Fundamentals of Programming & Computer Science CS 15-112

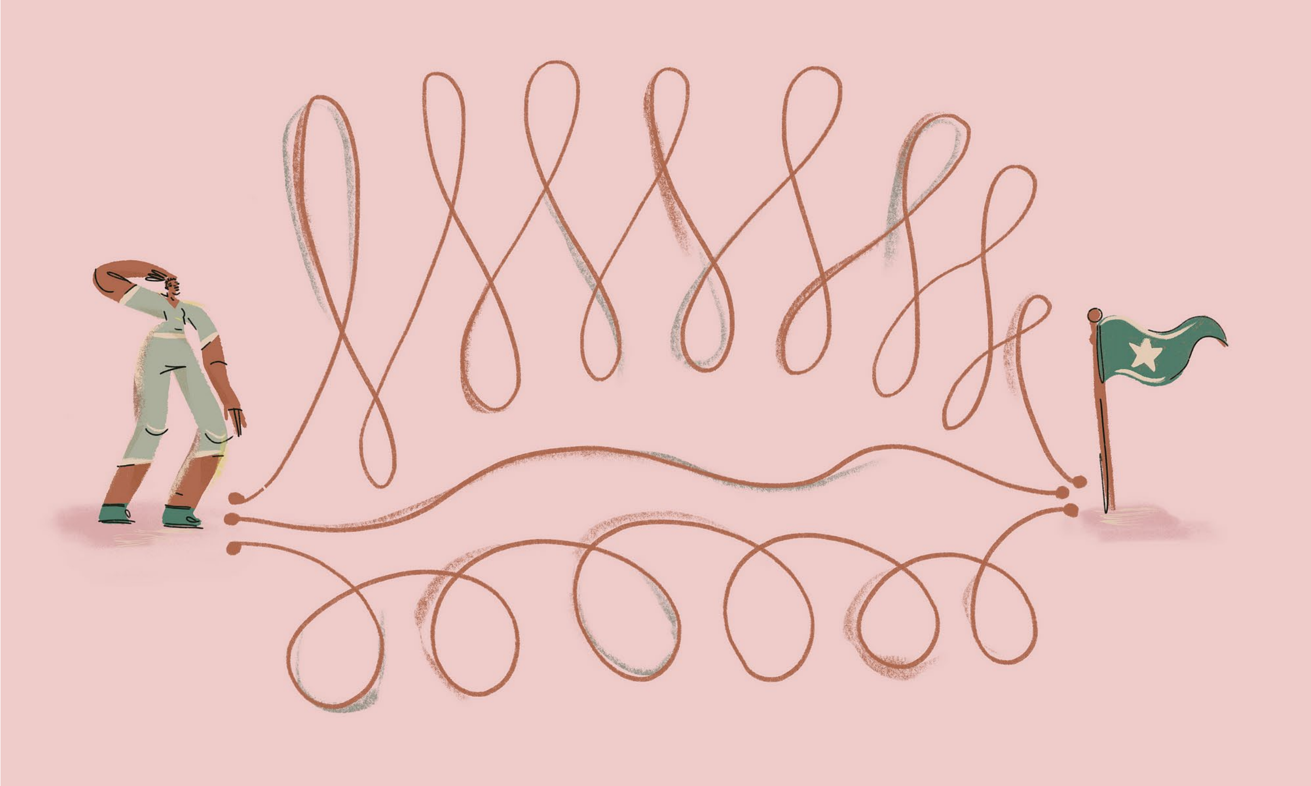
Efficiency

March 5

Hend Gedawy

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

There are Many Ways to Solve Any Given Problem

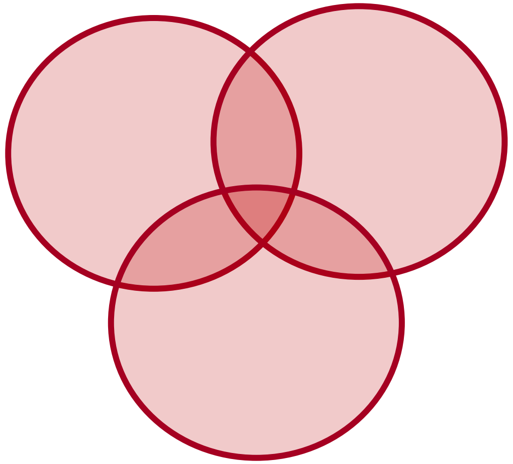


Some are Better or more Efficient than Others !

What is Efficiency?

Efficiency is a measure of how much of a resource an algorithm uses!

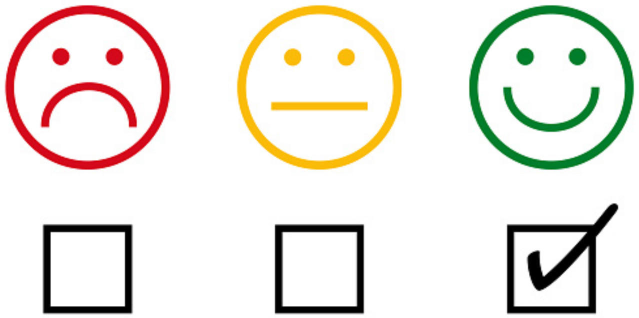
Time



Space

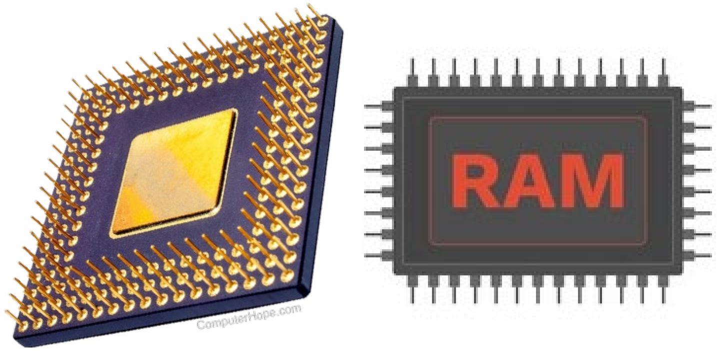
Why Care About Time Efficiency?

User Experience



Business/Commercial Costs

Compute Resources



Battery Lifetime

How to Assess Time Efficiency?

Measure Elapsed Time

Why isn't the elapsed time for an algorithm constant?

- **Hardware Differences:**
  - CPU speed, number of cores, memory (RAM), disk speed, etc.

- **Operating System:**
  - Different OSs may have different scheduling algorithms, memory management strategies, and other system-level optimizations that can impact runtime.

- **Resource Utilization:**
  - If the system is under heavy load or if other resource-intensive tasks are running concurrently, the algorithm may experience slower execution times.

How to Assess Time Efficiency?

We want to measure the efficiency of an algorithm independent of the speed of the computer it is run on.

A better alternative is Counting Steps that the code takes

… Given input of size (N) …

Very good proxy to time performance

(but always constant)

Counting Steps

Two rules:

- **A step takes constant amount of time;**

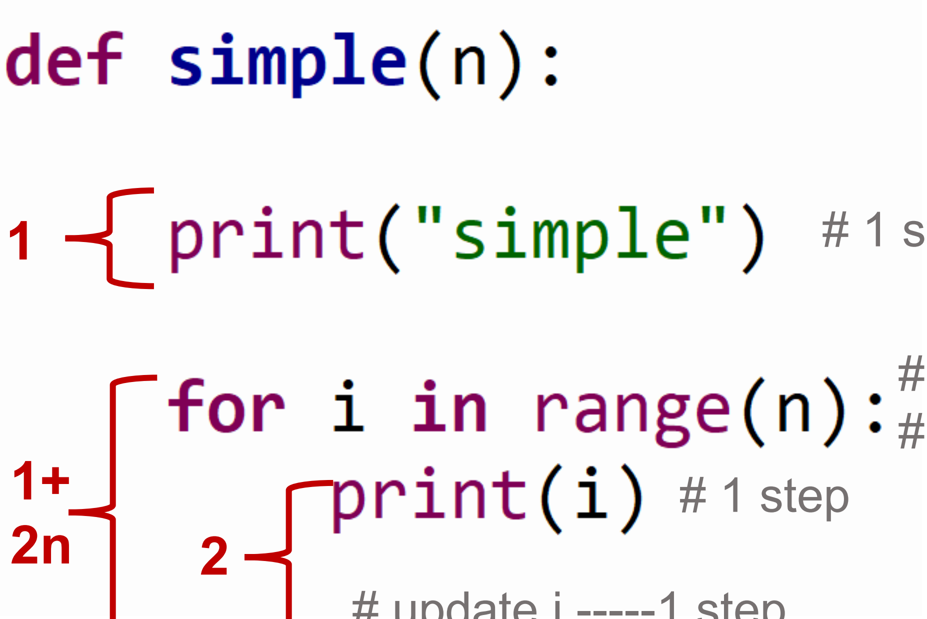
i.e. time doesn't increase as the input size (called n) increases

- **Generally, A line of code is a single step if the whole line runs in constant time**

Input Size (n) is the integer n

Counting Steps

Number of iterations with range = (end - start)/step



```
def simple(n):

    print("simple")        # 1 step

    for i in range(n):     # 1 step for range
                           # Loop runs for n iterations
        print(i)           # 1 step

        # update i -----1 step
```

1

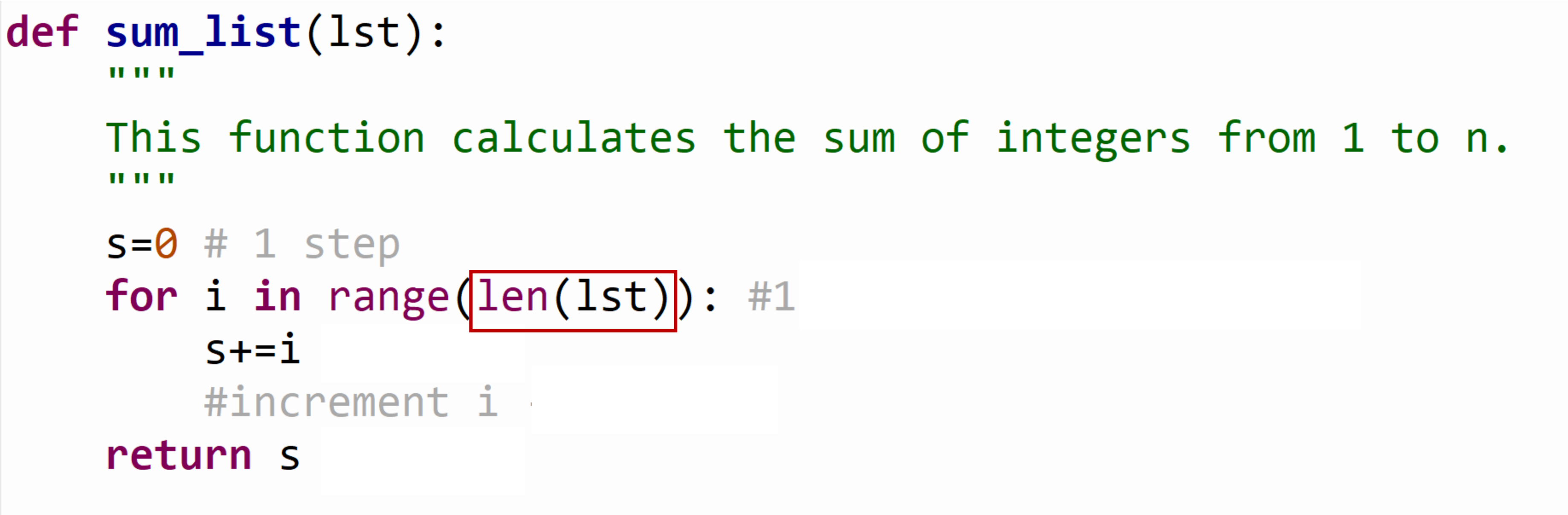
1+ 2n

2

Total Number of Steps: 1+1+ 2n = 2n + 2

Input Size (n) is the size of the list

Counting Steps



```
def sum_list(lst):
    """
    This function calculates the sum of integers from 1 to n.
    """
    s=0 # 1 step
    for i in range(len(lst)): #1
        s+=i
        #increment i
    return s
```

