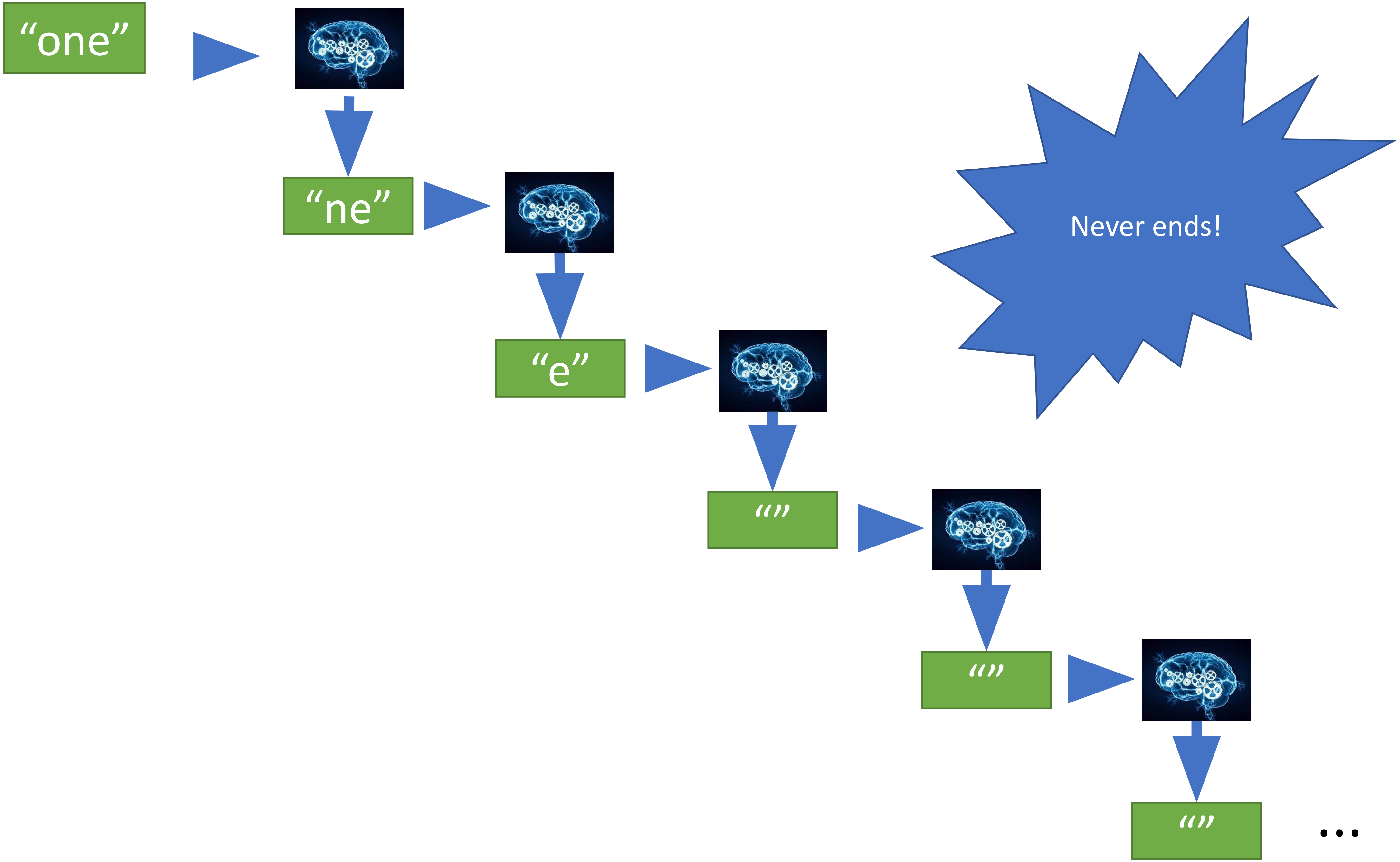


“one”
2
“ne”
1
“e”
1
“”
0

# A bad try

```python
def myRecursiveFunction(s):
    value = myRecursiveFunction(s[1:])
    if s[0] in "aeiou":
        value += 1
    return value
```

What do you notice?



“one”
“ne”
“e”
“”
“”
“”
…
Never ends!

