ECS32B

Introduction to Data Structures

Midterm Review Session

Lecture 15

Announcements

- The midterm is next Monday May 6.

  Open notes (2 pages front and back), closed book, no electronics. Complete instructions on the sample midterm

- TAs will monitor Piazza for your questions

- Homework 4 will be assigned next Tuesday May 7 and will be due the following Tuesday May 14 at 11:59pm

Strategy Suggestions

- Do the sample midterm.
- Understand the sample midterm.
- Read the instructions on the sample midterm. You will be responsible for knowing these.
- Save time on Monday.
- Work on your 2 pages of notes front and back this weekend. Hand written may be better for you, you may just remember what you wrote down.

Strategy Suggestions

- It’s long. After doing the home work and taking the sample midterm, there may be a topic and style of question you can answer quickly. If you recognize those, do them first.
- If you understand the homework and lecture slides no question should be too difficult to answer.
- Don’t panic if you can’t answer every question. Remember the curve.

Midterm Topics

- We will focus on the topics covered in lecture slides and on the homework assignments in these three areas:

  1. Algorithm Analysis

  2. Linear Data Structures

  3. Recursion

Algorithm Analysis

Know these

- Formal definition of Big-O
- Formal definition of Big-Omega
- Formal definition of Big-Theta

What they mean

- Big-O

  e.g. T(n) is O(n) if some linear function is an upper bound for T(n).

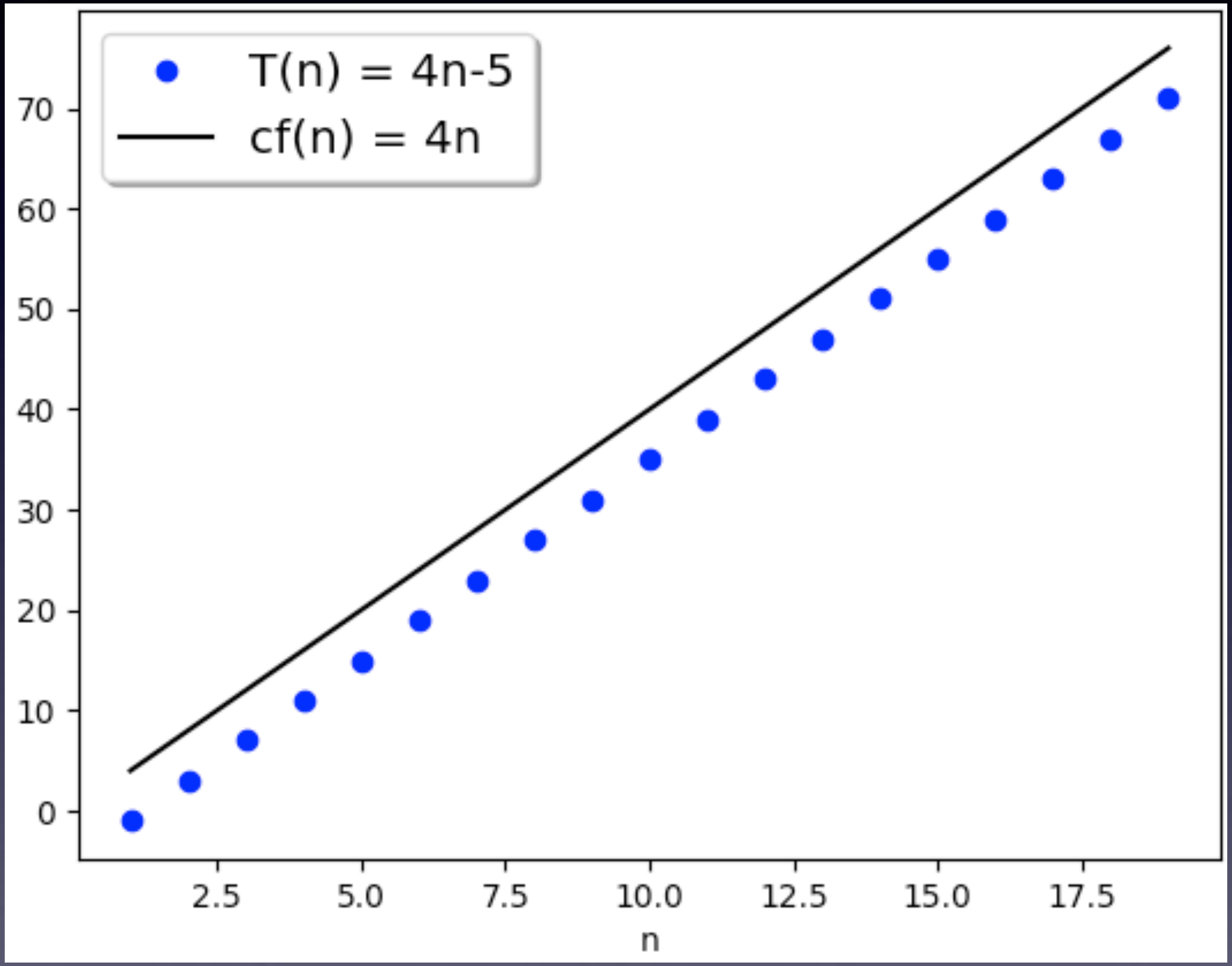
- Big-Omega aka Big-Ω

  e.g. T(n) is Omega(n^2) if some quadratic function is a lower bound for T(n).

- Big-Theta aka Big-Θ

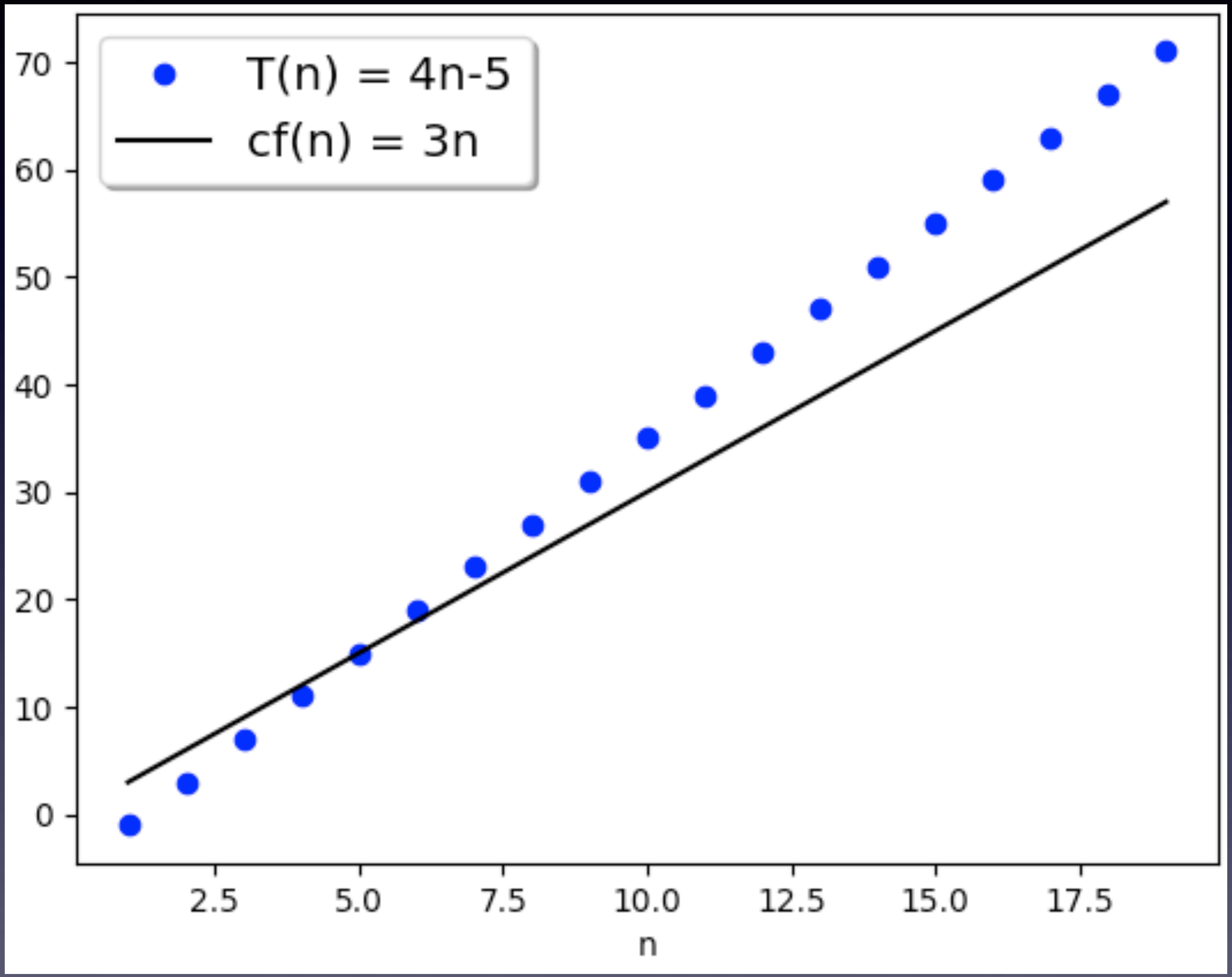
  e.g. T(n) is Theta(n^2) if and only if T(n) is O(n^2) and T(n) is Omega(n^2)

4n is an upper bound for T(n) = 4n-5 so T(n) is O(n)



What are c and n_0?

3n is a lower bound for T(n) = 4n-5 so T(n) is Omega(n)



What are c and n_0?