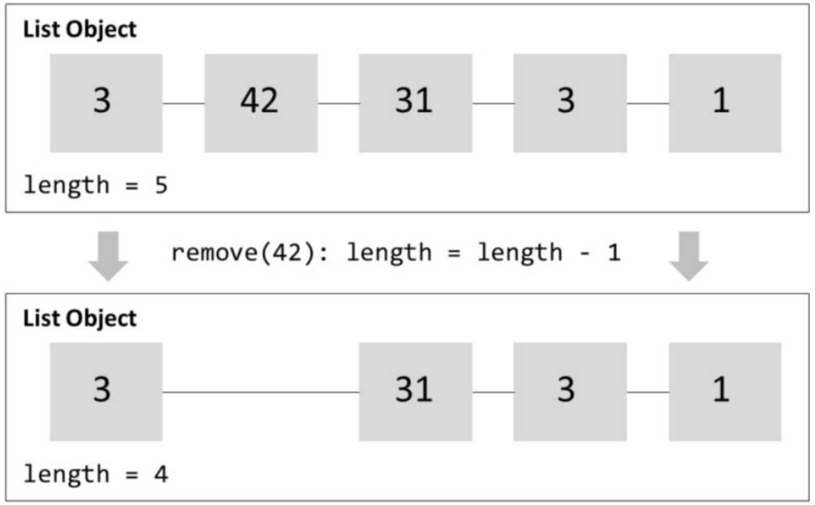
Counting Steps

Why does len() take a constant amount of time (1 step)???

How come it is not affected by List size??

Len() takes constant runtime no matter how many elements are in the list.

Because in Python the list object maintains an integer counter that increases and decreases as you add and remove list elements



List Object

3 42 31 3 1

length = 5

remove(42): length = length - 1

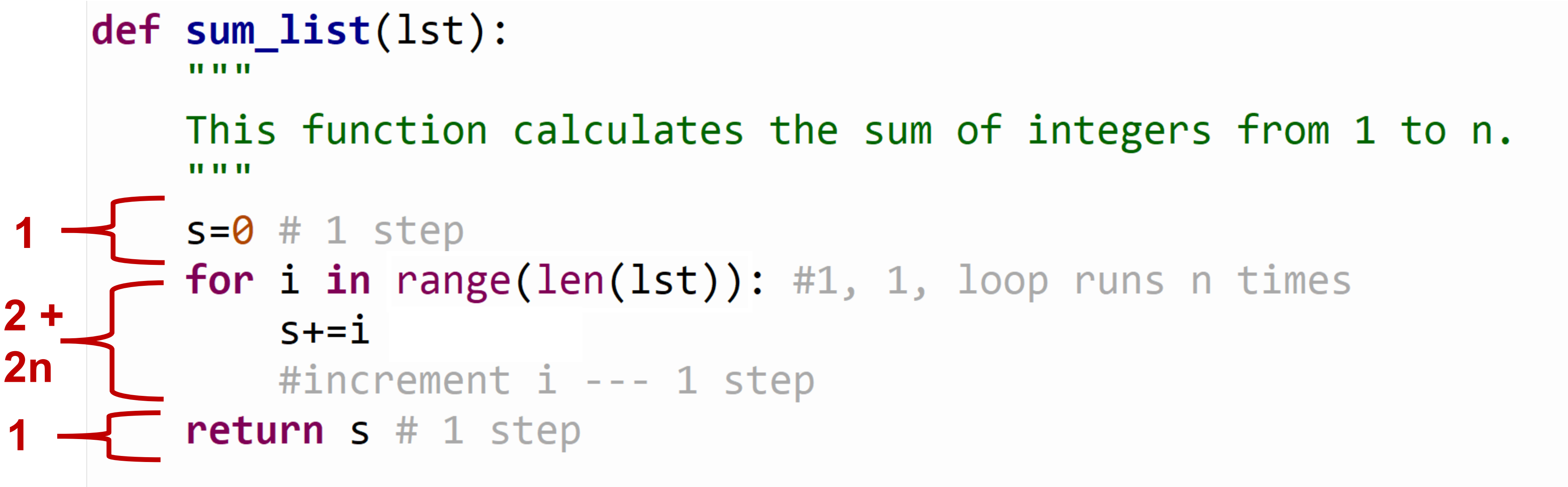
List Object

3 31 3 1

length = 4

Input Size (n) is the size of the list

Counting Steps

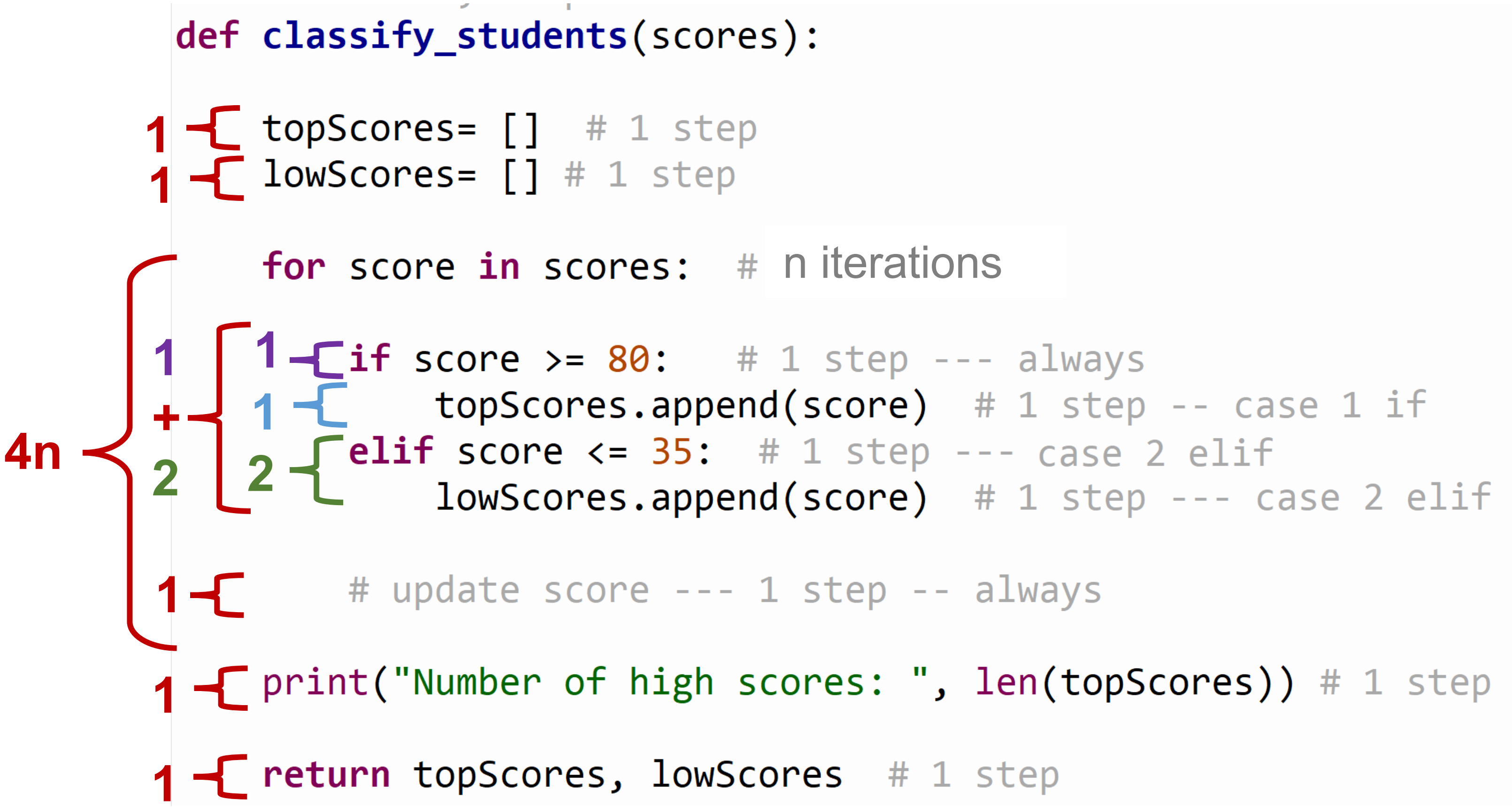


```
def sum_list(lst):
    """
    This function calculates the sum of integers from 1 to n.
    """
    s=0 # 1 step
    for i in range(len(lst)): #1, 1, loop runs n times
        s+=i
        #increment i --- 1 step
    return s # 1 step
```

Total Number of Steps: 1+ 2+ 2n + 1= 2n + 4

Input Size (n) is the size of the list

Counting Steps



```
def classify_students(scores):

    topScores= []  # 1 step
    lowScores= [] # 1 step

    for score in scores:   # n iterations

        if score >= 80:    # 1 step --- always
            topScores.append(score)  # 1 step -- case 1 if
        elif score <= 35:  # 1 step --- case 2 elif
            lowScores.append(score)  # 1 step --- case 2 elif

        # update score --- 1 step -- always

    print("Number of high scores: ", len(topScores)) # 1 step

    return topScores, lowScores  # 1 step
```