# Recursion: generic form

```
def recursiveFunction():
    if (this is the base case):
        # no recursion allowed here!
        do something non-recursive
    else:
        # this is the recursive case!
        do something recursive
```

# onlyVowels(s)

- Write a recursive function that, given a string s, returns the vowels contained in s in the same order (as a string):

- onlyVowels("hello") == "eo"
- onlyVowels("bcdfg") == ""
- onlyVowels("aaaaa") == "aaaaa"

```
def recursiveFunction():
    if (this is the base case):
        # no recursion allowed here!
        do something non-recursive
    else:
        # this is the recursive case!
        do something recursive
```

# Forward Recursion vs. Tail Recursion

**In forward recursion:**

- Call the function recursively on all the sub-problems
- then build the final result from the partial results.

```
def onlyVowels(s):
    if len(s) == 0:
        return ""
    else:
        othervowels = onlyVowels(s[1:])
        if s[0] in "aeiou":
            return s[0] + othervowels
        else:
            return othervowels
```

**In tail recursion:**

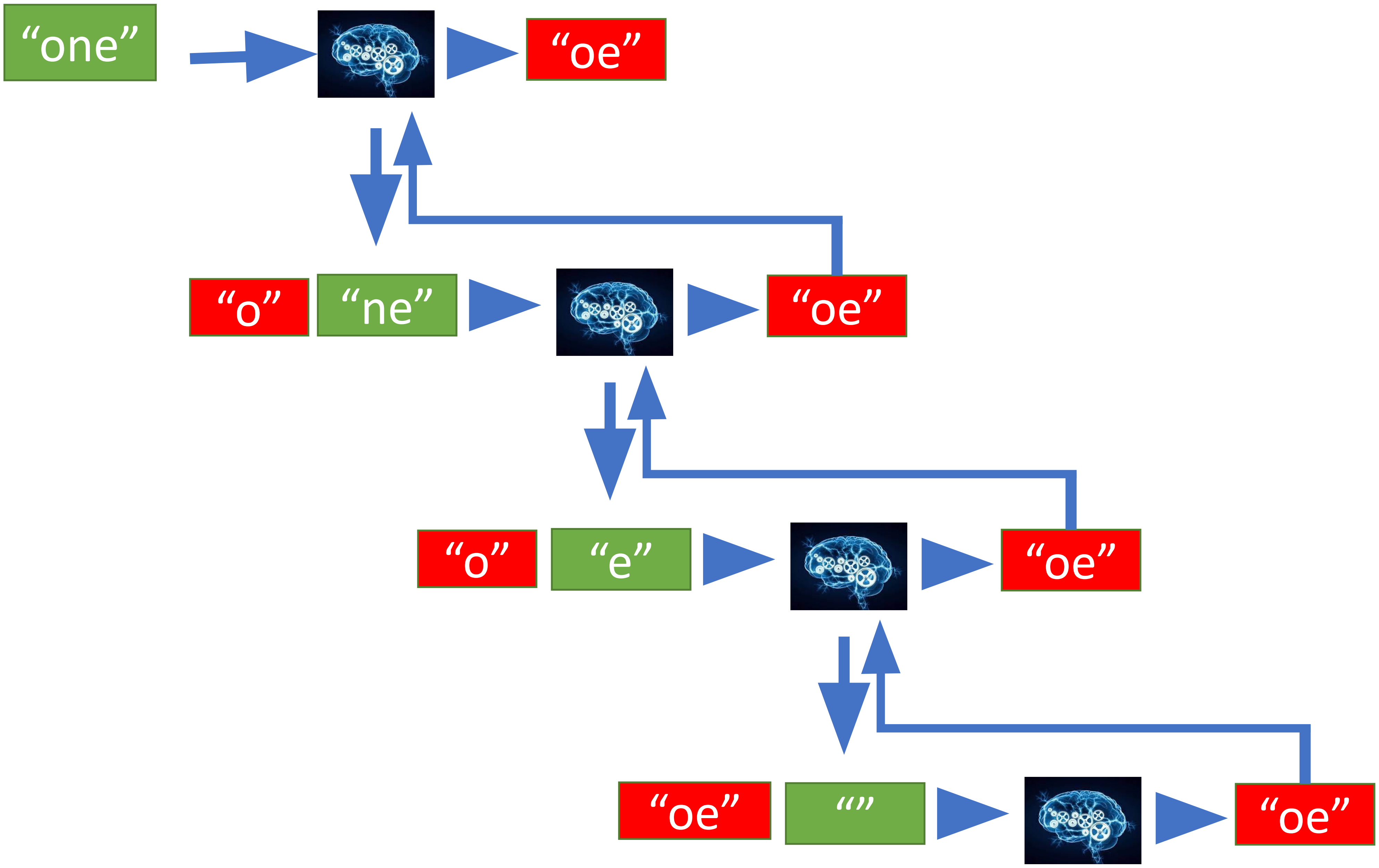
- A recursive function is tail-recursive if all recursive calls are the last thing that the function does

```
def onlyVowelsHelper(s, currentVowels):
    if len(s) == 0:
        return currentVowels
    if s[0] in 'aeiou':
        currentVowels += s[0]
    return onlyVowelsHelper(s[1:], currentVowels)

def onlyVowels(s):
    return onlyVowelsHelper(s, "")
```

