Finger Exercises 21

Recursion and the Call Stack

≈ 2 sittings of 1 hour + stretch
Comp 102 — Programming Fundamentals

What this is. A full workout for Lecture 21, in six levels. It starts with the idea of recursion and ends with code you write. Recursion comes back in every later course, so this sheet is longer than the others.

How to do it. On **paper**, **no computer**, in **two sittings**: Levels 1–3 (Parts A–M), then Levels 4–6 (Parts N–Y). Stuck for more than a minute? Skip the item and come back. Part Z is a **stretch**. The Console Quests are for *after* the sheet, *at* a computer. The key is on the last pages: do not peek until you are done.

Predict first, then check. Write your answer *before* you type anything. A wrong guess teaches you more than an answer you only watched the computer give.

Your tool. The bottom of every page shows how to draw one frame. Draw your frames the same way in Parts H and J.

LEVEL 1 • The Idea

what recursion is, and its two parts

Part A • Do You Remember?

RECALL ≈ 4 min

A1. **Recursion**: a function _____ to do a _____ copy of the same job.

A2. Every recursive function needs two parts: the _____ case (when to stop) and the _____ case (call yourself on a smaller job).

A3. Every call gets a new _____. Each one has its own _____. The newest one is on the _____. When a call ends, its frame is _____.

A4. Python allows about _____ frames on the stack.
One more, and Python stops with _____. Too many frames is called a _____.

A5. The leap of faith: to write `factorial(n)`, _____ that `factorial(n - 1)` already works. Then check only two things: the _____, and that each call is _____.

A6. Circle **T** (true) or **F** (false): *(T / F)*

(a) A recursive function needs a loop. T / F
(b) Each call has its own `n`. T / F
(c) The first call is the first to end. T / F
(d) `countup(3)` prints `3` first. T / F
(e) A missing `return` can give `None`. T / F
(f) `n <= 0` stops more inputs than `n == 0`. T / F
(g) A `RecursionError` means a typo. T / F
(h) In Thonny, each call opens a new window. T / F

Part B • Is It Recursive?

CLASSIFY ≈ 3 min

```
1  def a(n):
2      return n + 1
3
4  def b(n):
5      if n == 0:
6          return 0
7      return b(n - 1)
8
9  def c(n):
10     for i in range(n):
11         print(i)
```

```
12 def d(n):
13     print(n)
14     d(n)
15
16 def e(s):
17     if s == "":
18         return
19     print(s[0])
20     e(s[1:])
21
22 def g(n):
23     return a(n) + a(n)
```

B1. Write **Y** or **N** in each box. Write a dash where the question makes no sense.

	a	b	c	d	e	g
It calls *itself*.						
If so: it has a base case.						
The call `x(3)` ends with no error (for `e`: `e("hi")`).						

B2. Write the line number of every recursive call: _____

B3. `b(3)` returns _____ `b(100)` returns _____ `g(3)` returns _____ `e("hi")` prints _____

Part C • Two Parts, Always

CLASSIFY ≈ 4 min

```
1  def countdown(n):
2      if n <= 0:
3          print("Blastoff!")
4          return
5      print(n)
6      countdown(n - 1)
7
8  def factorial(n):
9      if n <= 1:
10         return 1
11     return n * factorial(n - 1)
```

```
12 def length(s):
13     if s == "":
14         return 0
15     return 1 + length(s[1:])
16
17 def multiply(a, b):
18     if b == 0:
19         return 0
20     return a + multiply(a, b - 1)
21
22 def countdown_list(n):
23     if n == 0:
24         return []
25     return [n] + countdown_list(n - 1)
```

C1. Fill in the table for the five functions above.

function	base case test	returns (or does) there	smaller job: the new argument
`countdown`	_____	_____	_____
`factorial`	_____	_____	_____
`length`	_____	_____	_____
`multiply`	_____	_____	_____
`countdown_list`	_____	_____	_____

C2. Write every string that `length` gets in `length("hello")`, in order:

_____

C3. In `multiply(a, b)`, which argument gets smaller? _____ Which one never changes? _____

Part D • Trust the Smaller Call

RECALL ≈ 4 min

Do **not** trace. Take the fact on the left as true, and do **one** step. The functions are in Part C. `power` and `halvings` are in Part N.