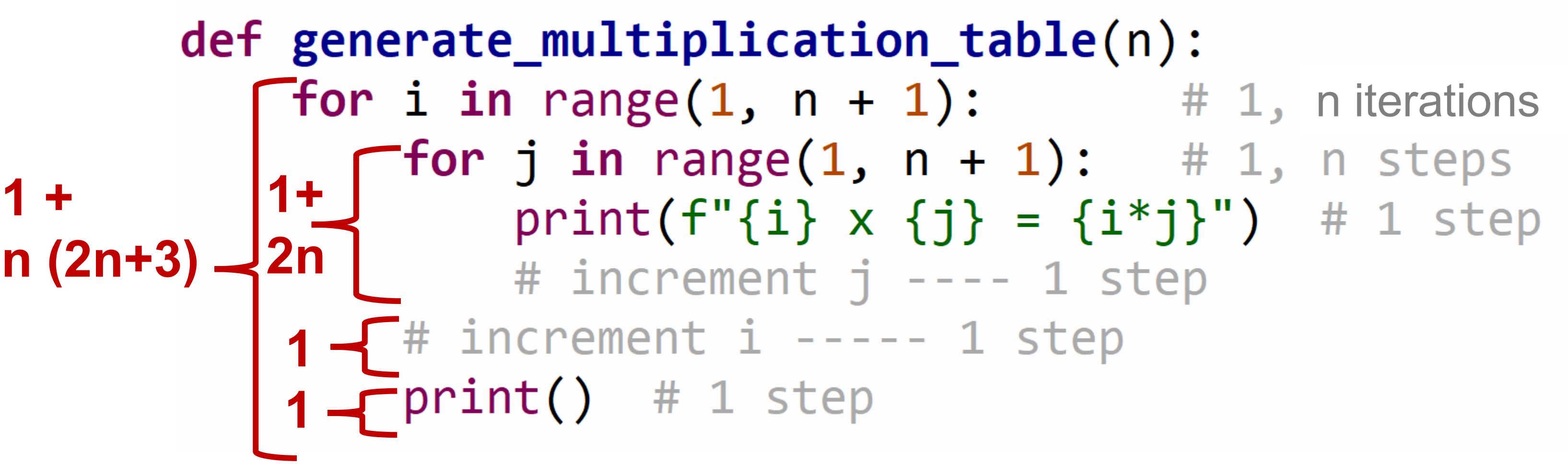
Case 2 is the Worst Case

In this course, we apply **worst case** analysis

Total Number of Steps: 4n+ 4

Input Size (n) is the passed integer value

Counting Steps



```
def generate_multiplication_table(n):
    for i in range(1, n + 1):        # 1, n iterations
        for j in range(1, n + 1):    # 1, n steps
            print(f"{i} x {j} = {i*j}")  # 1 step
            # increment j ---- 1 step
        # increment i ----- 1 step
        print()  # 1 step
```

1 +
n (2n+3)

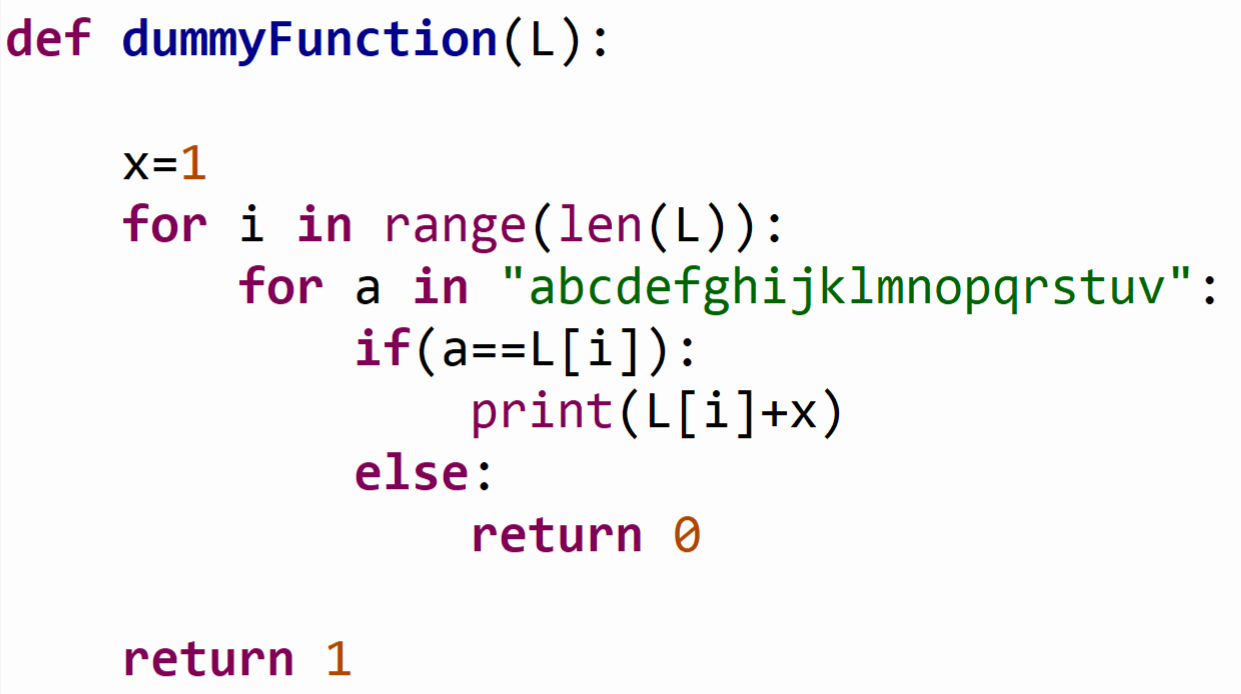
1+
2n

1

1

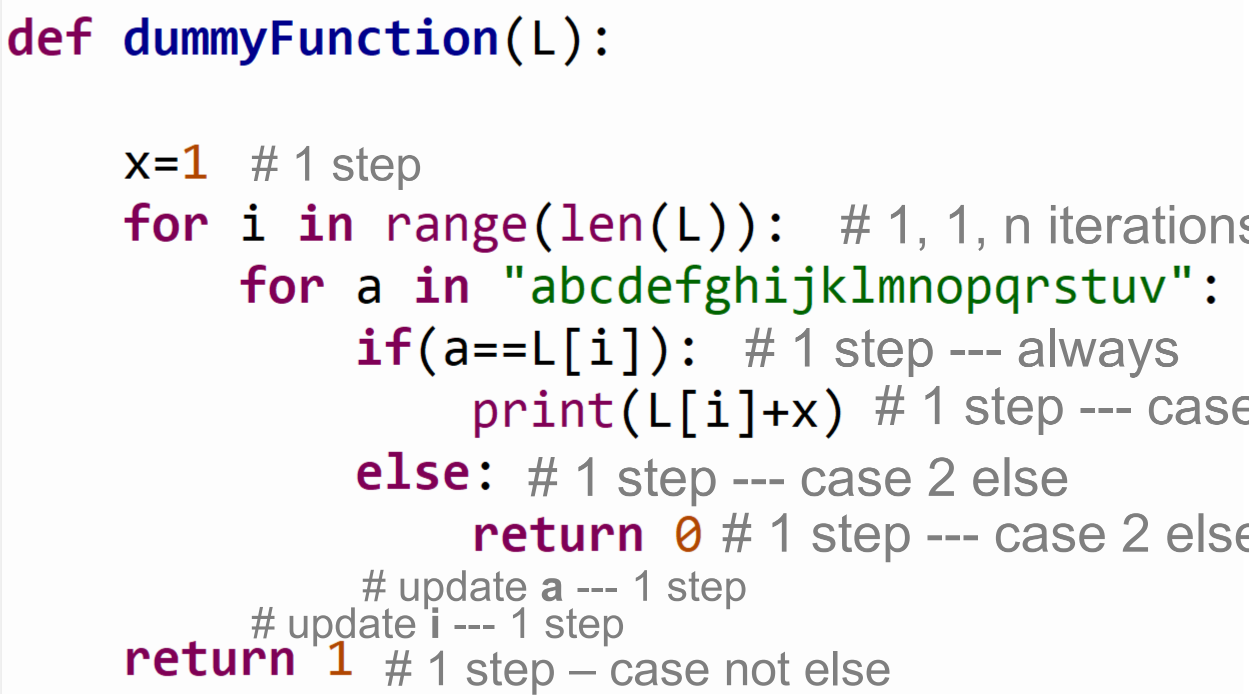
Total Number of Steps: 1+ n (2n+3) = $2n^2$ +3n+1

Practice



```python
def dummyFunction(L):

    x=1
    for i in range(len(L)):
        for a in "abcdefghijklmnopqrstuv":
            if(a==L[i]):
                print(L[i]+x)
            else:
                return 0

    return 1
```

Practice

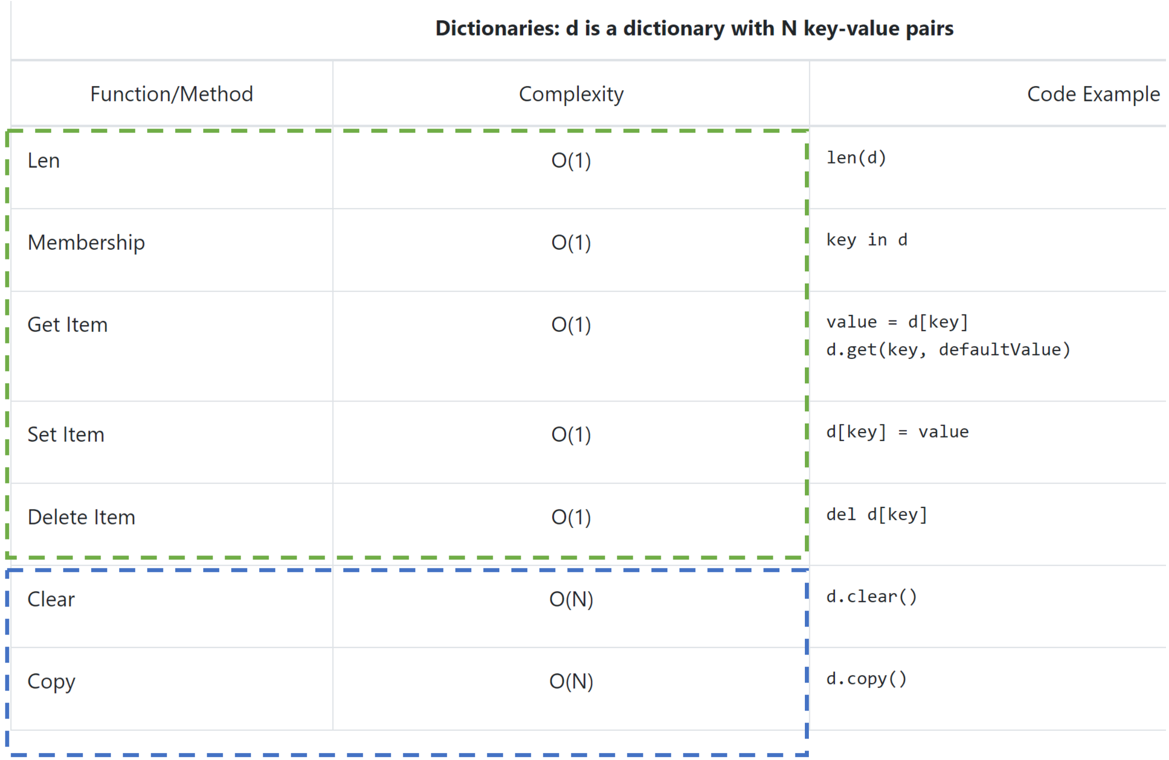


```
def dummyFunction(L):

    x=1  # 1 step
    for i in range(len(L)):  # 1, 1, n iterations
        for a in "abcdefghijklmnopqrstuv":  # 22 iterations
            if(a==L[i]):  # 1 step --- always
                print(L[i]+x) # 1 step --- case 1 if
            else:  # 1 step --- case 2 else
                return 0 # 1 step --- case 2 else
            # update a --- 1 step
        # update i --- 1 step
    return 1  # 1 step – case not else
```

If-else: 2
For a loop: 22* (1+1 +1)= 66
For i loop: 1+ n*67 + 1= 67n+2
Total= 67n+2+2= 67n+4