Algorithm Analysis

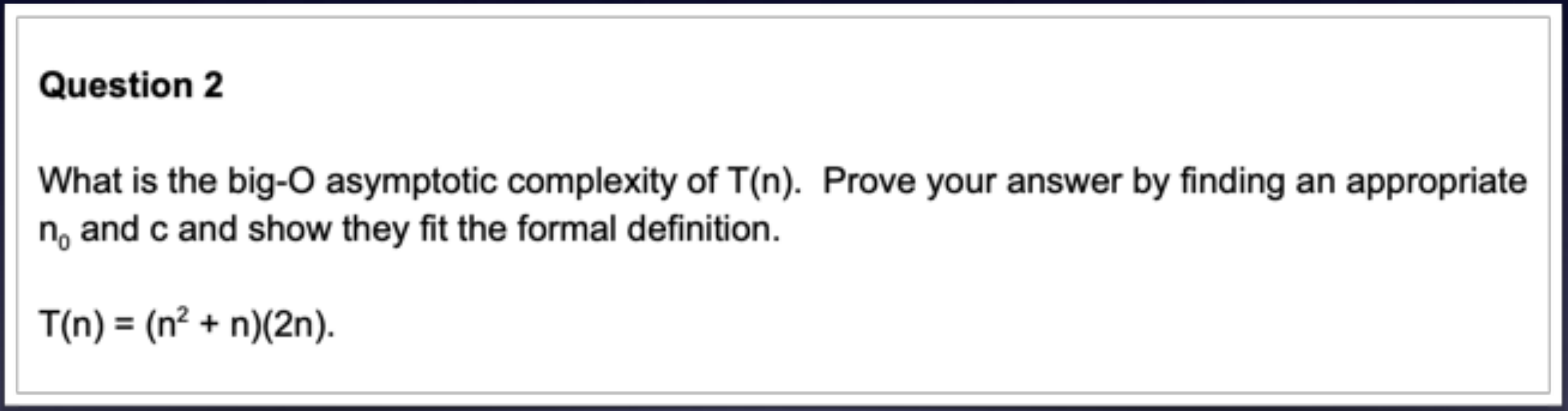
- Order of Dominance
- Analyzing code
  - Looking at code/pseudocode and determining complexity
- Complexity of data structure operations
  - Stack, Queue, Deque, Linked List, Python List
- Estimating runtime/problem size for a given complexity e.g. $O(2^n)$

"Order of dominance" table

The "order of dominance" is in this table. Orders closer to the bottom are faster growing. Read from the bottom up. O(n!) dominates $O(2^n)$. $O(2^n)$ dominates $O(n^k)$. And so on...

Big O	**Name**
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	**Quadratic**
$O(n^3)$	Cubic
$O(n^k)$	Polynomial – k is constant
$O(2^n)$	Exponential
$O(n!)$	Factorial

Big-O Proof

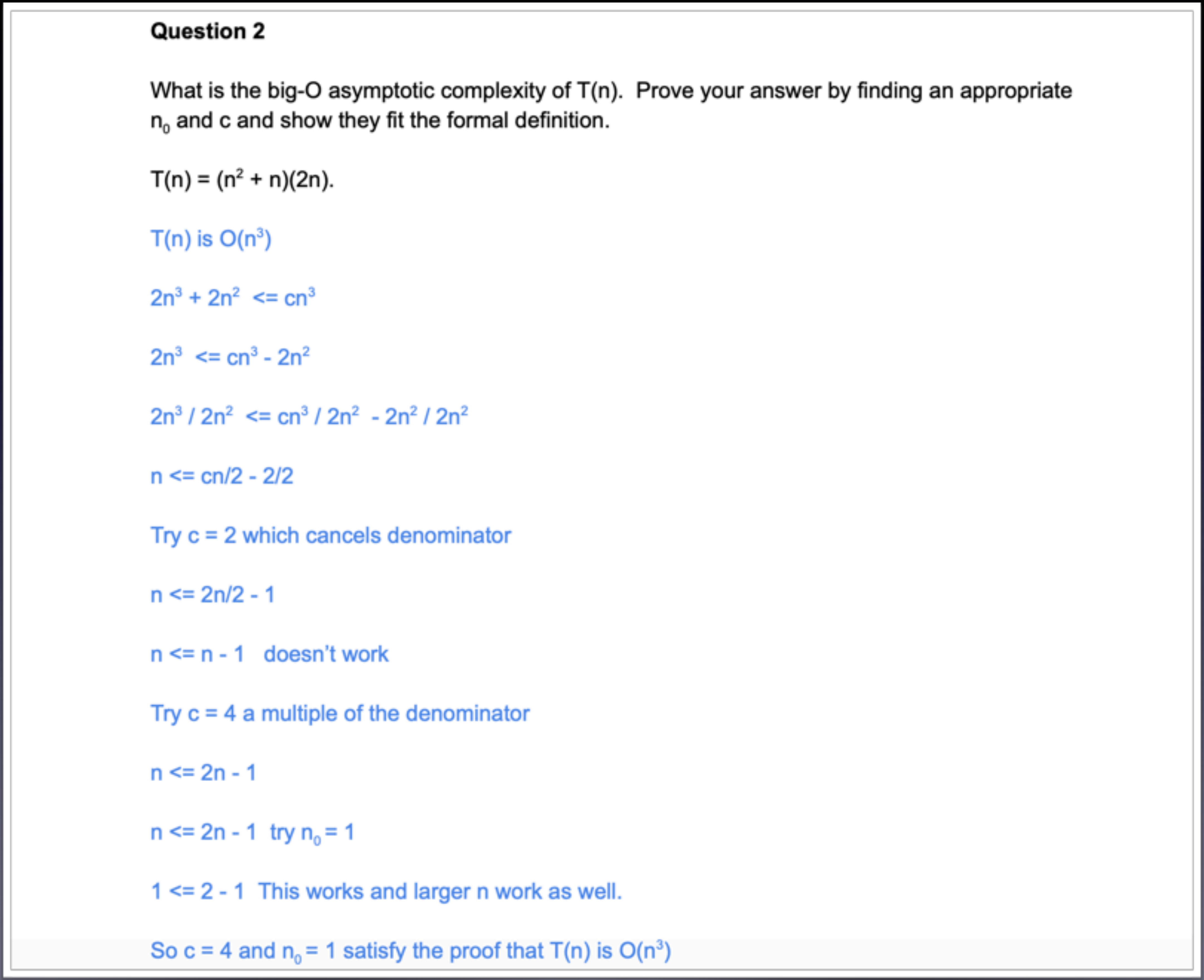


Question 2

What is the big-O asymptotic complexity of T(n). Prove your answer by finding an appropriate n_0 and c and show they fit the formal definition.

$T(n) = (n^2 + n)(2n)$.

Big-O Proof



Question 2

What is the big-O asymptotic complexity of T(n). Prove your answer by finding an appropriate n_0 and c and show they fit the formal definition.

$T(n) = (n^2 + n)(2n)$.

T(n) is $O(n^3)$

$2n^3 + 2n^2 <= cn^3$

$2n^3 <= cn^3 - 2n^2$

$2n^3 / 2n^2 <= cn^3 / 2n^2 - 2n^2 / 2n^2$

$n <= cn/2 - 2/2$

Try c = 2 which cancels denominator

$n <= 2n/2 - 1$

$n <= n - 1$ doesn't work

Try c = 4 a multiple of the denominator

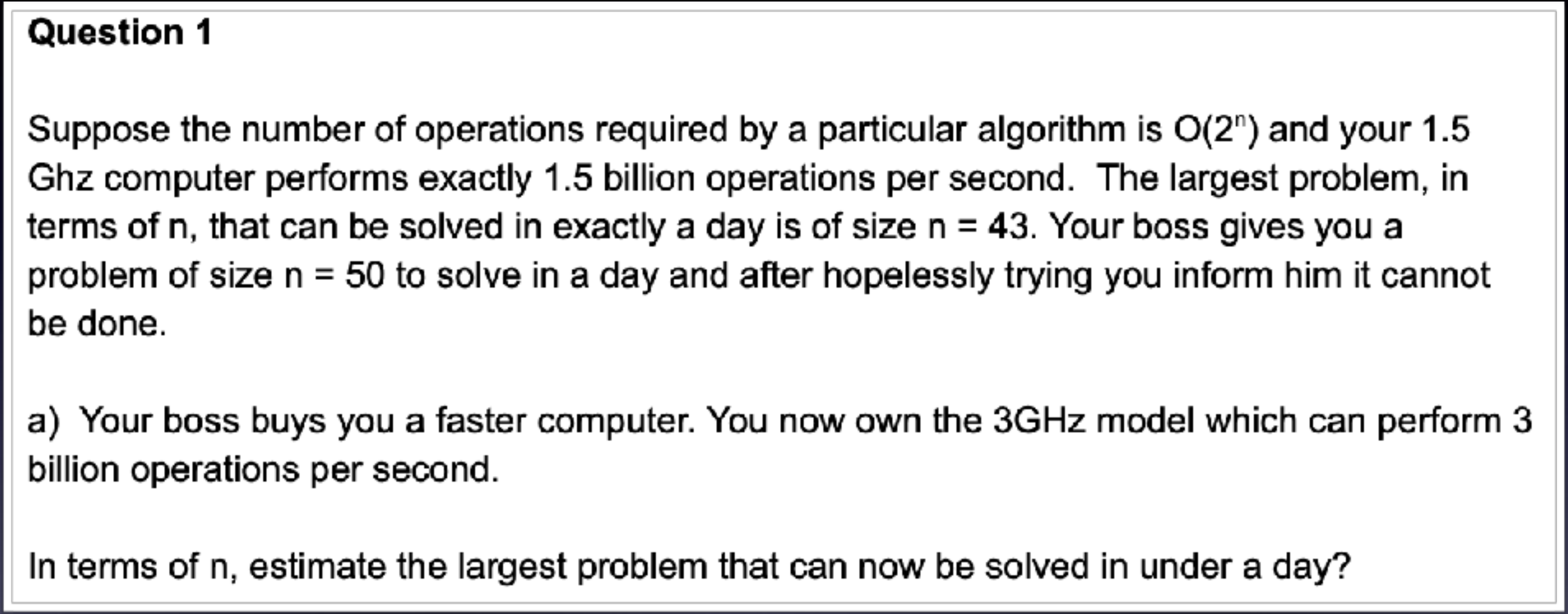
$n <= 2n - 1$

$n <= 2n - 1$ try $n_0 = 1$

$1 <= 2 - 1$ This works and larger n work as well.

So c = 4 and $n_0 = 1$ satisfy the proof that T(n) is $O(n^3)$

Estimating problem size



Question 1

Suppose the number of operations required by a particular algorithm is $O(2^n)$ and your 1.5 Ghz computer performs exactly 1.5 billion operations per second. The largest problem, in terms of n, that can be solved in exactly a day is of size n = 43. Your boss gives you a problem of size n = 50 to solve in a day and after hopelessly trying you inform him it cannot be done.