D1.

`sum_to(9)` is 45	so `sum_to(10)` = _____ + _____ = _____
`factorial(6)` is 720	so `factorial(7)` = _____ * _____ = _____
`length("ecursion")` is 8	so `length("recursion")` = _____ + _____ = _____
`multiply(6, 7)` is 42	so `multiply(6, 8)` = _____ + _____ = _____
`power(3, 4)` is 81	so `power(3, 5)` = _____ * _____ = _____
`halvings(50)` is 5	so `halvings(100)` = _____ + _____ = _____
`countdown_list(4)` is `[4, 3, 2, 1]`	so `countdown_list(5)` = _____ + _____ = _____

D2. Now backwards. `sum_to(20)` is 210, so `sum_to(19)` is _____
`factorial(5)` is 120, so `factorial(4)` is _____

D3. `halvings(50)` is 5. So `halvings(101)` is _____, because `101 // 2` is _____.

LEVEL 2 • Following the Calls

what happens going down, and coming back up

Part E • Before or After the Call?

TRACE ≈ 5 min

```
1 def down(n):
2     if n <= 0:
3         print("Go")
4         return
5     print(n)
6     down(n - 1)
7
8 def up(n):
9     if n == 0:
10        print("Go")
11        return
12    up(n - 1)
13    print(n)
```

E1. Write what each call prints, all on one line. If it stops with an error, write the error's name.

`down(3)` _____ `up(3)` _____

`down(1)` _____ `up(1)` _____

`down(0)` _____ `up(0)` _____

`down(-2)` _____ `up(-2)` _____

E2. In `up(3)`, which frame prints first: `n` = ____. Which call ends last? _____

E3. In `up`, swap lines 12 and 13 (the `print` comes first). Now `up(3)` prints _____.

Part F • Down and Up

TRACE ≈ 4 min

```
1 def echo(n):
2     if n == 0:
3         print("base")
4         return
5     print("down", n)
6     echo(n - 1)
7     print("up", n)
```

F1. Write every line that `echo(3)` prints.

1 _____ 5 _____

2 _____ 6 _____

3 _____ 7 _____

4 _____

F2. At most, how many `echo` frames are on the stack during `echo(3)`? _____ How many lines does `echo(10)` print? _____ And `echo(0)`? _____

F3. Line 2 now reads `if n <= 0:` and line 6 reads `echo(n - 2)`. Now `echo(3)` prints

_____

Part G • Which Function Printed It?

CLASSIFY ≈ 4 min

```
def p(n):
    if n == 0:
        return
    print(n)
    p(n - 1)
```

```
def q(n):
    if n == 0:
        return
    q(n - 1)
    print(n)
```

```
def r(n):
    if n == 0:
        return
    print(n)
    r(n - 1)
    print(n)
```

```
def t(n):
    if n == 0:
        return
    t(n - 1)
    print(n)
    print(n)
```

G1. Each row is what one call `x(3)` printed. Write the function: `p`, `q`, `r`, `t`, or **none**.

`1 2 3` _____ `1 1 2 2 3 3` _____

`3 2 1` _____ `3 3 2 2 1 1` _____

`3 2 1 1 2 3` _____ `1 2 3 3 2 1` _____

G2. In `r(3)`, the first three numbers print going _____ the calls. The last three print coming back _____.

G3. `w(3)` prints `2 4 6`. It starts like `p`. Write its last two lines: _____ _____

G4. `y(3)` prints `3 2 1 0`. It is `p` with one more line. Which line, and where?