Python Built-Ins Cost

The efficiency of the built-in functions in Python will affect the efficiency of the functions they are used in. ([Built-in Functions Efficiency Table](#))



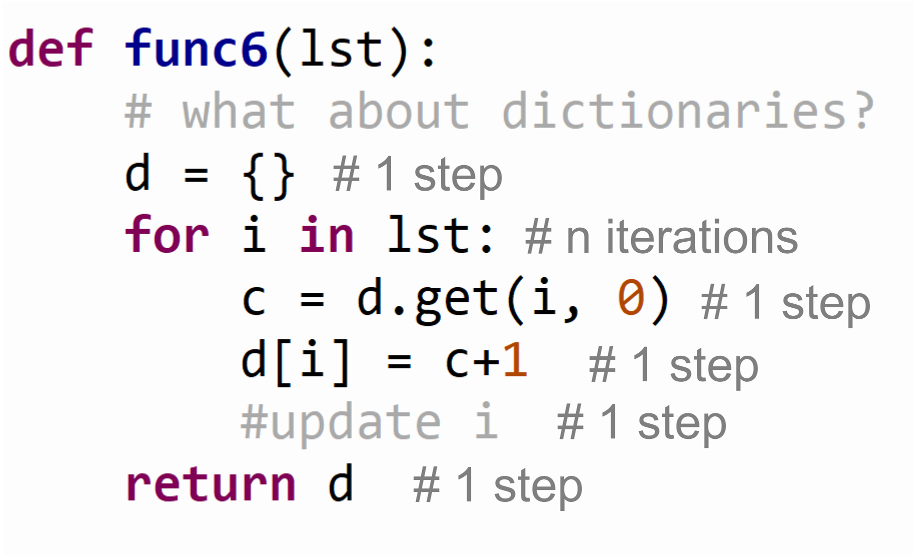
Dictionaries: d is a dictionary with N key-value pairs

Function/Method	Complexity	Code Example
Len	O(1)	`len(d)`
Membership	O(1)	`key in d`

Get Item	O(1)	`value = d[key]`
		`d.get(key, defaultValue)`

Set Item	O(1)	`d[key] = value`
Delete Item	O(1)	`del d[key]`
Clear	O(N)	`d.clear()`
Copy	O(N)	`d.copy()`

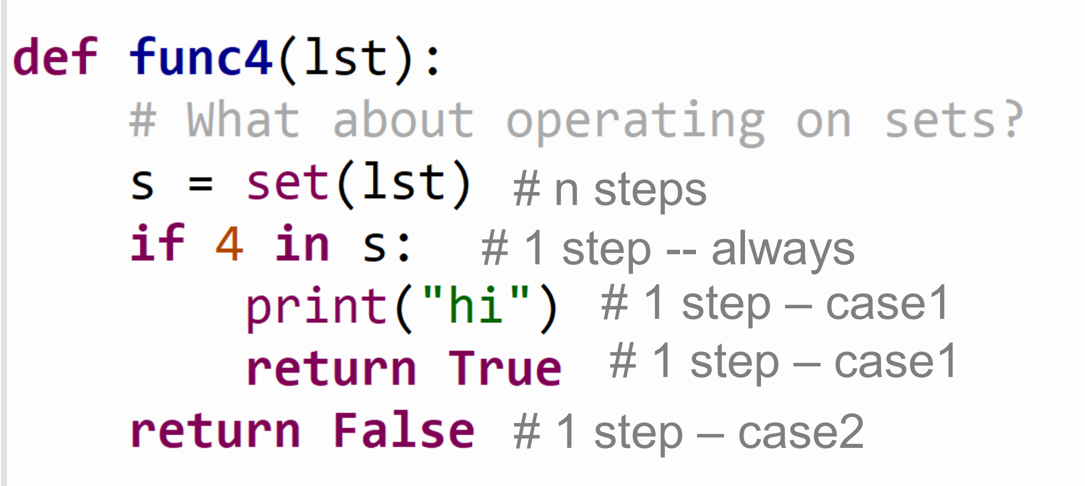
Total Steps: 3n + 2



```
def func6(lst):
    # what about dictionaries?
    d = {}  # 1 step
    for i in lst: # n iterations
        c = d.get(i, 0) # 1 step
        d[i] = c+1   # 1 step
        #update i   # 1 step
    return d   # 1 step
```

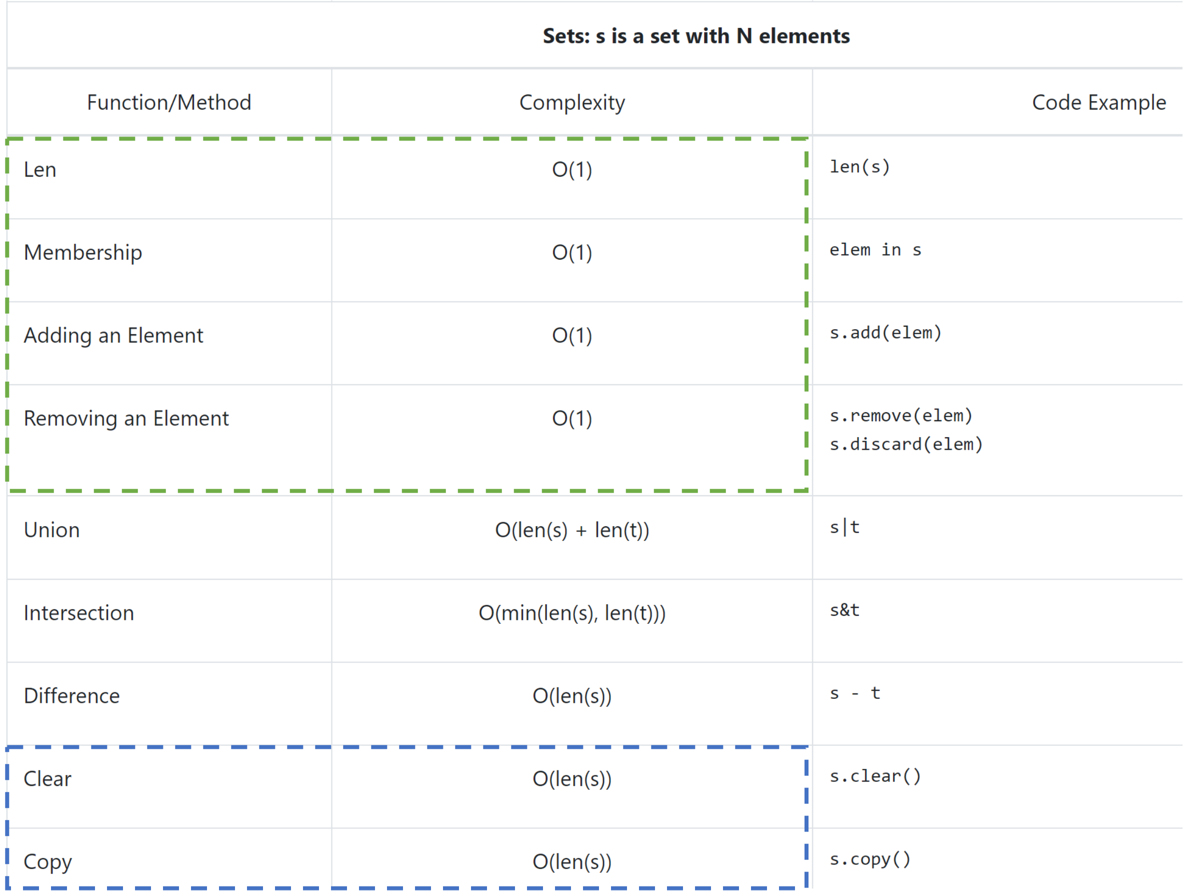
Python Built-Ins Cost

The efficiency of the built-in functions in Python will affect the efficiency of the functions they are used in.
([Built-in Functions Efficiency Table](#))



```
def func4(lst):
    # What about operating on sets?
    s = set(lst)  # n steps
    if 4 in s:   # 1 step -- always
        print("hi")  # 1 step – case1
        return True  # 1 step – case1
    return False  # 1 step – case2
```

Total steps: n + 3

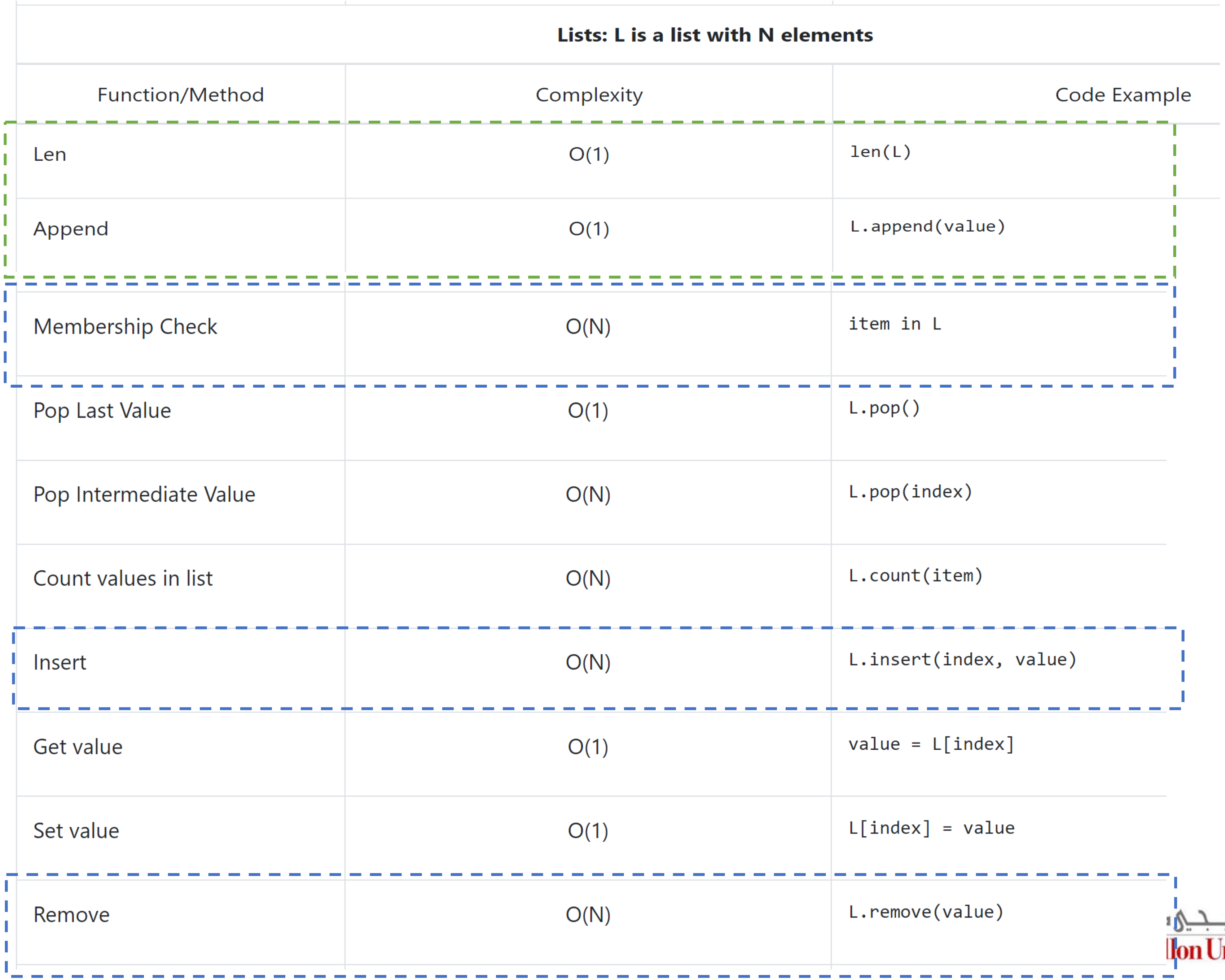


Sets: s is a set with N elements		
Function/Method	Complexity	Code Example
Len	O(1)	`len(s)`
Membership	O(1)	`elem in s`
Adding an Element	O(1)	`s.add(elem)`