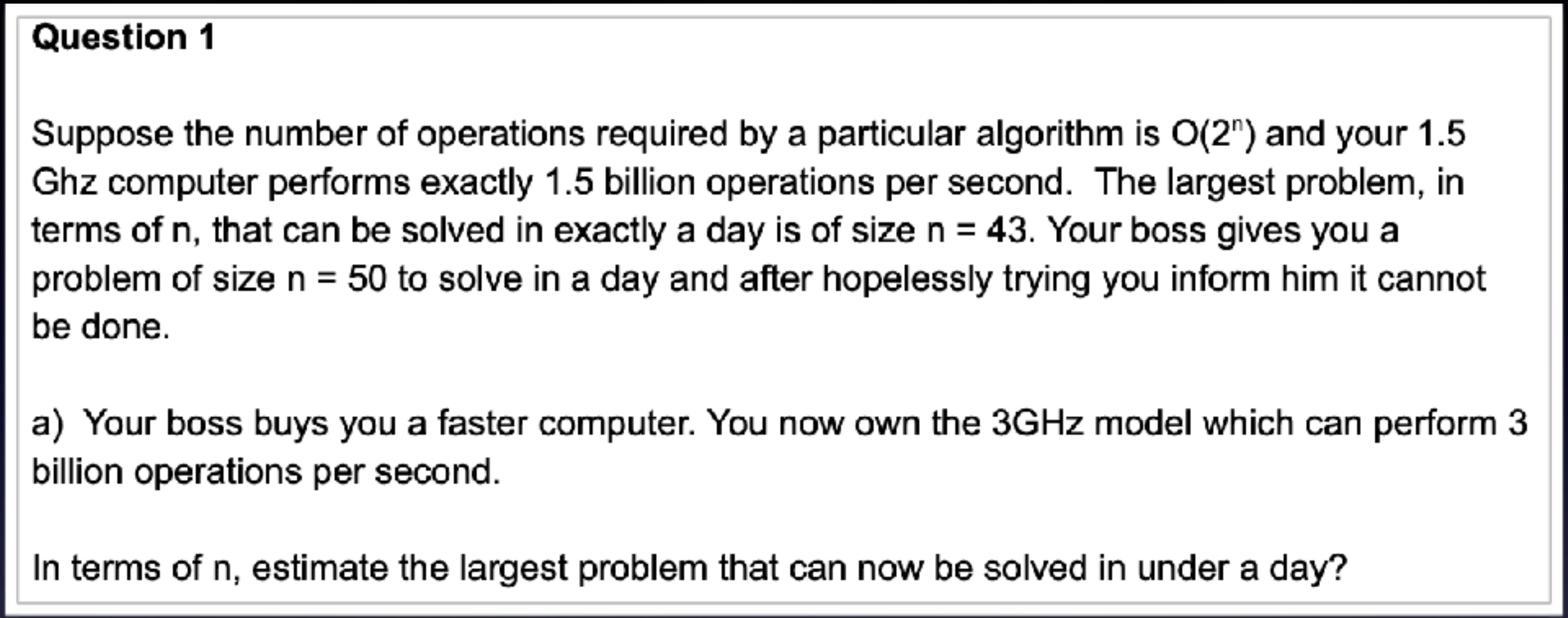
a) Your boss buys you a faster computer. You now own the 3GHz model which can perform 3 billion operations per second.

In terms of n, estimate the largest problem that can now be solved in under a day?

For estimation purposes,
if $T(n)$ is $O(2^n)$ then just assume $T(n) = c2^n$

What is $c2^{n+1}$

Estimating problem size



Question 1

Suppose the number of operations required by a particular algorithm is $O(2^n)$ and your 1.5 Ghz computer performs exactly 1.5 billion operations per second. The largest problem, in terms of n, that can be solved in exactly a day is of size n = 43. Your boss gives you a problem of size n = 50 to solve in a day and after hopelessly trying you inform him it cannot be done.

a) Your boss buys you a faster computer. You now own the 3GHz model which can perform 3 billion operations per second.

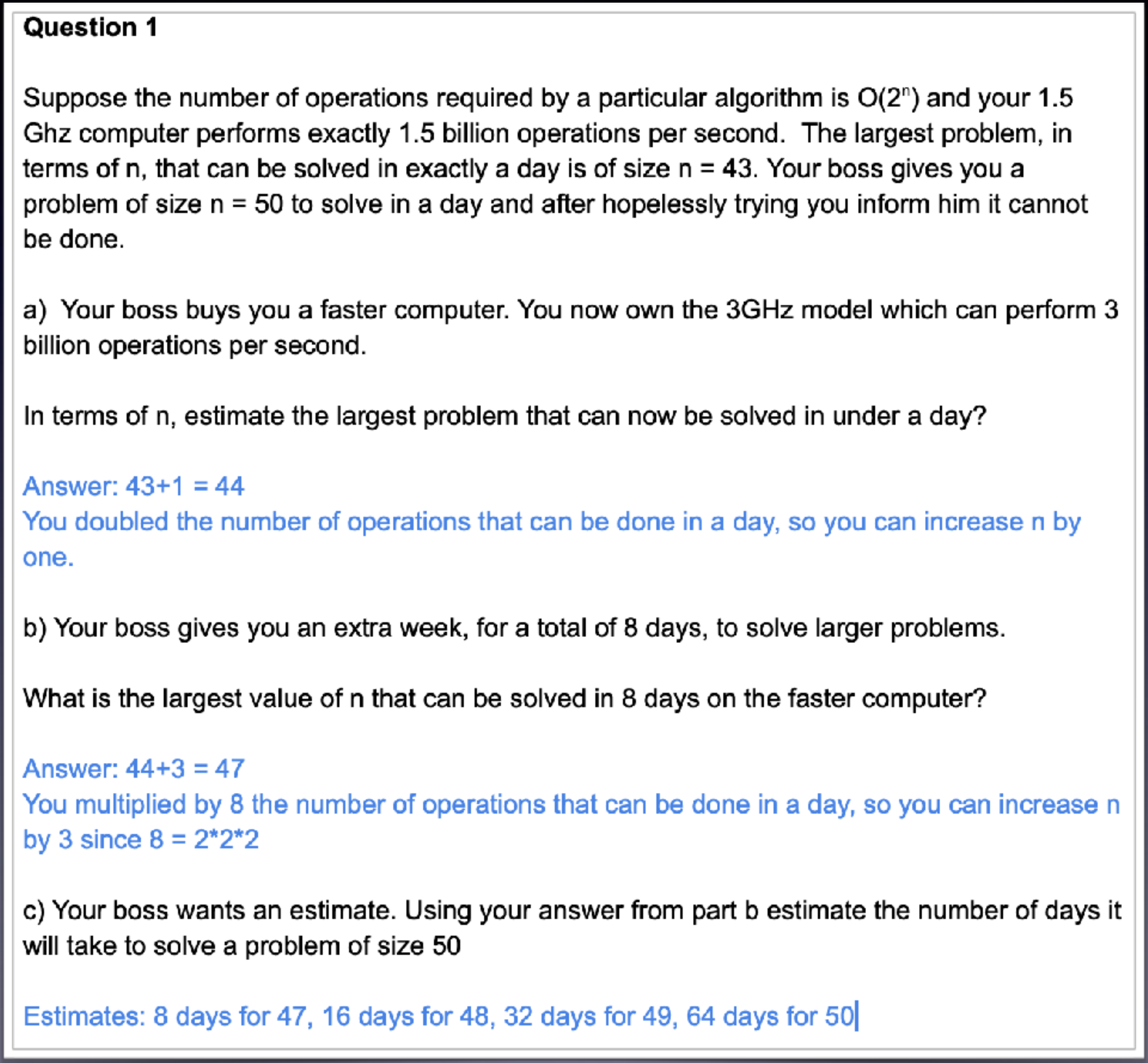
In terms of n, estimate the largest problem that can now be solved in under a day?

For estimation purposes,
if $T(n)$ is $O(2^n)$ then just assume $T(n) = c2^n$

What is $c2^{n+1}$

It's just $2T(n)$. It doesn't matter what c is.
Don't solve for c. You can do this without a calculator!

Estimating problem size



Question 1

Suppose the number of operations required by a particular algorithm is $O(2^n)$ and your 1.5 Ghz computer performs exactly 1.5 billion operations per second. The largest problem, in terms of n, that can be solved in exactly a day is of size n = 43. Your boss gives you a problem of size n = 50 to solve in a day and after hopelessly trying you inform him it cannot be done.

a) Your boss buys you a faster computer. You now own the 3GHz model which can perform 3 billion operations per second.

In terms of n, estimate the largest problem that can now be solved in under a day?

Answer: 43+1 = 44
You doubled the number of operations that can be done in a day, so you can increase n by one.

b) Your boss gives you an extra week, for a total of 8 days, to solve larger problems.

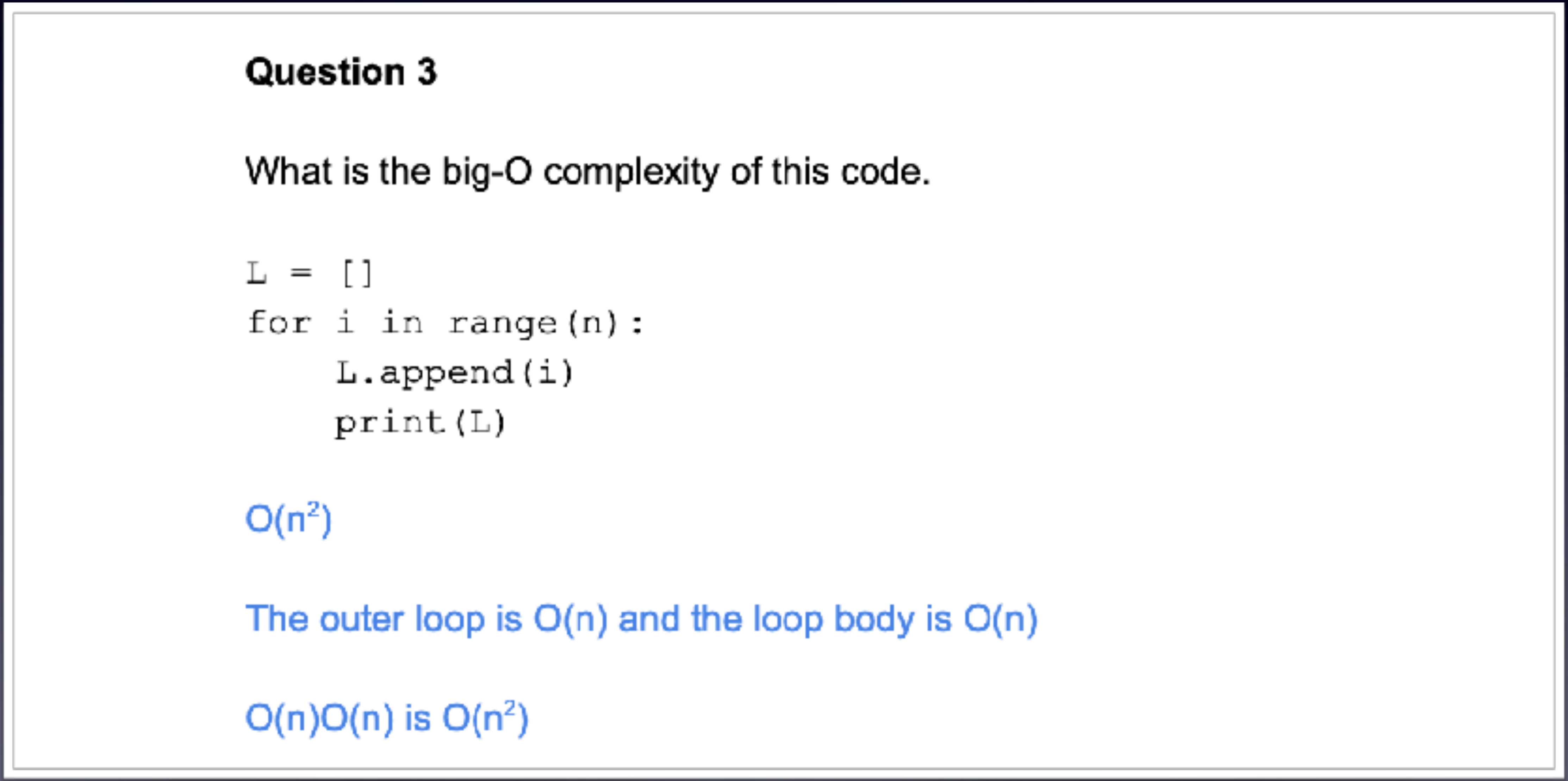
What is the largest value of n that can be solved in 8 days on the faster computer?