_____

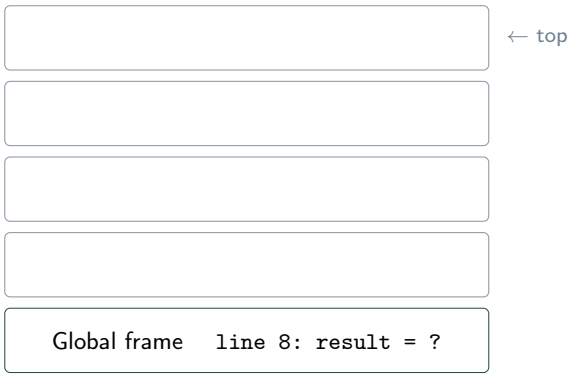
LEVEL 3 • The Call Stack

frames, lines, calls and depth

Part H • Draw the Stack

TRACE ≈ 6 min

```
1 def sum_to(n):
2     # Base case
3     if n <= 0:
4         return 0
5     # Recursive case
6     return n + sum_to(n - 1)
7
8 result = sum_to(3)
```



H1. Python has just reached line 4 for the first time. Fill in the stack at this moment, like the footer: the function, its `n`, and the line it waits on with its `?`. The running frame waits on nothing: write its line.

H2. Now coming back up. Fill in each step.

frame removed	its `?` becomes	it returns
`n: 0` returns 0, so the frame `n: 1`...	_____	_____
then the frame `n: 2`...	_____	_____
then the frame `n: 3`...	_____	_____

`result` is _____

H3. Which frame is removed first? _____ Which `sum_to` frame is removed last? _____ How many `sum_to` frames are there at most in `sum_to(100)`? _____

Part I • Line by Line

TRACE ≈ 5 min

Use the program of Part H, but change line 8 to `result = sum_to(2)`. Python runs one line at a time.

I1. Write the line Python runs at each step, and the frame it is in: **G** for the Global frame, or the frame's `n`. The first four steps are done.

step	1	2	3	4	5	6	7	8	9	10	11
line	1	8	3	6							
frame	G	G	2	2							

I2. How many times does line 3 run? ____ Line 4? ____ Which lines never run? _____

I3. The frame `n: 2` visits line 6 two times. The first time it _____. The second time it _____.

I4. After line 4 returns 0, Python goes to line ____ of the frame `n:` ____. It does **not** start that line's function again from line 1. T / F

Part J • The Stack, Moment by Moment

TRACE ≈ 6 min

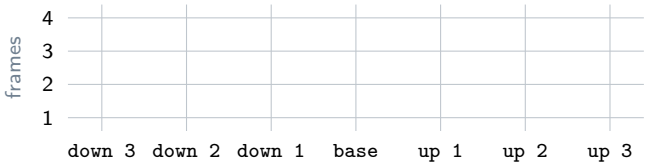
```
1 def countup(n):
2     if n == 0:
3         return
4     countup(n - 1)
5     print(n)
6
7 countup(3)
```

J1. Write the `n` of every `countup` frame on the stack at each moment, oldest first.

moment	frames: their `n`	how many
Python reaches line 3	_____	____
it prints 1	_____	____
it prints 2	_____	____
it prints 3	_____	____

J2. Now `countdown(3)` of Part C: its `print` is *before* the call. How many `countdown` frames are on the stack when it prints `3`? ____ `1`? ____ `Blastoff!`? ____

J3.



`echo(3)` of Part F prints seven lines. For each line, mark a dot: how many `echo` frames are on the stack when it prints. Join the dots.

The shape looks like a _____

J4.

```
1 def tag(n):
2     if n == 0:
3         return
4     x = n * 10
5     tag(n - 1)
6     x = x + 1
7     print(n, x)
```

`tag(3)` prints three lines: _____ _____ _____

How many different `x` variables are there during `tag(3)`? ____

The frame `n: 1` sets its `x` to 10. What is `x` in the frame `n: 2` at that moment? ____

Does the frame `n: 0` have an `x`? _____

Part K • Step by Step

TRACE ≈ 5 min

Finish each line the way the lecture did, one call per line, until the base case. Then write the answer. Use the functions of Part C, and `power` from Part N.