Removing an Element	O(1)	`s.remove(elem)`
		`s.discard(elem)`

Union	O(len(s) + len(t))	`s|t`
Intersection	O(min(len(s), len(t)))	`s&t`
Difference	O(len(s))	`s - t`
Clear	O(len(s))	`s.clear()`
Copy	O(len(s))	`s.copy()`

Lists Compared to Sets/Dicts



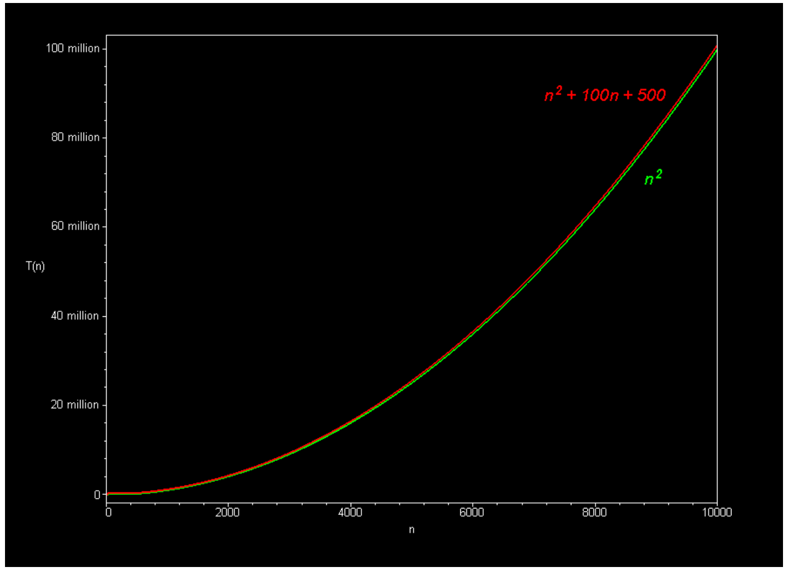
Lists: L is a list with N elements

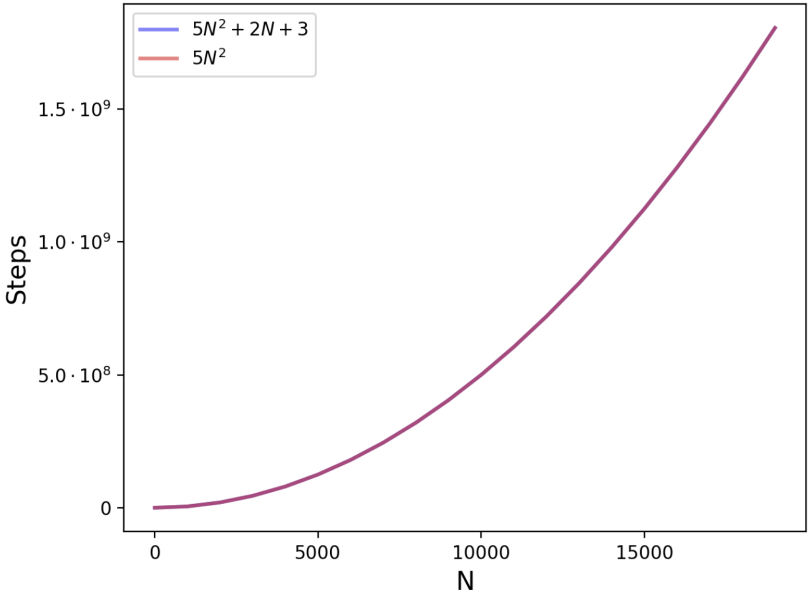
Function/Method	Complexity	Code Example
Len	O(1)	`len(L)`
Append	O(1)	`L.append(value)`
Membership Check	O(N)	`item in L`
Pop Last Value	O(1)	`L.pop()`
Pop Intermediate Value	O(N)	`L.pop(index)`
Count values in list	O(N)	`L.count(item)`
Insert	O(N)	`L.insert(index, value)`
Get value	O(1)	`value = L[index]`
Set value	O(1)	`L[index] = value`
Remove	O(N)	`L.remove(value)`

What is this O that appears with the complexity value

Ignoring Lower Order Terms

- Consider the following example complexity (steps count)
  - $N^2 + 100N + 500$
  - $5N^2 + 2N + 3$
- We say that N^2 is the **highest order term**. This is the term that grows the fastest.
  - The rest of the terms are called **lower order terms**
- **What would happen if we remove lower order Terms?**





In general, we ignore lower order terms for efficiency **because for large inputs, they make very little difference in the total**.

Ignoring Lower Order Terms

- We say that N^2 is the **highest order term**. This is the term that grows the fastest.
    - The rest of the terms are called **lower order terms**
- In general, we ignore lower order terms for efficiency **because for large inputs, they make very little difference in the total**.

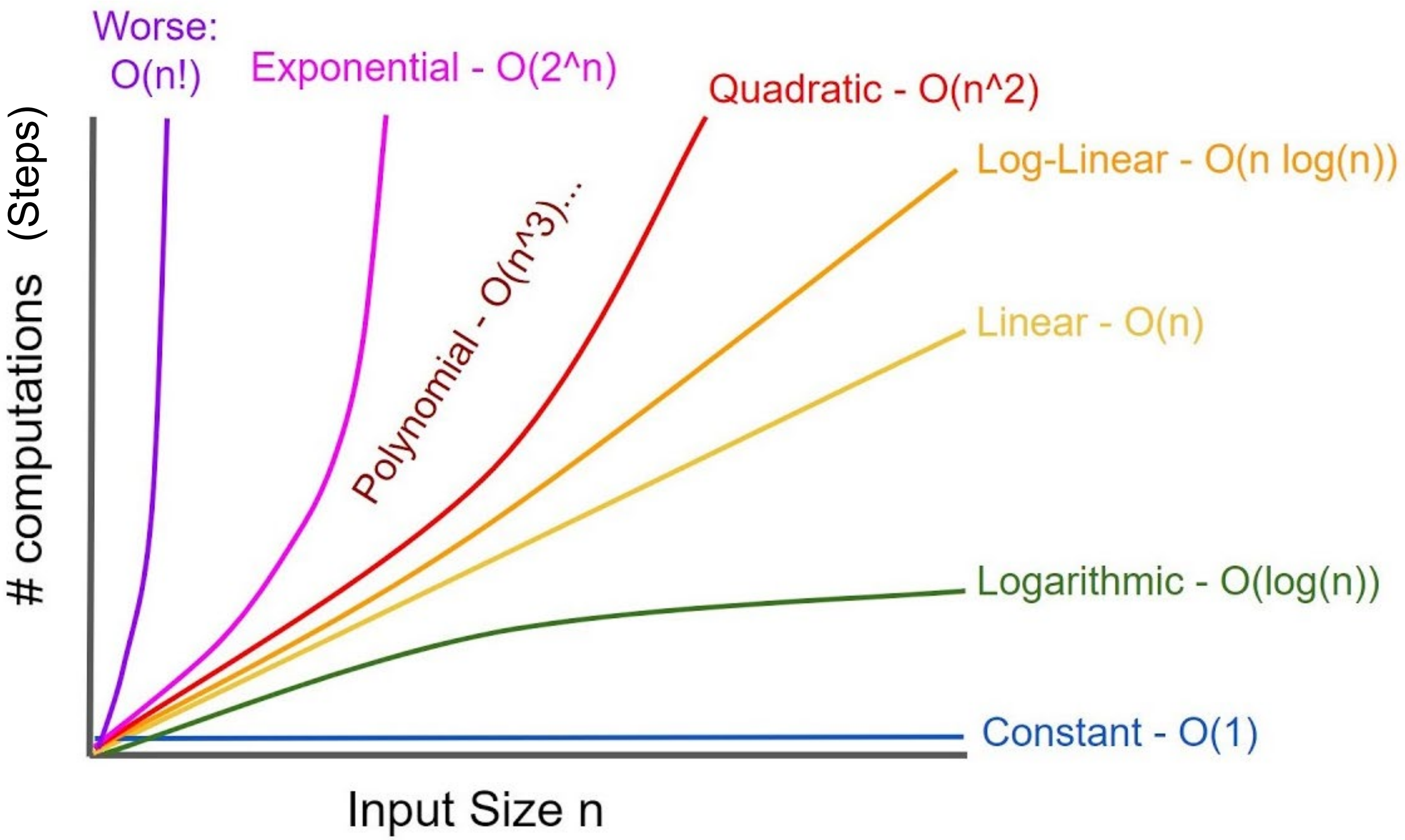
This is called BigO

The notion we use to describe the efficiency of a program, without considering lower order terms or coefficients.

BigO Function Families

We define **a function family** by the highest order term of a function without any coefficients.

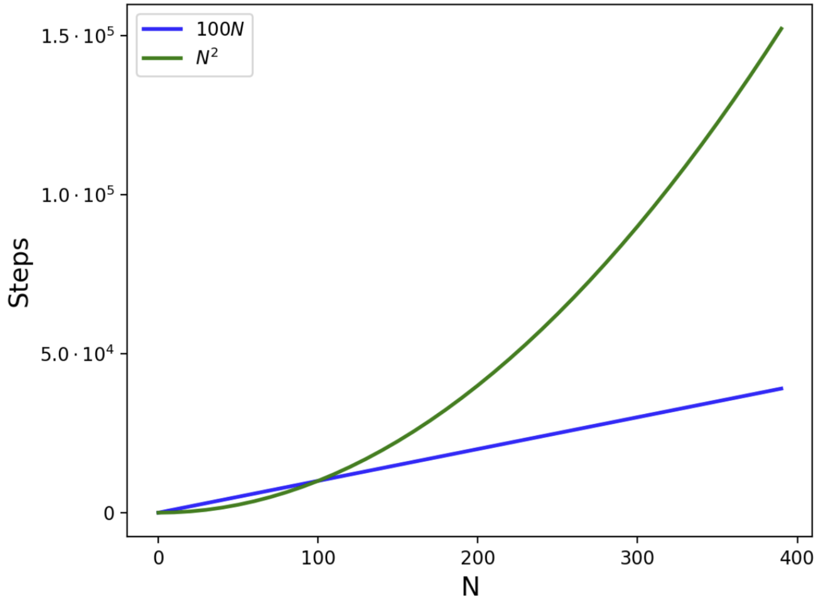
- For example, the N^2 (quadratic) function family, contains all the functions where the highest order term is N^2.
- Example functions that belong to the N^2 function family
  - $N^2+3N+25$
  - $3N^2 +30$
  - $100N^2+N$



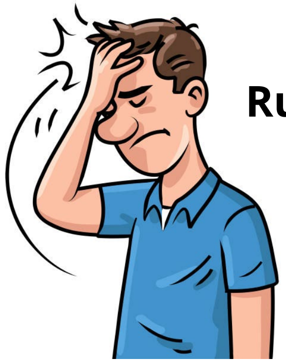
Big O – Ignoring Constants

Multiplying by a constant does not change the relationship between the function families.