# Using an *accumulator*

- Parameter that contains data processed by previous recursive calls
- If can be used to *store* partial solutions
- Normally used to implement tail-recursion
- Useful when the solution is a mutable type (e.g., lists, dictionaries)



“one”
“oe”
“o”
“ne”
“oe”
“o”
“e”
“oe”
“oe”
“”
“oe”

# `DigitSum(n)`

- Return the sum of all digits in n
- Do not use `for` or `while` loops

# recSumConsecutivePairs(L)

Write the function **recSumConsecutivePairs(L)** that returns a new list with the sums consecutive pairs of elements in L, in the corresponding order. If there are no consecutive pairs, it should return an empty list.

For instance,

```
recSumConsecutivePairs([3,2,5,1]) == [5,7,6]           #  3+2, 2+5,  5+1
recSumConsecutivePairs([-1,4,10,2,0]) == [3,14,12,2]  # -1+4, 4+10, 10+2, 2+0
recSumConsecutivePairs([1]) == []  # no consecutive pairs
recSumConsecutivePairs([]) == []   # no consecutive pairs
```

**Your solution must use recursion. If you use any loops, comprehensions, or iterative functions, you will receive no points on this problem.**

# `interleave(A, B)`

Recursive function that interleaves two lists A and B

Easy case: assume `len(list1) == len(list2)`

Example:

```
interleave([1,2,3], [4,5,6]) == [1,4,2,5,3,6]
interleave([1], [2]) == [1,2]
interleave(['a','c'], ['b', 'd']) == ['a', 'b', 'c', 'd']
```

# `interleave(A, B)`

Recursive function that interleaves two lists A and B

Easy case: **do not assume** `len(list1) == len(list2)`

Example:

```
interleave([1,2], [4,5,6]) == [1,4,2,5,6]
interleave([1], []) == [1]
interleave(['a','b','c'], ['d']) == ['a', 'd','b','c']
```

# Debugging

- Add “default” argument d = depth of the recursion
- Use the depth to add an offset to the print statements

# Solving problems with recursion

- Consider the generic form
- How can you split the problem?
  - How would the next recursive call look like?
- What’s the return type?
  - Usually, it is the same for the base case and the recursive case
- Base case
- Recursive case (assume that the recursive call works)

# Unlocking the power of recursion

Tree Recursion: When you make a recursive call more than once in your recursive case