Answer: 44+3 = 47
You multiplied by 8 the number of operations that can be done in a day, so you can increase n by 3 since 8 = 2*2*2

c) Your boss wants an estimate. Using your answer from part b estimate the number of days it will take to solve a problem of size 50

Estimates: 8 days for 47, 16 days for 48, 32 days for 49, 64 days for 50

Estimating Complexity of Code



Question 3

What is the big-O complexity of this code.

```
L = []
for i in range(n):
    L.append(i)
    print(L)
```

O(n^2)

The outer loop is O(n) and the loop body is O(n)

O(n)O(n) is O(n^2)

Linear Data Structures

- Stack
    - What stack operations do
        - push pop peek
- Algorithms that use a stack
        - Postfix evaluation, how to convert infix to postfix
        - Balanced parentheses

Linear Data Structures

- Queue
  - What queue operations do
    - enqueue dequeue
- Deque
  - Deque operations
    - addFront, addRear, removeFront, removeRear
  - Palindrome algorithm

Infix to Postfix

```
(1+2)*(3+4)
```

Fully Parenthesize

```
((1+2)*(3+4))
```

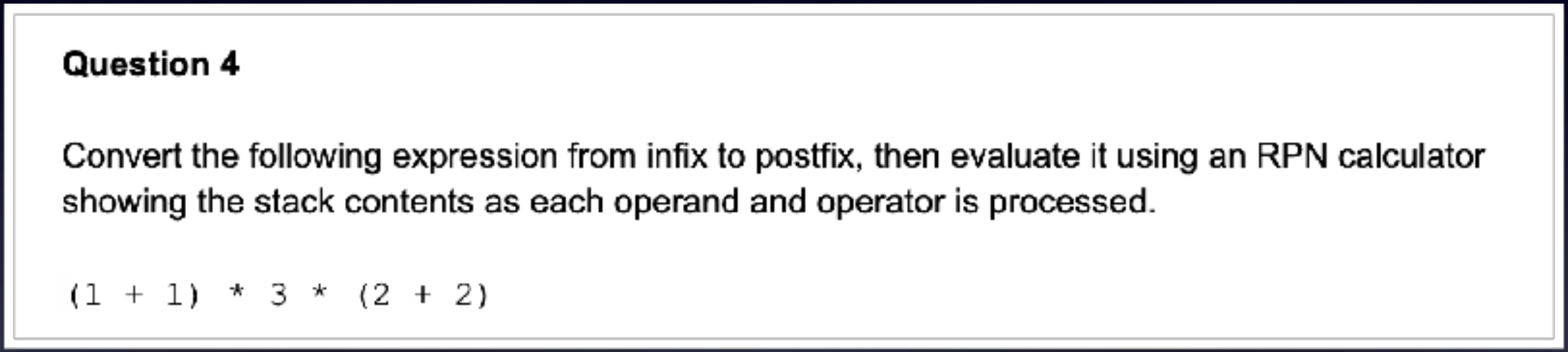
Move over operators

```
((1 2 +) (3 4 +) *)
```

Remove parenthesis

```
1 2 + 3 4 + *
```

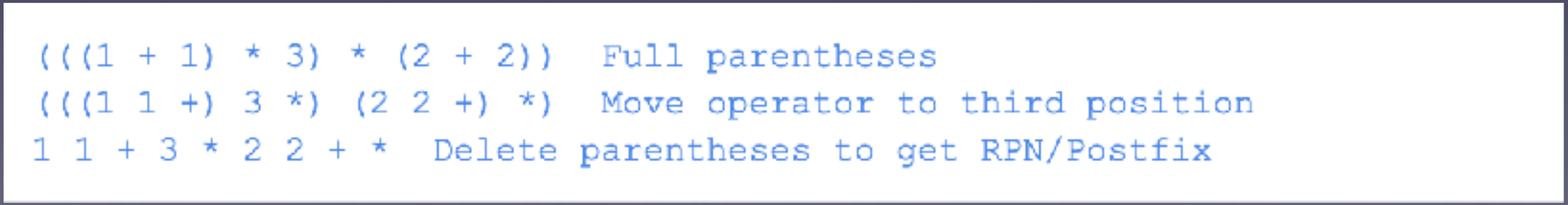
Infix to Postfix a.k.a RPN



Question 4

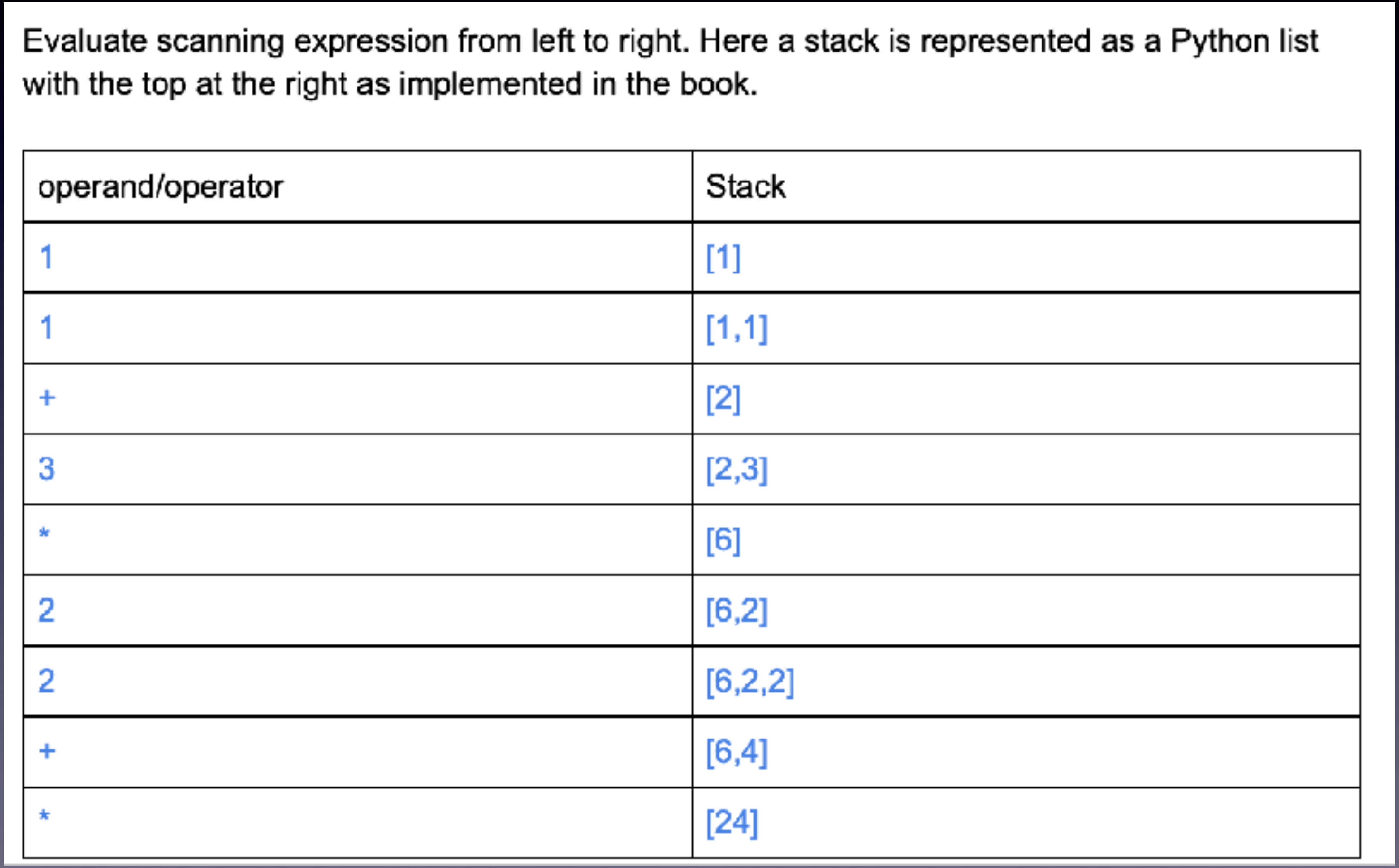
Convert the following expression from infix to postfix, then evaluate it using an RPN calculator showing the stack contents as each operand and operator is processed.

```
(1 + 1) * 3 * (2 + 2)
```



```
(((1 + 1) * 3) * (2 + 2))   Full parentheses
(((1 1 +) 3 *) (2 2 +) *)   Move operator to third position
1 1 + 3 * 2 2 + *  Delete parentheses to get RPN/Postfix
```