- *A faster growing function family will always eventually overtake a slower growing function family.*
- This is why we ignore coefficients for efficiency and function families.



Does this mean you can change your algorithm's function family by just changing the hardware?



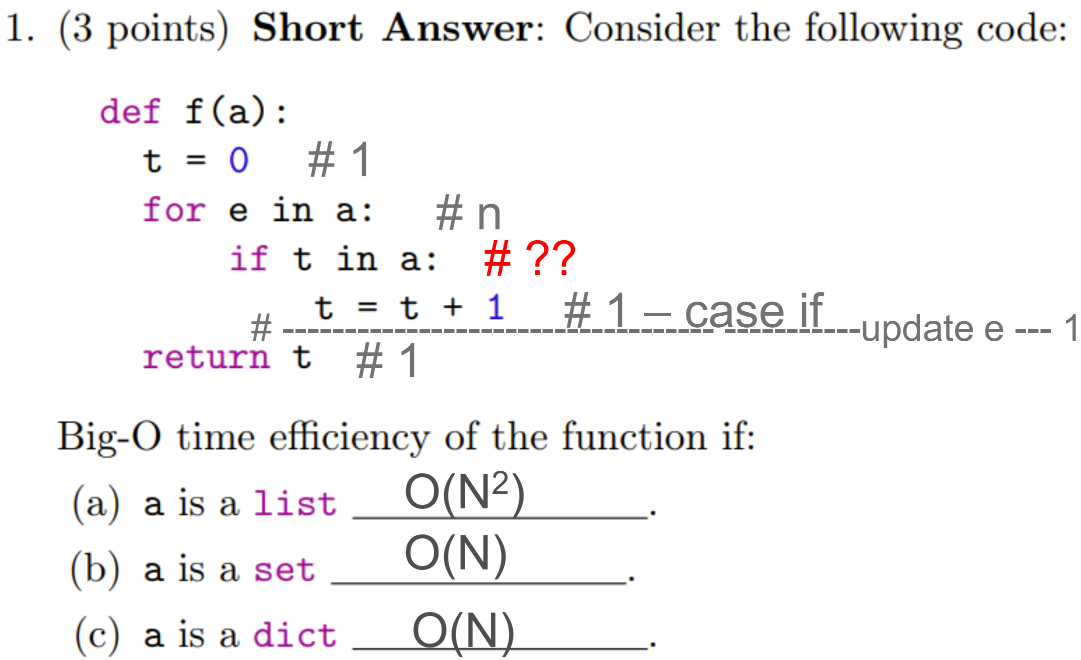
Running on a faster machine, can speed up our program by a constant factor.

You will not change your algorithm's function family by changing the hardware

Practice

Which Step highlights efficiency difference for these data structures??

S'23 Quiz Question



1. (3 points) **Short Answer**: Consider the following code:

```
def f(a):
  t = 0    # 1
  for e in a:    # n
      if t in a:    # ??
          t = t + 1    # 1 – case if
  # ------------------------------------------------update e --- 1 step
  return t    # 1
```

Big-O time efficiency of the function if:

(a) a is a list $O(N^2)$.

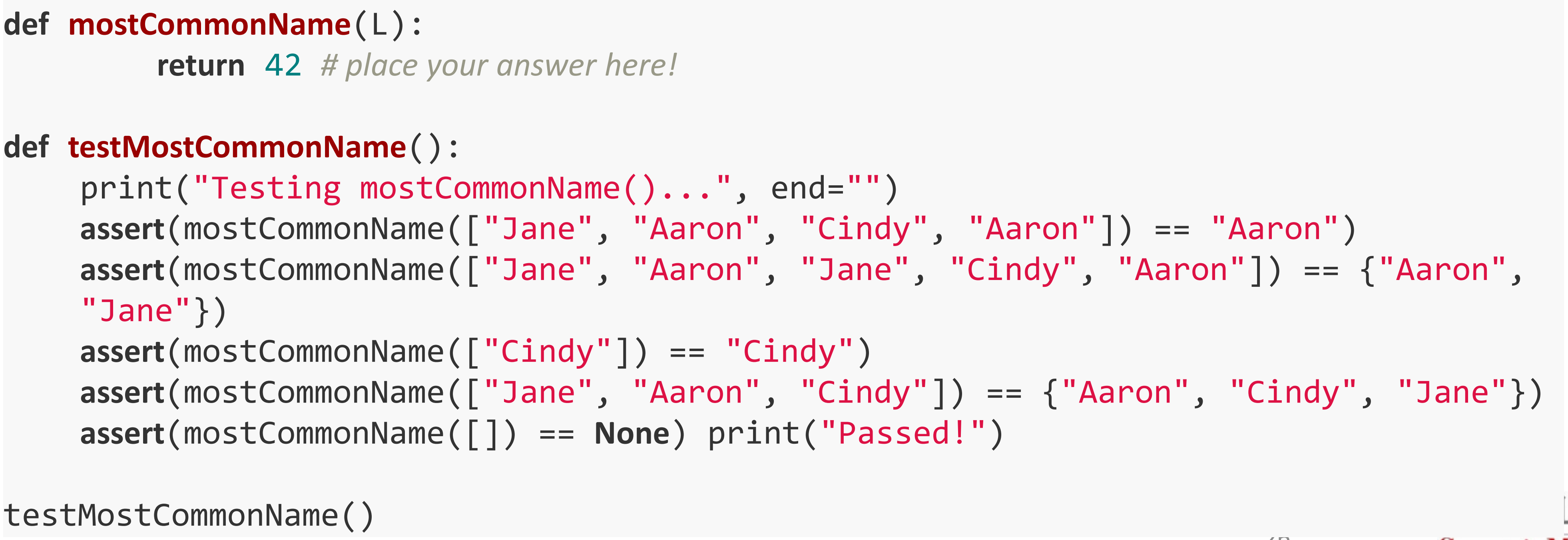
(b) a is a set O(N).

(c) a is a dict O(N).

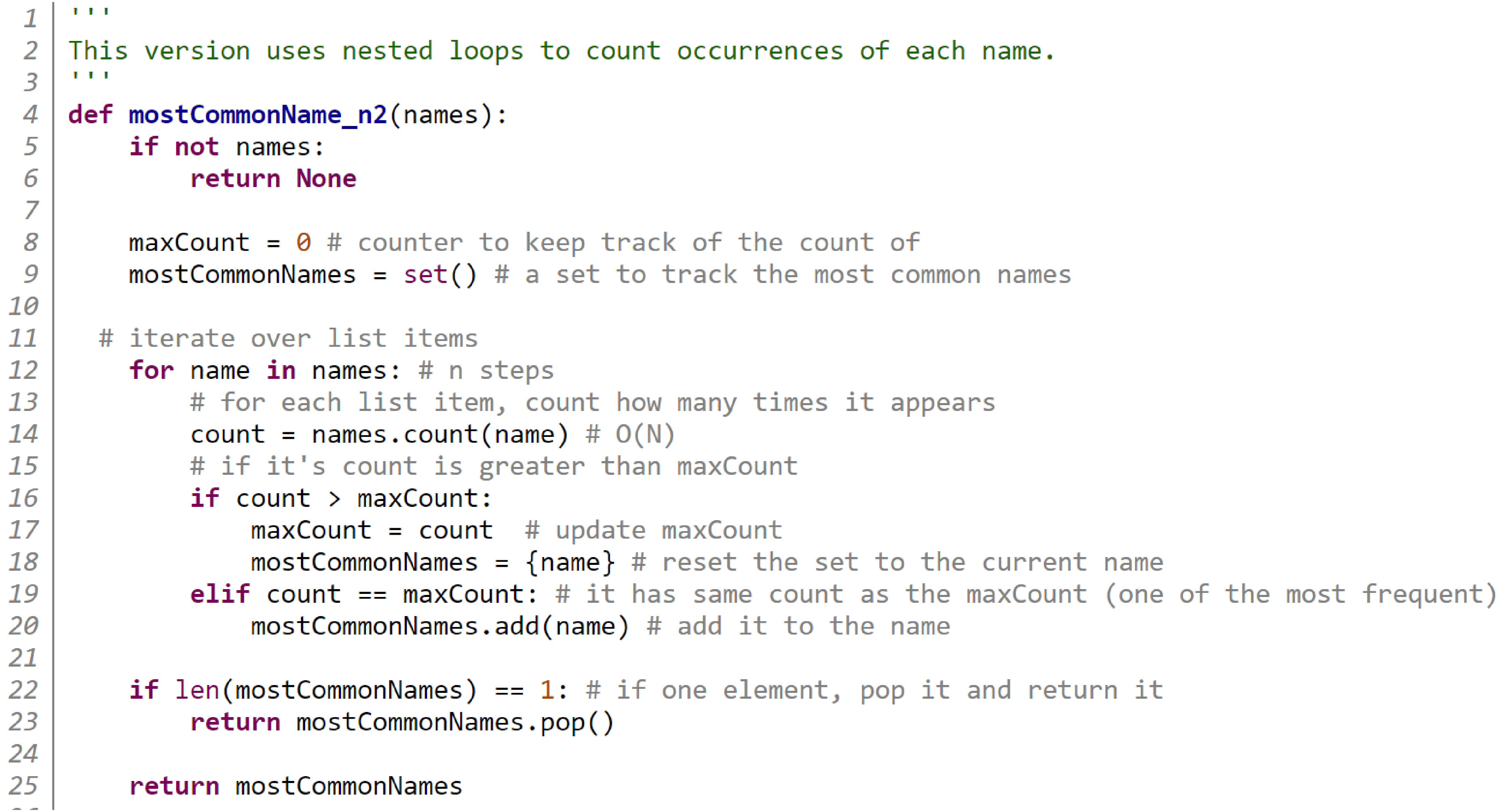
Practice – Free Response

mostCommonName(L)