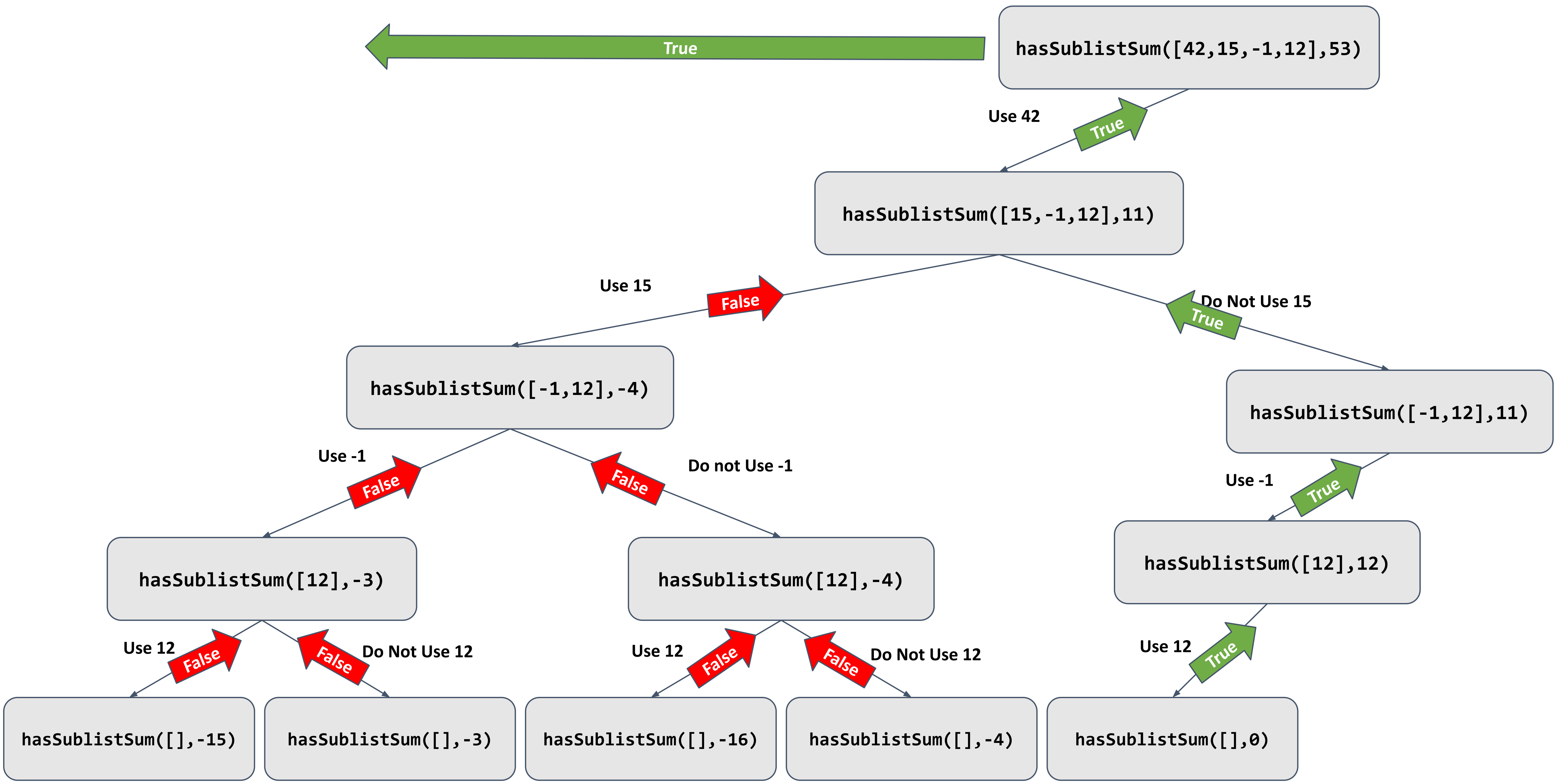
Why?

- Some problems are more easily solved by tree recursion.
- Brute forcing solutions (Backtracking)

Some Examples: `hasSublistSum`

# `hasSublistSum(L, s)`

Write the function `hasSublistSum(L, s)` that takes a list of integers `L` and an integer `s`, and returns `True` if there exist elements in `L` that sum to `s`. Otherwise, the function returns `False`.


True
hasSublistSum([42,15,-1,12],53)
Use 42
True
hasSublistSum([15,-1,12],11)
Use 15
False
Do Not Use 15
True
hasSublistSum([-1,12],-4)
hasSublistSum([-1,12],11)
Use -1
False
Do not Use -1
False
Use -1
True
hasSublistSum([12],-3)
hasSublistSum([12],-4)
hasSublistSum([12],12)
Use 12
False
Do Not Use 12
False
Use 12
False
Do Not Use 12
False
Use 12
True
hasSublistSum([],-15)
hasSublistSum([],-3)
hasSublistSum([],-16)
hasSublistSum([],-4)
hasSublistSum([],0)

# Example A: **`getHiLo(L)`**

- Write the function `getHiLo(L)` that receives a list of integers `L` and returns a tuple `(a,b)` where a is the lowest number and b is the highest. You can assume `len(L) > 0`
- Examples:
  - `getHiLo([1,2,3,4,5]) == (1,5)`
  - `getHiLo([42,4,5,-6]) == (-6,42)`
  - `getHiLo([42]) == (42,42)`

# Example B: **`indexMap(L)`**

- Write the function `indexMap(L)` that takes a 1D list L and returns a dictionary that maps each value in L to a set of the indices in L where that value occurs. For example:
- `indexMap([5, 6, 5]) == { 5:{0,2}, 6:{1} }`
- `indexMap([9, 6, 3, 6, 9]) == { 3:{2}, 6:{1,3}, 9:{0,4} }`