K1. `factorial(4)`

`= 4 * factorial(3)`

`= 4 * 3 *` _____

`=` _____

`=` _____

K2. `multiply(5, 3)`

`= 5 + multiply(5, 2)`

`= 5 + 5 +` _____

`=` _____

`=` _____

K3. `length("dog")`

`= 1 + length("og")`

`=` _____

`=` _____

`=` _____

K4. `countdown_list(3)`

`= [3] + countdown_list(2)`

`=` _____

`=` _____

`=` _____

Part L • Count the Calls

COUNT ≈ 4 min

L1. For each first call, write how many calls are made **in all** (the first call counts), and the value it returns.

	calls	*returns*		*calls*	*returns*
`sum_to(4)`	_____	_____	`multiply(3, 4)`	_____	_____
`sum_to(0)`	_____	_____	`power(2, 10)`	_____	_____
`factorial(5)`	_____	_____	`halvings(8)`	_____	_____
`factorial(1)`	_____	_____	`halvings(100)`	_____	_____
`length("hello")`	_____	_____	`countdown_list(4)`	_____	_____

(`power` and `halvings` are in Part N.)

L2. In every function of this sheet, the most frames on the stack at once (not counting the Global frame) is the same as the number of calls. Why? Each call makes only _____ call, and _____ until it returns.

Part M • How Deep?

COUNT ≈ 4 min

M1. Python allows about 1000 frames. For each call, write the most frames it needs at once. Then write **OK** or **E** (`RecursionError`).

	frames	*OK / E*		*frames*	*OK / E*
`sum_to(50)`	_____	_____	`power(2, 5000)`	_____	_____
`sum_to(3000)`	_____	_____	`power(1, 5000)`	_____	_____
`factorial(2000)`	_____	_____	`multiply(5000, 3)`	_____	_____
`halvings(1000000)`	_____	_____	`multiply(3, 5000)`	_____	_____
`length("ha" * 40)`	_____	_____	`countdown(100)`	_____	_____

M2. What decides if a call fits: how *big* the numbers are, or how *many* calls wait at once?

_____

`multiply(3, 5000)` fails. Write a call that gives the same answer and works: _____

M3.

```
  File "lec21.py", line 34, in sum_to
    return n + sum_to(n - 1)
  File "lec21.py", line 34, in sum_to
    return n + sum_to(n - 1)
  [Previous line repeated 995 more times]
RecursionError: maximum recursion depth
exceeded
```

Python printed this. Which line of the file was waiting again and again? _____

Each pair of lines stands for one _____ on the stack.

Was the answer too big a number? Y / N

LEVEL 4 • Odd Inputs and Small Changes

0, negatives, empty strings, and one line changed

Part N • Odd Inputs

TRACE ≈ 5 min

```
1 def power(b, e):
2     if e == 0:
3         return 1
4     return b * power(b, e - 1)
5
6 def halvings(n):
7     if n <= 1:
8         return 0
9     return 1 + halvings(n // 2)
```

`sum_to` is in Part H. The others are in Part C.

`halvings_bad` is `halvings` with line 7 as `if n == 1:`

Write the value. If the call stops with an error, write the error's name.

Call	Value	Call	Value
`sum_to(-5)`	_____	`power(2, -1)`	_____
`factorial(0)`	_____	`countdown_list(0)`	_____
`factorial(-3)`	_____	`countdown_list(-1)`	_____
`length("")`	_____	`halvings(1)`	_____
`multiply(7, 0)`	_____	`halvings(0)`	_____
`multiply(3, -1)`	_____	`halvings_bad(0)`	_____
`power(2, 0)`	_____	`sum_to(100)`	_____
`power(5, 1)`	_____	`sum_to(5000)`	_____

N1. Which line of the lecture's Checklist would have caught the errors on `multiply(3, -1)` and `countdown_list(-1)`?

_____

A fix for `countdown_list`, in one symbol: _____

Part O • Does Every Call Get Closer?