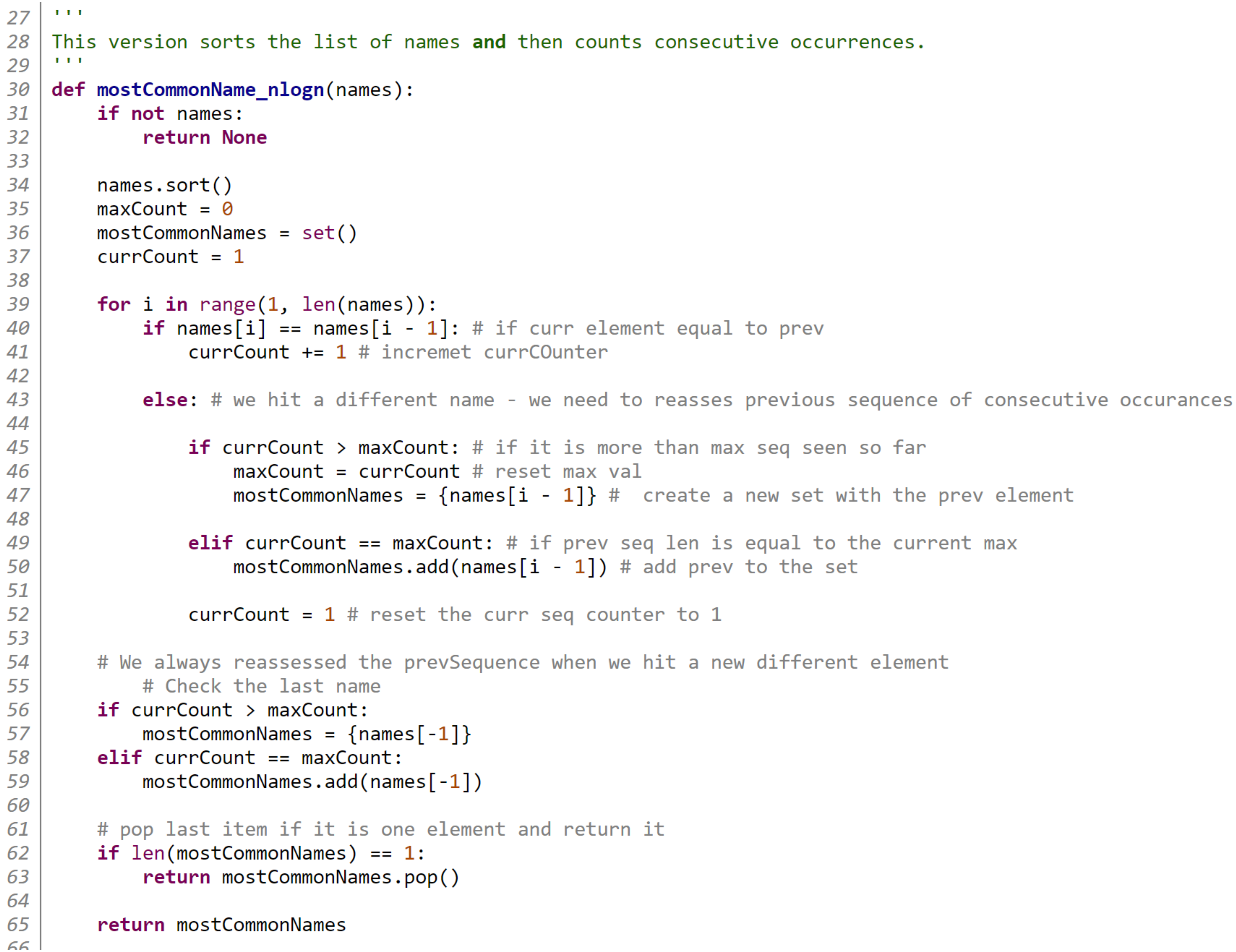
Write the function **mostCommonName**, that takes a list of names (such as ["Jane", "Aaron", "Cindy", "Aaron"], and returns the most common name in this list (in this case, "Aaron"). If there is more than one such name, return a set of the most common names. So mostCommonName(["Jane", "Aaron", "Jane", "Cindy", "Aaron"]) returns the set {"Aaron", "Jane"}. If the set is empty, return None. Also, treat names case sensitively, so "Jane" and "JANE" are different names. **You should write three different versions, one that runs in O(n**2), O(nlogn) and O(n).**



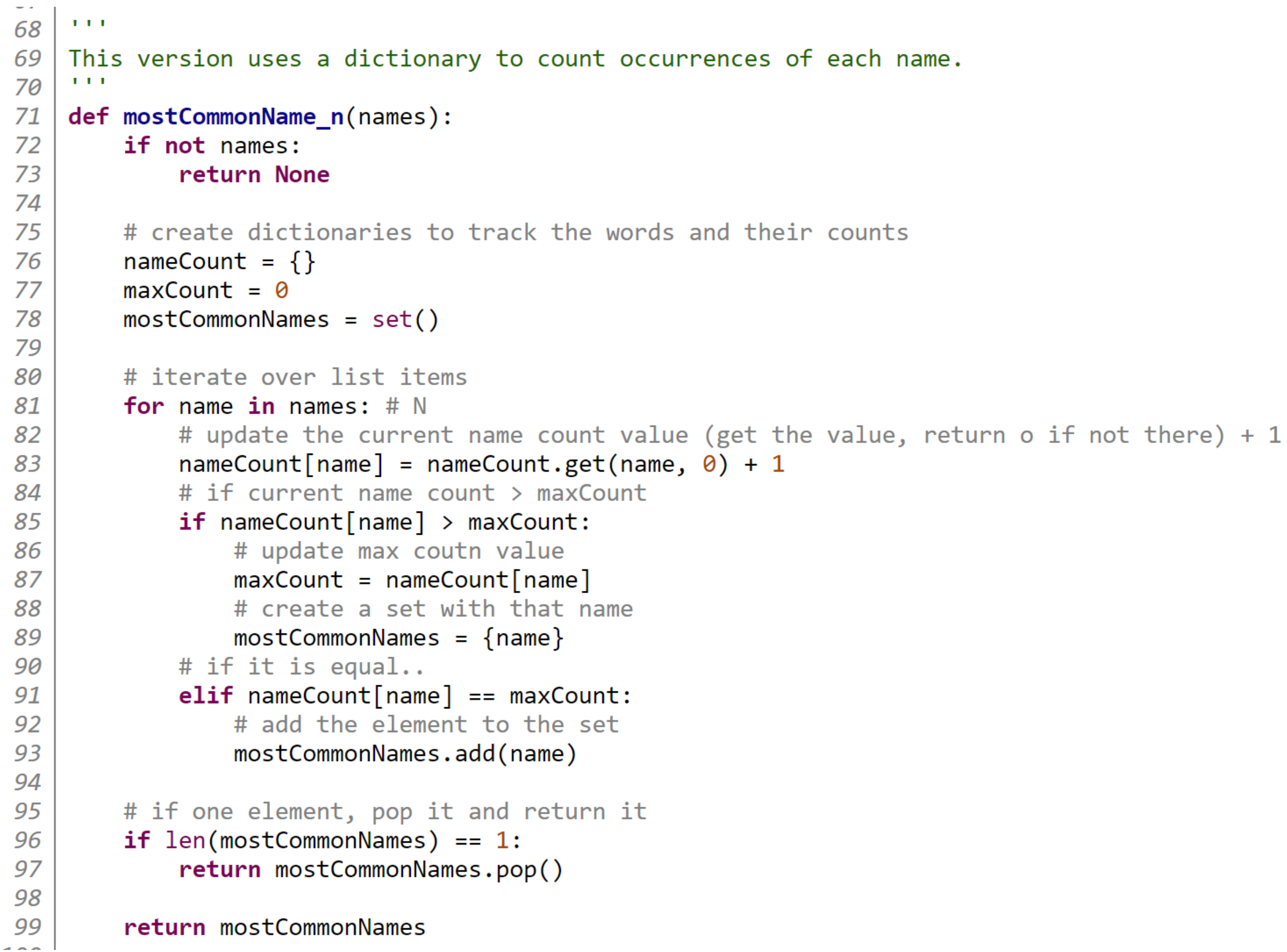
```
def mostCommonName(L):
        return 42 # place your answer here!

def testMostCommonName():
    print("Testing mostCommonName()...", end="")
    assert(mostCommonName(["Jane", "Aaron", "Cindy", "Aaron"]) == "Aaron")
    assert(mostCommonName(["Jane", "Aaron", "Jane", "Cindy", "Aaron"]) == {"Aaron",
    "Jane"})
    assert(mostCommonName(["Cindy"]) == "Cindy")
    assert(mostCommonName(["Jane", "Aaron", "Cindy"]) == {"Aaron", "Cindy", "Jane"})
    assert(mostCommonName([]) == None) print("Passed!")

testMostCommonName()
```



```python
'''
This version uses nested loops to count occurrences of each name.
'''
def mostCommonName_n2(names):
    if not names:
        return None

    maxCount = 0 # counter to keep track of the count of
    mostCommonNames = set() # a set to track the most common names

  # iterate over list items
    for name in names: # n steps
        # for each list item, count how many times it appears
        count = names.count(name) # O(N)
        # if it's count is greater than maxCount
        if count > maxCount:
            maxCount = count  # update maxCount
            mostCommonNames = {name} # reset the set to the current name
        elif count == maxCount: # it has same count as the maxCount (one of the most frequent)
            mostCommonNames.add(name) # add it to the name

    if len(mostCommonNames) == 1: # if one element, pop it and return it
        return mostCommonNames.pop()

    return mostCommonNames
```