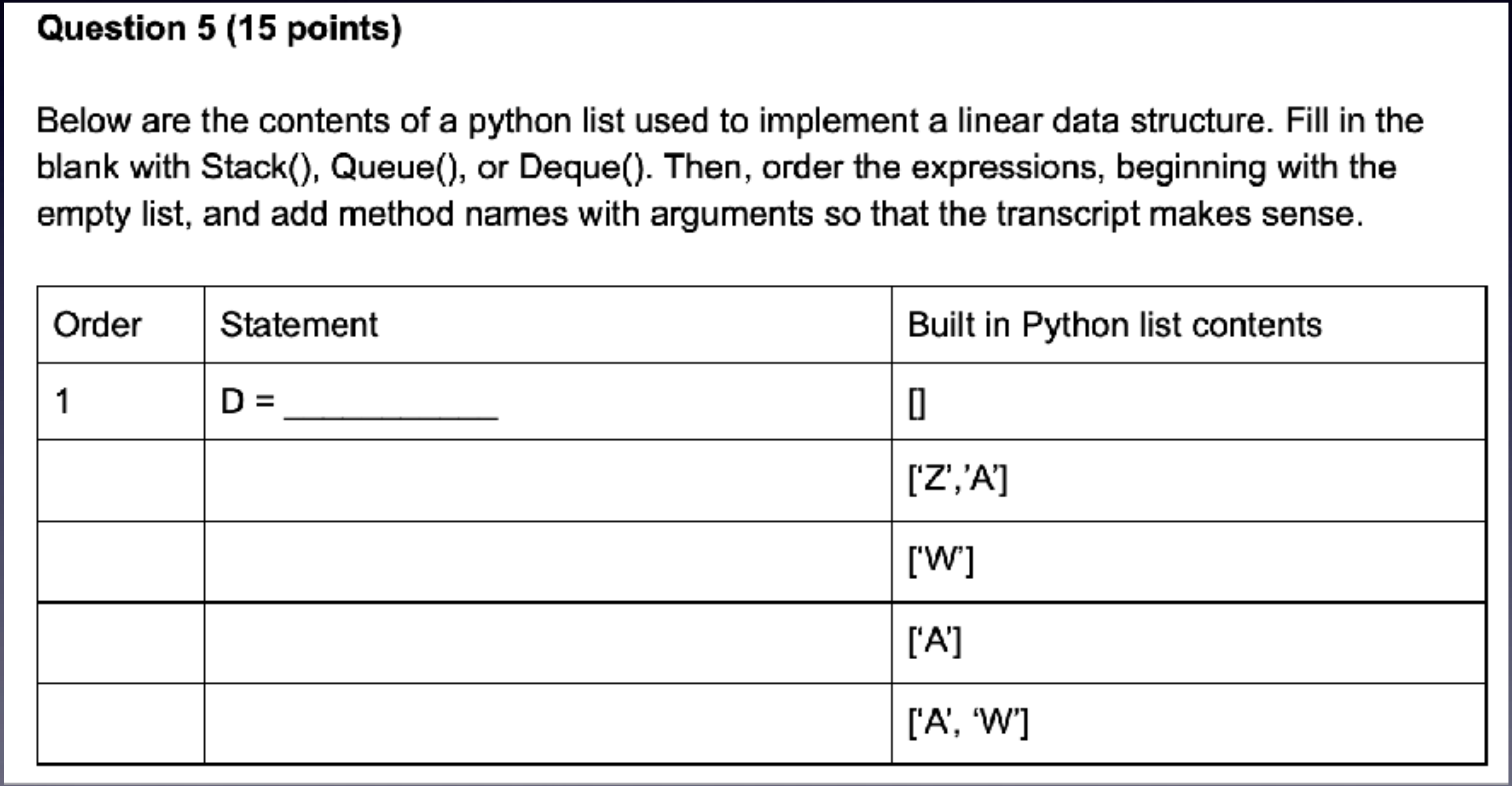
Infix to Postfix a.k.a RPN



Evaluate scanning expression from left to right. Here a stack is represented as a Python list with the top at the right as implemented in the book.

operand/operator	Stack
1	[1]
1	[1,1]
+	[2]
3	[2,3]
*	[6]
2	[6,2]
2	[6,2,2]
+	[6,4]
*	[24]

Ordering Operations

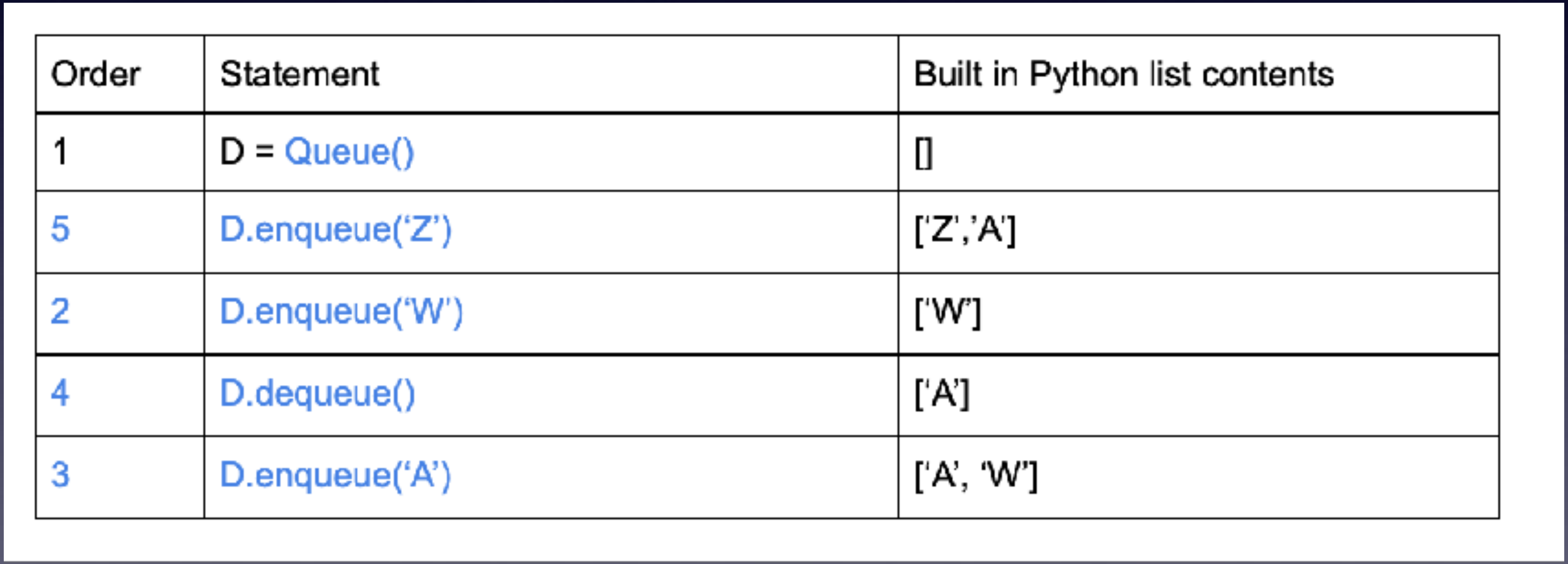


Question 5 (15 points)

Below are the contents of a python list used to implement a linear data structure. Fill in the blank with Stack(), Queue(), or Deque(). Then, order the expressions, beginning with the empty list, and add method names with arguments so that the transcript makes sense.

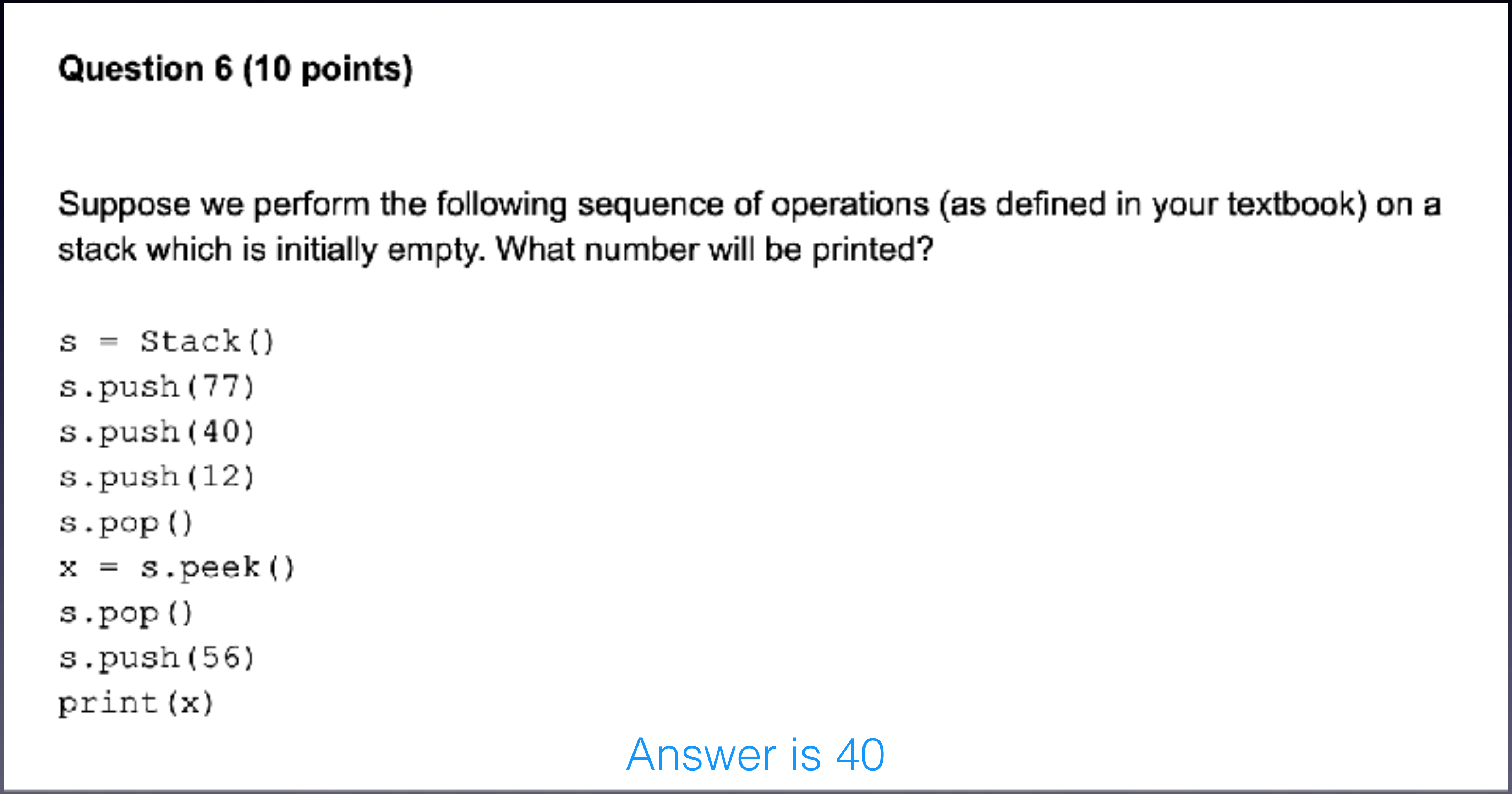
Order	Statement	Built in Python list contents
1	D = ___________	[]
		['Z','A']
		['W']
		['A']
		['A', 'W']

Ordering Operations



Order	Statement	Built in Python list contents
1	D = Queue()	[]
5	D.enqueue(‘Z’)	[‘Z’,’A’]
2	D.enqueue(‘W’)	[‘W’]
4	D.dequeue()	[‘A’]
3	D.enqueue(‘A’)	[‘A’, ‘W’]

Know what the operations do.