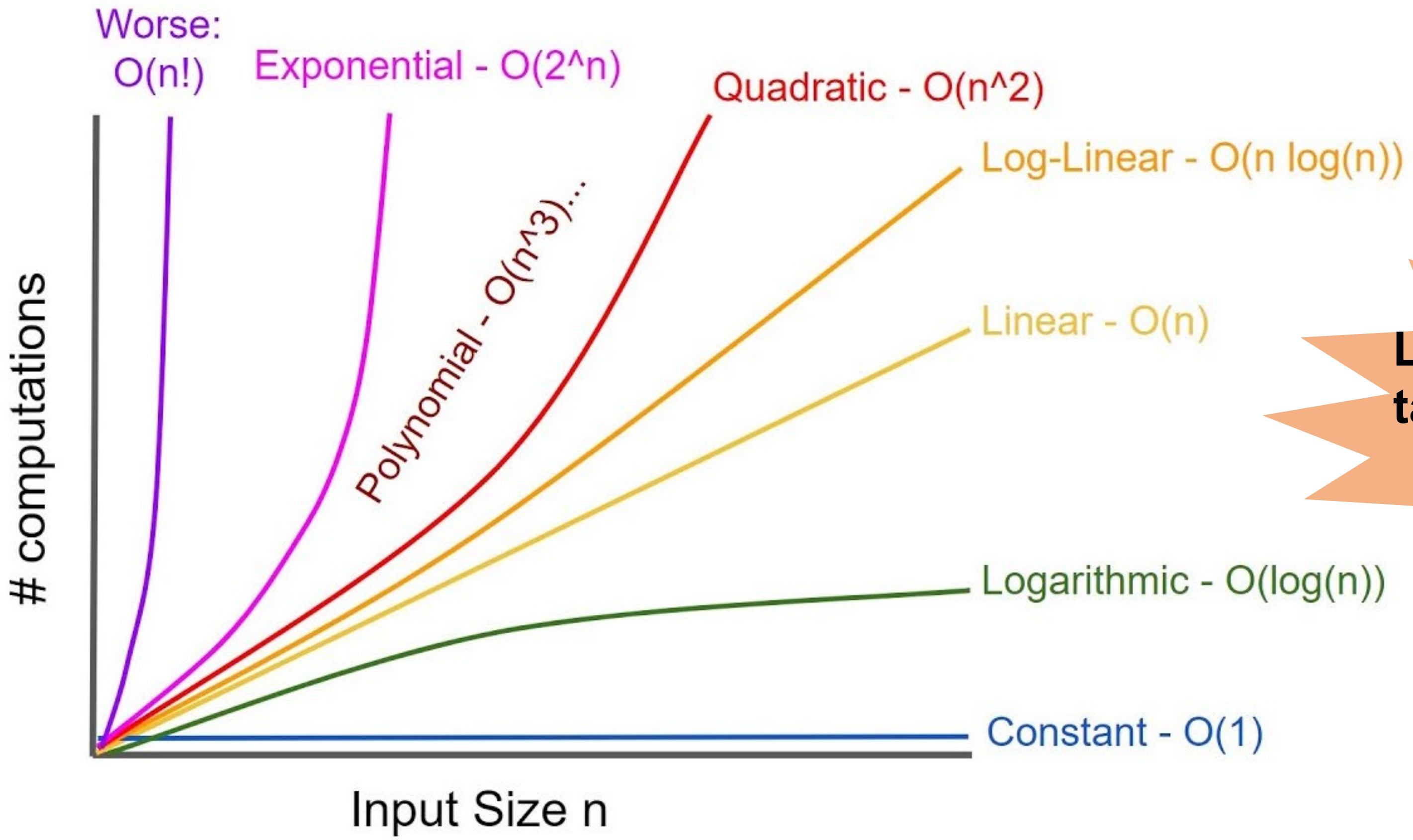


```python
'''
This version sorts the list of names and then counts consecutive occurrences.
'''
def mostCommonName_nlogn(names):
    if not names:
        return None

    names.sort()
    maxCount = 0
    mostCommonNames = set()
    currCount = 1

    for i in range(1, len(names)):
        if names[i] == names[i - 1]: # if curr element equal to prev
            currCount += 1 # incremet currCOunter

        else: # we hit a different name - we need to reasses previous sequence of consecutive occurances

            if currCount > maxCount: # if it is more than max seq seen so far
                maxCount = currCount # reset max val
                mostCommonNames = {names[i - 1]} #  create a new set with the prev element

            elif currCount == maxCount: # if prev seq len is equal to the current max
                mostCommonNames.add(names[i - 1]) # add prev to the set

            currCount = 1 # reset the curr seq counter to 1

    # We always reassessed the prevSequence when we hit a new different element
        # Check the last name
    if currCount > maxCount:
        mostCommonNames = {names[-1]}
    elif currCount == maxCount:
        mostCommonNames.add(names[-1])

    # pop last item if it is one element and return it
    if len(mostCommonNames) == 1:
        return mostCommonNames.pop()

    return mostCommonNames
```



```python
'''
This version uses a dictionary to count occurrences of each name.
'''
def mostCommonName_n(names):
    if not names:
        return None

    # create dictionaries to track the words and their counts
    nameCount = {}
    maxCount = 0
    mostCommonNames = set()

    # iterate over list items
    for name in names: # N
        # update the current name count value (get the value, return o if not there) + 1
        nameCount[name] = nameCount.get(name, 0) + 1
        # if current name count > maxCount
        if nameCount[name] > maxCount:
            # update max coutn value
            maxCount = nameCount[name]
            # create a set with that name
            mostCommonNames = {name}
        # if it is equal..
        elif nameCount[name] == maxCount:
            # add the element to the set
            mostCommonNames.add(name)

    # if one element, pop it and return it
    if len(mostCommonNames) == 1:
        return mostCommonNames.pop()

    return mostCommonNames
```

What is Log?



L.sort() & sorted(L) take O(nlogn)

What is Log?

Think of it as **repeated division**

For (log N), Starting at the number N, How many times do we need to divide by 2 to get to 1

Log(8) = ??

8/2= 4
4/2=2
2/2= 1

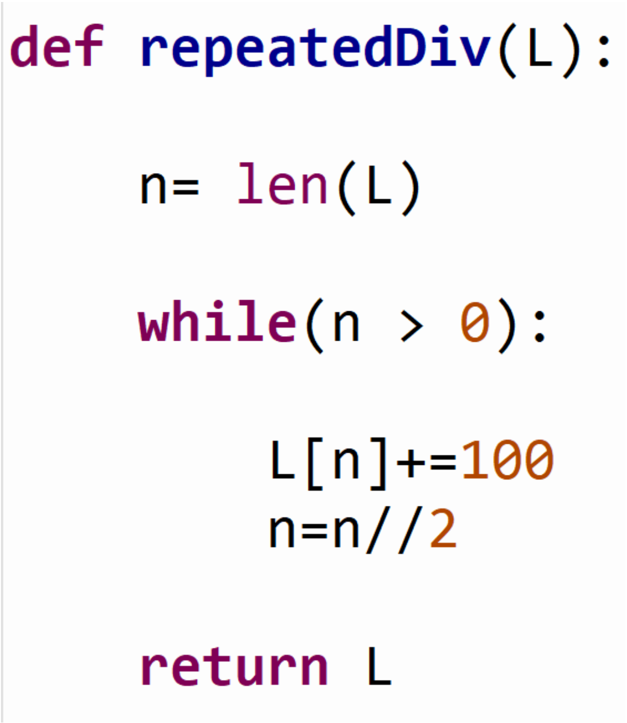
Log (8) = 3

What is Log?

Think of it as **repeated division**

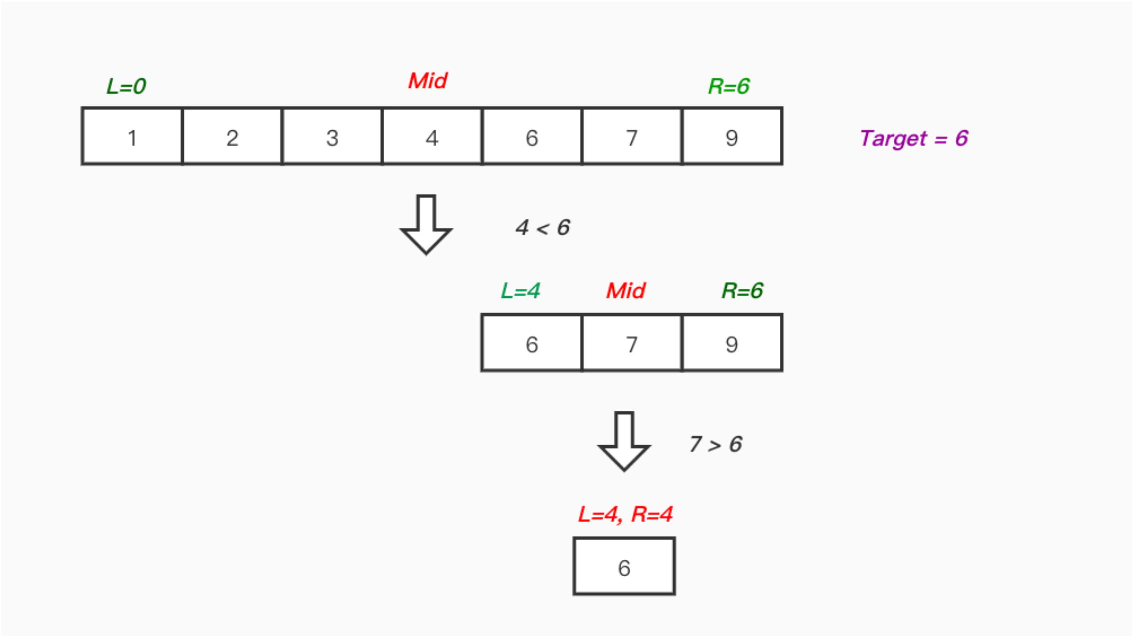
Starting at the number, How many times do we need to divide by 2 to get to 1

Often come up in code when we are repeatedly cutting our input size (n) in half



```
def repeatedDiv(L):

    n= len(L)

    while(n > 0):

        L[n]+=100
        n=n//2

    return L
```

Real Algorithm Example



Binary Search

Why is it O(LogN) ???

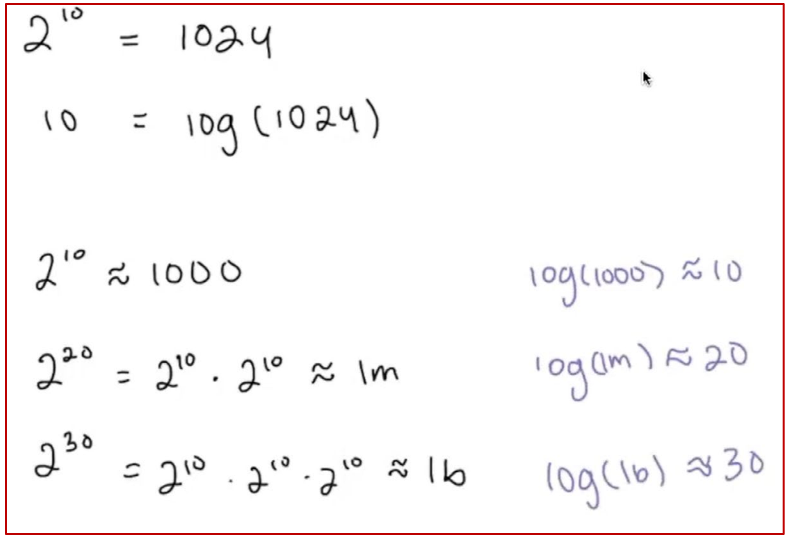
At every iteration, you are getting rid of half of the list
So you are repeatedly dividing the input size by half

Why is Log Fast?

- We can see that **log takes big numbers and converts them into much smaller numbers**

- So if your algorithm has log(n) complexity, this means that if your input size is:
  - Thousand – 10 steps
  - million – it will only take 20 steps
  - Billion- 30 steps
  - Trillion – 40 steps

- Your algorithm will run very fast for large inputs.
  - **Logs are very small**



$2^{10} = 1024$

$10 = \log(1024)$

$2^{10} \approx 1000$ $\log(1000) \approx 10$

$2^{20} = 2^{10} \cdot 2^{10} \approx 1m$ $\log(1m) \approx 20$

$2^{30} = 2^{10} \cdot 2^{10} \cdot 2^{10} \approx 1b$ $\log(1b) \approx 30$

Recap

- Steps Counting gives a standard way to assess time efficiency of an algorithm regardless of the hardware on which the algorithm is running
  - While elapsed time for a given algorithm varies depending on different factors such as hardware specifications, operating system, and resource utilizations.
- Two rules for counting steps
  - A step takes constant amount of time (i.e. time doesn’t increase as the input size (called n) increases)
  - Generally, A line of code is a single step if the whole line runs in constant time
- We consider highest order term in an efficiency function and ignore lower order terms
  - because for large inputs, they make very little difference
- BigO is The notion we use to describe the efficiency of a program, without considering lower order terms or coefficients.
- We define a function family by the highest order term of a function without any coefficients (Big O function families)
  - For example, the N^2 (quadratic) function family, contains all the functions where the highest order term is N^2.
- Built-in Functions Efficiency Table
- Multiplying by a constant does not change the relationship between the function families.
- Running the program on a faster hardware only improves time performance by a constant factor