Question 6 (10 points)

Suppose we perform the following sequence of operations (as defined in your textbook) on a stack which is initially empty. What number will be printed?

```
s = Stack()
s.push(77)
s.push(40)
s.push(12)
s.pop()
x = s.peek()
s.pop()
s.push(56)
print(x)
```

Answer is 40

Simple Balanced Parentheses

Returns True or False if an expression consisting of only parentheses is balanced.

```
from Stack import Stack


def balanced(inStr):
    s = Stack()
        for symbol in inStr:
            if symbol == "(":
                s.push(symbol)
            else:
                if s.isEmpty():
                    return False
                else:
                    return s.pop()

    if s.isEmpty():
        return True
    else:
        return False
```

Simple Balanced Parentheses

Example expression = "()(("

Scanned	Stack contents as Python list
(	
)	
(	
(	

Simple Balanced Parentheses

Example expression = "()(("

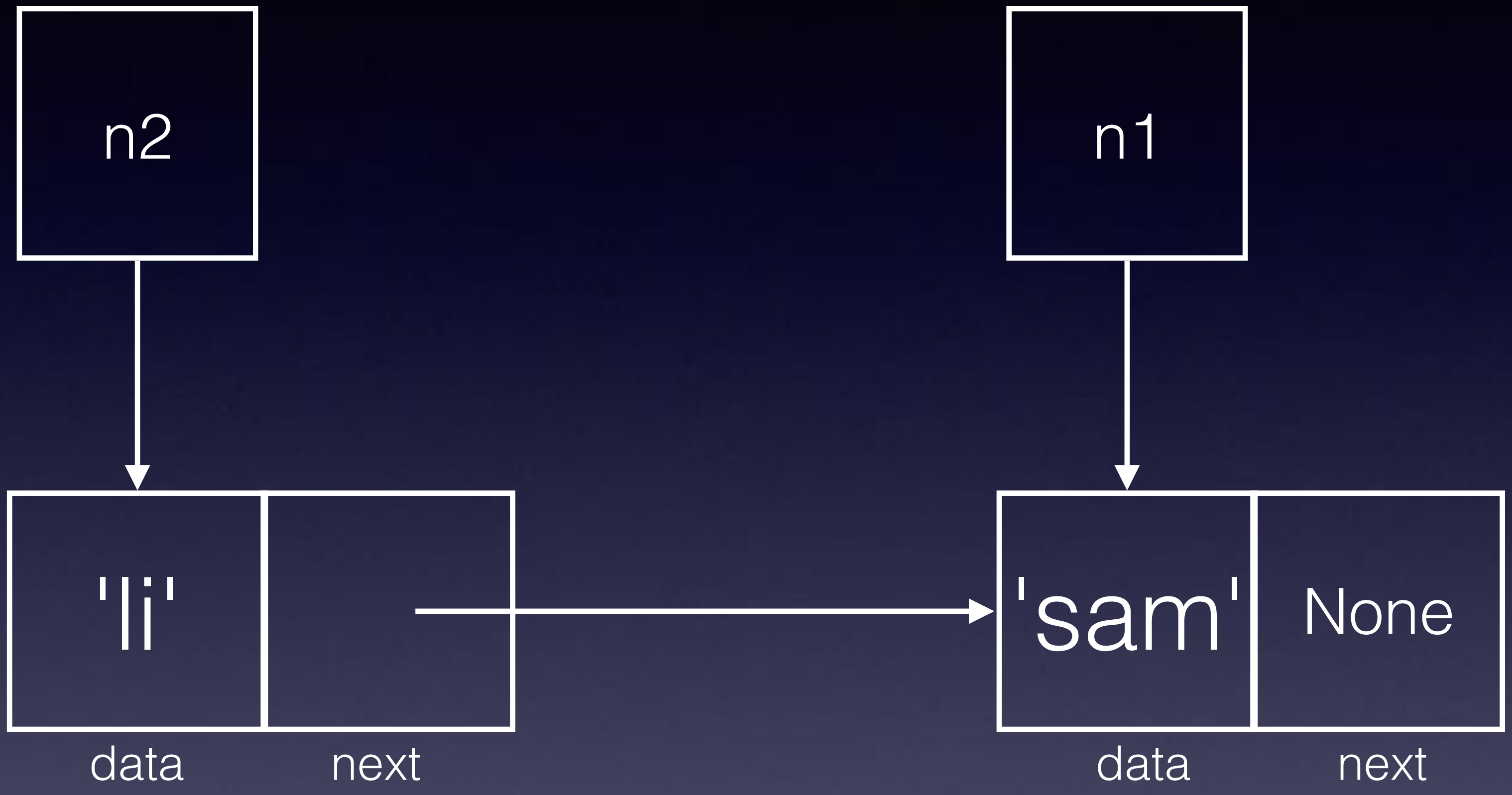
Scanned	Stack contents as Python list
(	['(']
)	[]
(	['(']
(	['(','(']

Return value? False

Linear Data Structures

- Linked List
    - How to use a Node object to implement the unordered list operations covered in lecture
        - add, append, size, delete, isEmpty
    - Traversing a linked list
    - Recursive algorithms on linked lists

Using the node ADT



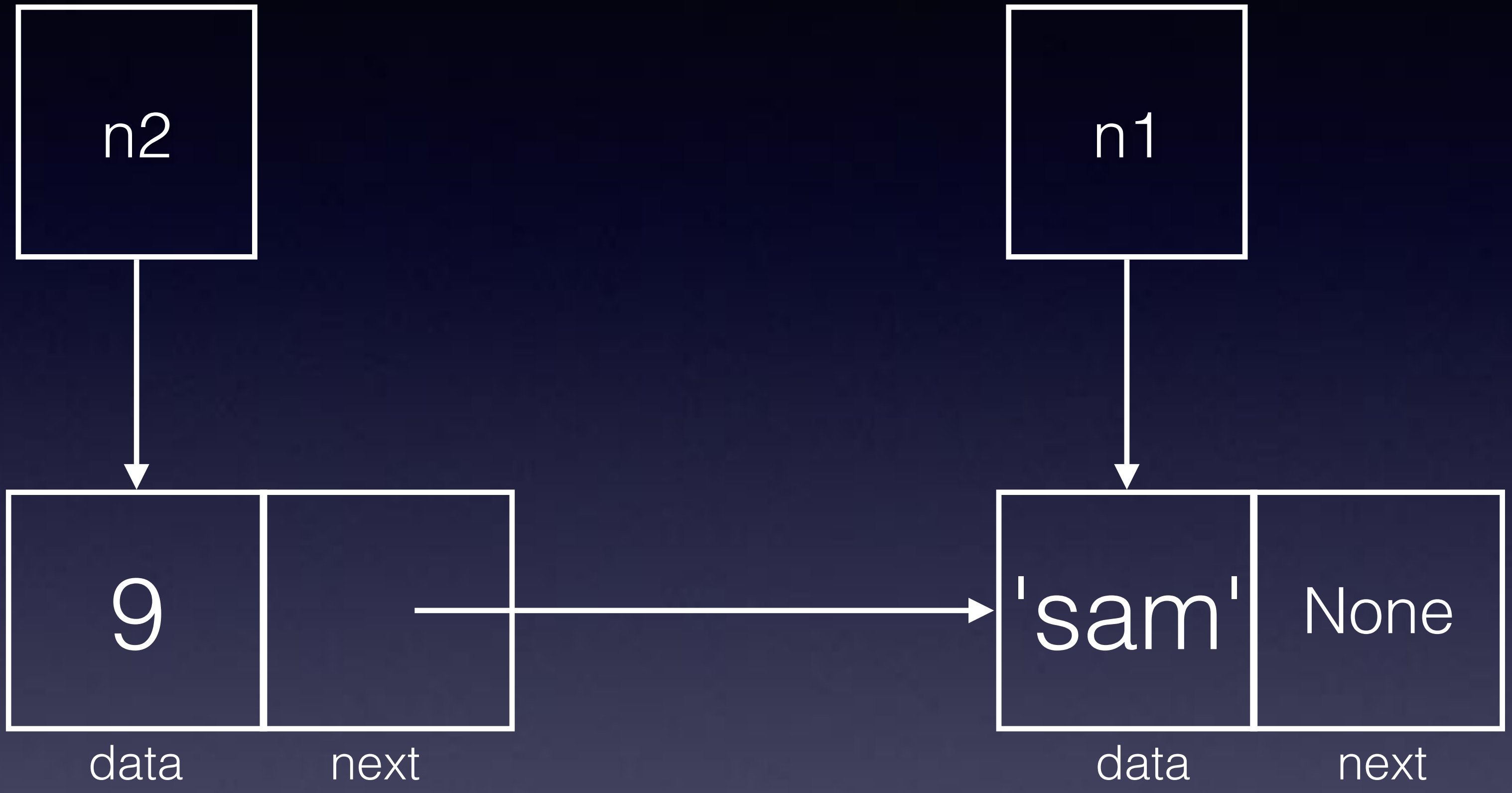
```
n2.setNext(n1)
```

we’ve created a linked list with n2 as the head

```
n1.getNext()
```

returns None because we are at the last node

Using the node ADT



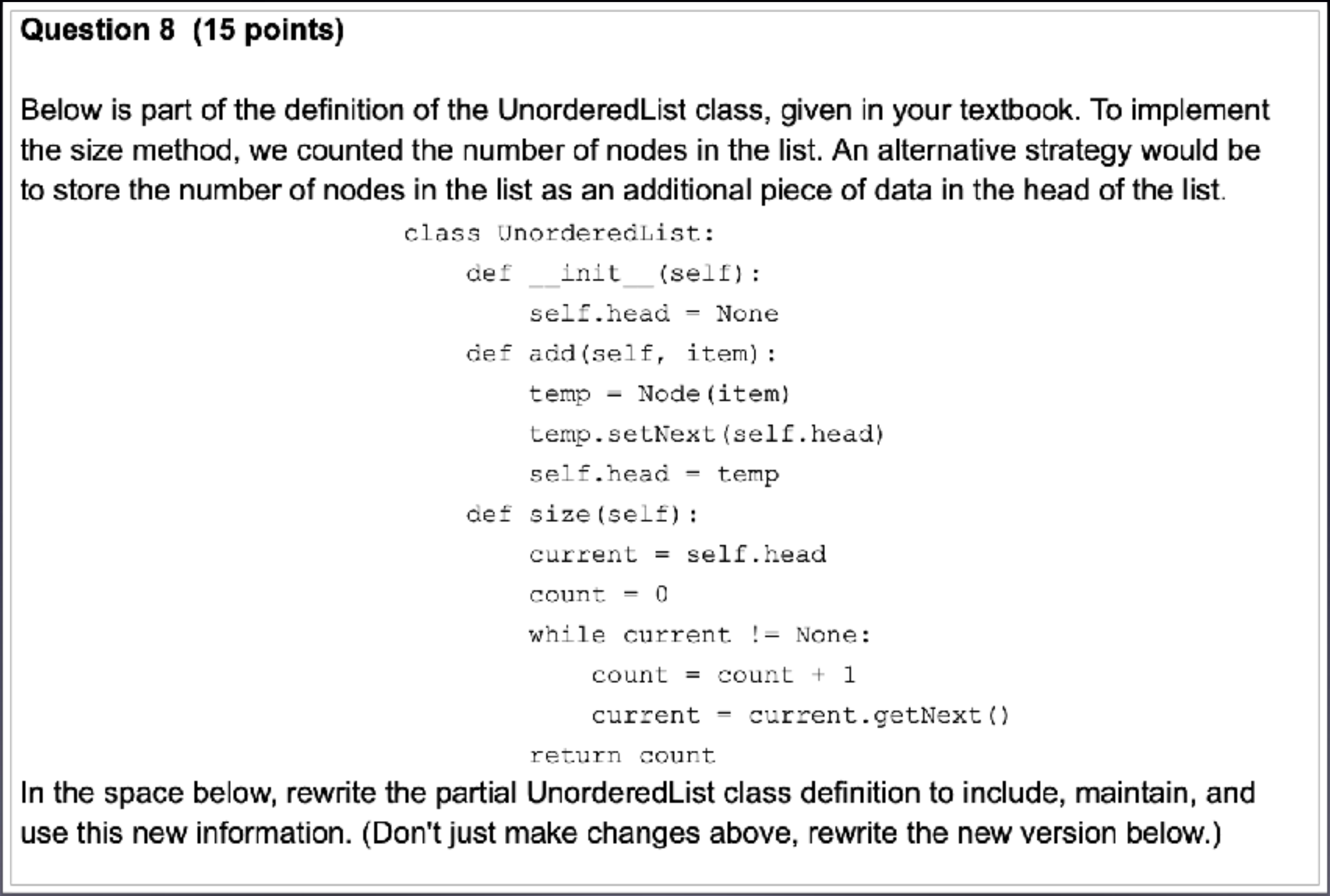
```
n2.setData(9)
```

Changes 'li' to 9

```
n1.getData()
```

returns 'sam'

Linked List



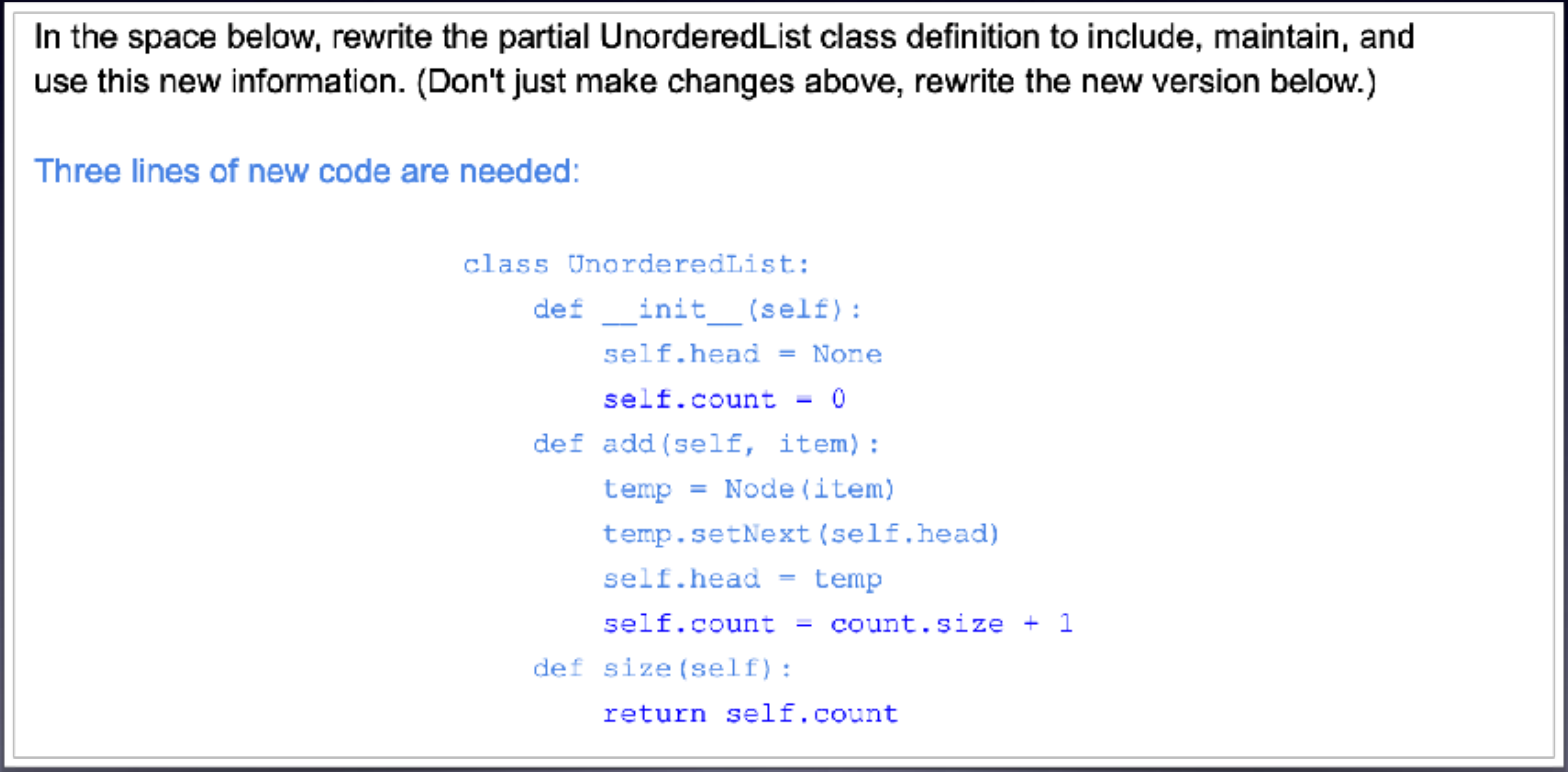
Question 8 (15 points)

Below is part of the definition of the UnorderedList class, given in your textbook. To implement the size method, we counted the number of nodes in the list. An alternative strategy would be to store the number of nodes in the list as an additional piece of data in the head of the list.

```
class UnorderedList:
    def __init__(self):
        self.head = None
    def add(self, item):
        temp = Node(item)
        temp.setNext(self.head)
        self.head = temp
    def size(self):
        current = self.head
        count = 0
        while current != None:
            count = count + 1
            current = current.getNext()
        return count
```

In the space below, rewrite the partial UnorderedList class definition to include, maintain, and use this new information. (Don't just make changes above, rewrite the new version below.)

Linked List



In the space below, rewrite the partial UnorderedList class definition to include, maintain, and use this new information. (Don't just make changes above, rewrite the new version below.)

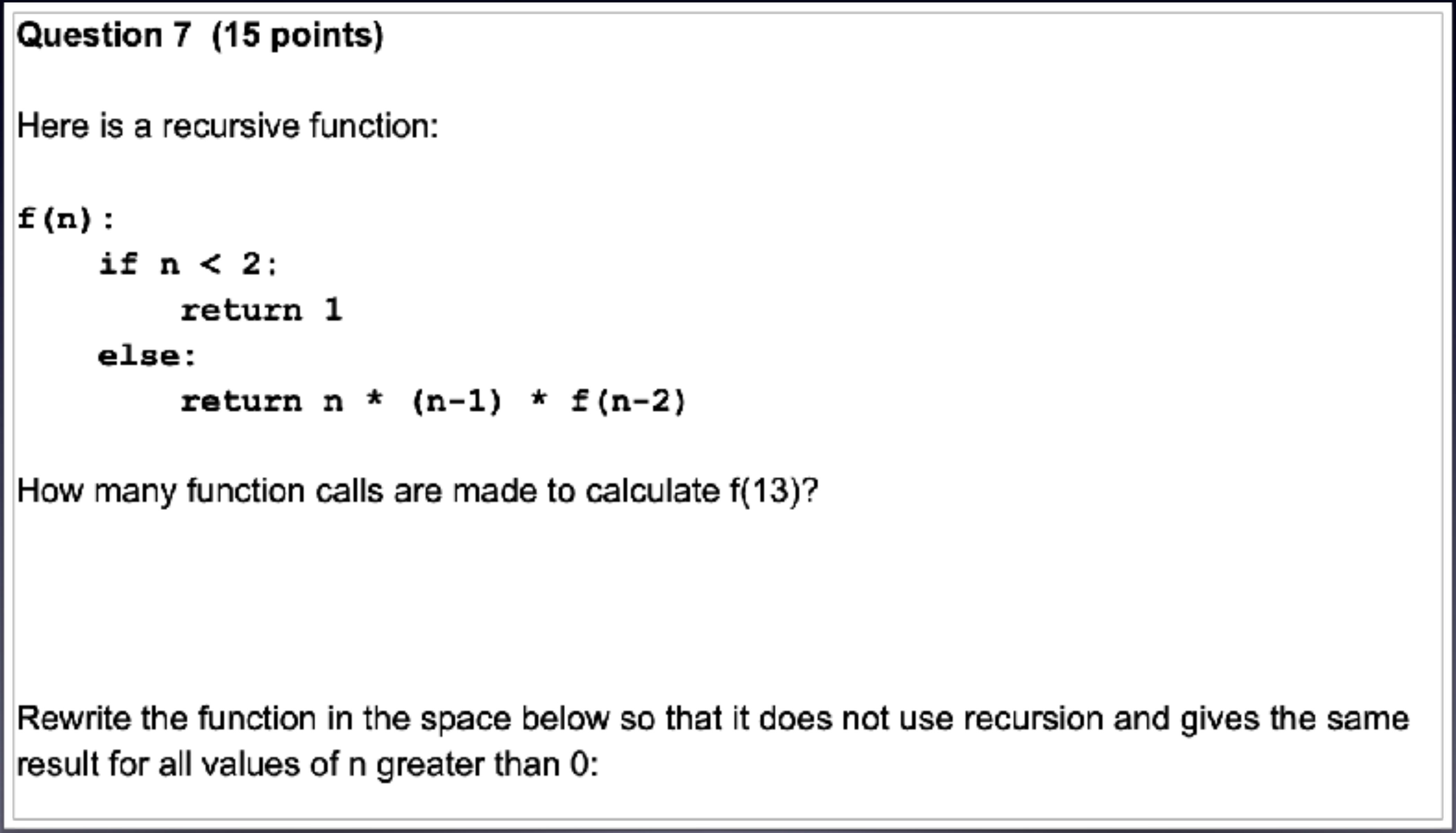
Three lines of new code are needed:

```
class UnorderedList:
    def __init__(self):
        self.head = None
        self.count = 0
    def add(self, item):
        temp = Node(item)
        temp.setNext(self.head)
        self.head = temp
        self.count = count.size + 1
    def size(self):
        return self.count
```

Recursion

- Three laws of recursion, how they inform the design of recursive algorithms.
- Recursive algorithms
    - Factorial, Fibonacci, Change, Ladder,
    - Linked list and Python list Algorithms
- Implementing new recursive solutions to problems
- Visualizing recursion:
    - Recursion trees, substitution method
- Memoizing recursive code with a dictionary

Recursion



Question 7 (15 points)

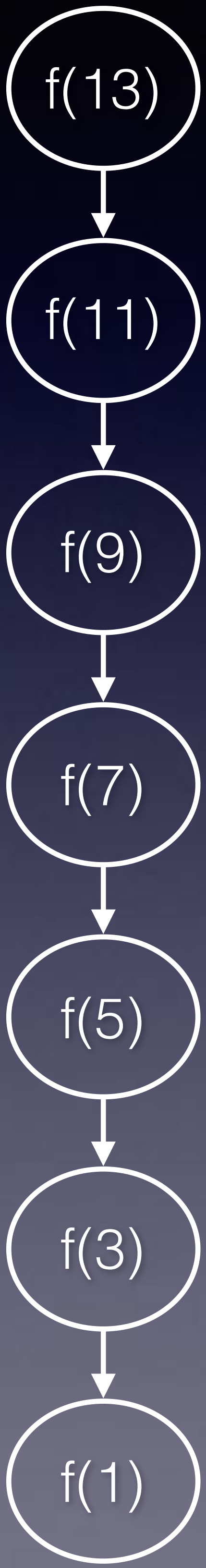
Here is a recursive function:

```
f(n):
    if n < 2:
        return 1
    else:
        return n * (n-1) * f(n-2)
```

How many function calls are made to calculate f(13)?

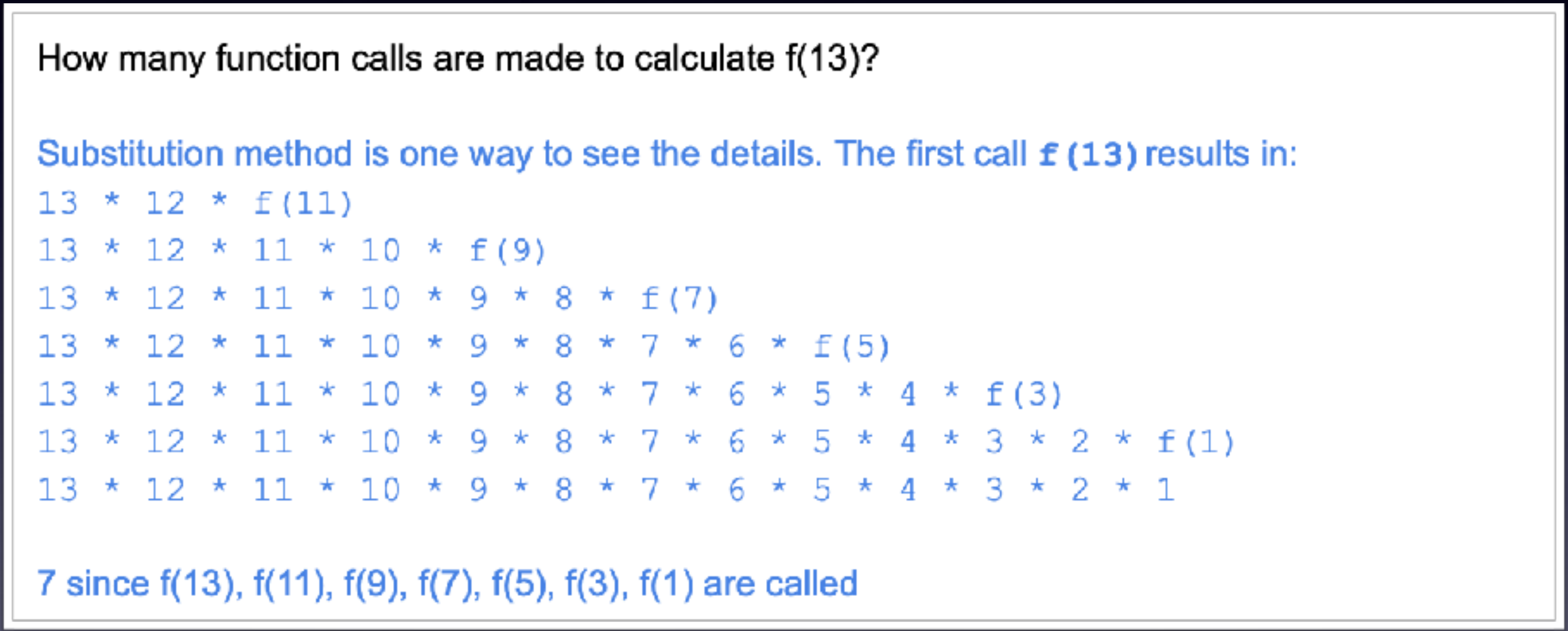
Rewrite the function in the space below so that it does not use recursion and gives the same result for all values of n greater than 0:

Recursion Tree


$$\begin{tikzcd}[row sep=small] f(13) \arrow[d] \\ f(11) \arrow[d] \\ f(9) \arrow[d] \\ f(7) \arrow[d] \\ f(5) \arrow[d] \\ f(3) \arrow[d] \\ f(1) \end{tikzcd}$$

7 CALLS

Substitution Method



How many function calls are made to calculate f(13)?

Substitution method is one way to see the details. The first call `f(13)` results in:

```
13 * 12 * f(11)
13 * 12 * 11 * 10 * f(9)
13 * 12 * 11 * 10 * 9 * 8 * f(7)
13 * 12 * 11 * 10 * 9 * 8 * 7 * 6 * f(5)
13 * 12 * 11 * 10 * 9 * 8 * 7 * 6 * 5 * 4 * f(3)
13 * 12 * 11 * 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * f(1)
13 * 12 * 11 * 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1
```

7 since f(13), f(11), f(9), f(7), f(5), f(3), f(1) are called

Recognize the function?

Substitution method shows the iterative solution

n=12 results in the following calculation

12 * 11 * 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 * 1

n=13 results in the following calculation

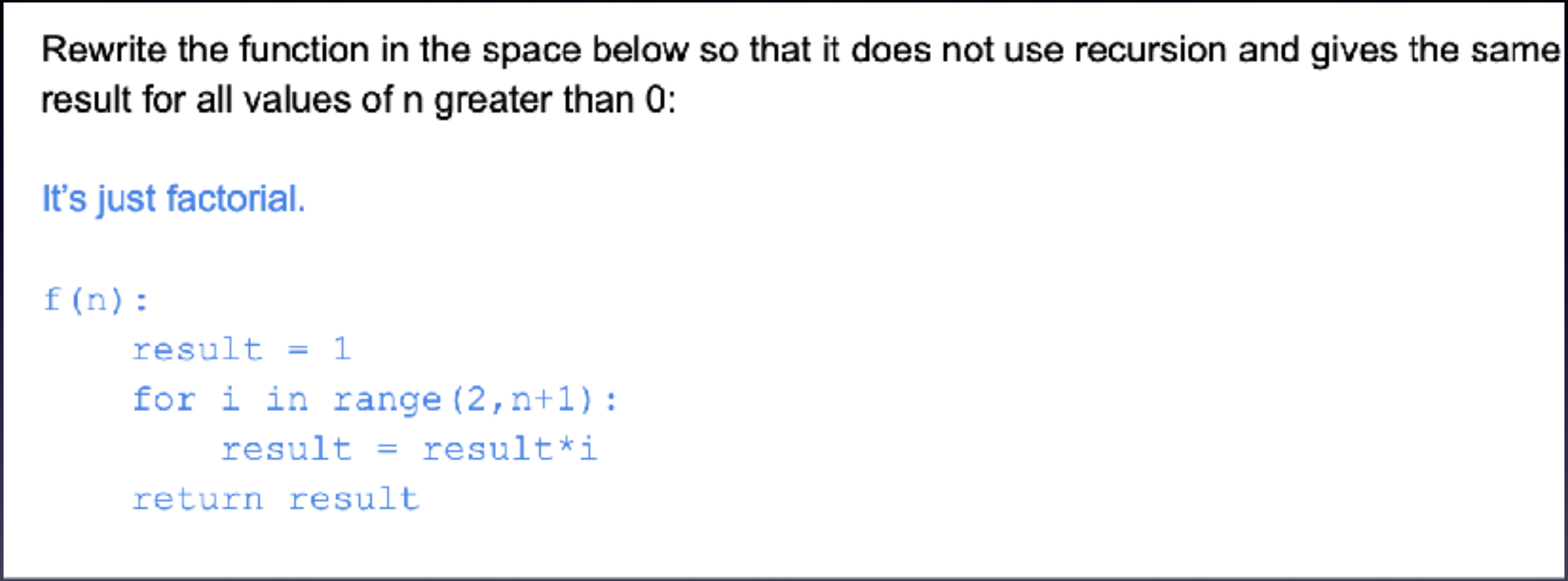
13 * 12 * 11 * 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1

n=14 results in the following calculation

14 * 13 * 12 * 11 * 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 * 1

Looks like factorial

Recognize the function?



Rewrite the function in the space below so that it does not use recursion and gives the same result for all values of n greater than 0:

It's just factorial.

```
f(n):
    result = 1
    for i in range(2,n+1):
        result = result*i
    return result
```

Best of luck!