CLASSIFY ≈ 4 min

Each row is a function `f` with one base case test and one recursive call. For each first call, write ✓ if it reaches the base case, or ✗ if it ends in a `RecursionError`.

	base case	call	first call					
(a)	`if n == 0:`	`f(n - 1)`	`f(3)`	___	`f(0)`	___	`f(-2)`	___
(b)	`if n == 0:`	`f(n - 2)`	`f(4)`	___	`f(5)`	___	`f(0)`	___
(c)	`if n <= 0:`	`f(n - 2)`	`f(4)`	___	`f(5)`	___	`f(-1)`	___
(d)	`if n == 1:`	`f(n // 2)`	`f(8)`	___	`f(6)`	___	`f(0)`	___
(e)	`if n <= 0:`	`f(n + 1)`	`f(3)`	___	`f(0)`	___	`f(-2)`	___
(f)	`if s == "":`	`f(s[1:])`	`f("hi")`	___	`f("")`	___	`f("a b")`	___
(g)	`if n == 10:`	`f(n + 1)`	`f(7)`	___	`f(10)`	___	`f(12)`	___

Part P • Change One Thing

TRACE ≈ 6 min

```
1 def sum_to(n):
2     if n <= 0:
3         return 0
4     return n + sum_to(n - 1)
```

P1. Each row changes **one** line of this function. All the other lines stay as they are. Write what the three calls give now. Write **E** for a `RecursionError`.

P2. Which change is harmless for every whole number? _____

Which one is harmless only when `n` is 0 or more? _____

Which one returns `n` itself when `n` is 0 or more? _____

the changed line	sum_to(4)	sum_to(0)	sum_to(-3)
(no change)	10	0	0
(a) 3: `return 10`	____	____	____
(b) 3: `return n`	____	____	____
(c) 2: `if n < 0:`	____	____	____
(d) 2: `if n == 0:`	____	____	____
(e) 2: `if n <= 2:`	____	____	____
(f) 4: `return n + sum_to(n - 2)`	____	____	____
(g) 4: `return n * sum_to(n - 1)`	____	____	____
(h) 4: `return 1 + sum_to(n - 1)`	____	____	____
(i) 4: `return sum_to(n - 1)`	____	____	____
(j) 4: `return n + sum_to(n + 1)`	____	____	____
(k) 4: `return n + sum_to(n) - 1`	____	____	____

Part Q • Same Function, Different Clothes

CLASSIFY ≈ 5 min

Each `s` below is meant to be `sum_to` of Part P, written another way. Write **same** if it gives the same answer as `sum_to` for *every* whole number. If not, write one call that shows the difference, and what `s` gives for it.

(a) _____

```
def s(n):
    if n <= 0:
        return 0
    else:
        return n + s(n - 1)
```

(b) _____

```
def s(n):
    if n > 0:
        return n + s(n - 1)
    return 0
```

(c) _____

```
def s(n):
    if n <= 0:
        return 0
    rest = s(n - 1)
    return rest + n
```

(d) _____

```
def s(n):
    if n == 1:
        return 1
    return n + s(n - 1)
```

(e) _____

```
def s(n):
    total = 0
    if n > 0:
        total = n + s(n - 1)
    return total
```

(f) _____

```
def s(n):
    if n <= 0:
        return 0
    s(n - 1)
    return n
```

Q1. In (b) and (e) there is no `return 0` under an `if`. Where is the base case?

_____

Part R • The Right Starting Value

FILL IN ≈ 4 min

R1. The base case returns the value that **changes nothing**. Fill in the last column.

the job	the recursive case returns	the base case returns
add numbers up	`n + f(n - 1)`	_____
multiply numbers	`n * f(n - 1)`	_____
count letters	`1 + f(s[1:])`	_____
build a list	`[n] + f(n - 1)`	_____
build a string	`"*" + f(n - 1)`	_____

R2. `factorial` with `return 0` in its base case: `factorial(5)` gives _____

`stars(n)` returns `"*" + stars(n - 1)`, with `return 0` in its base case: `stars(0)` gives _____ and `stars(2)` gives _____

R3.

```
1 def d1(n):
2     if n == 0:
3         return ""
4     return str(n) + d1(n - 1)
5
6 def d2(n):
7     if n == 0:
8         return ""
9     return d2(n - 1) + str(n)
```

`d1(4)`: _____ `d2(4)`: _____ `d1(0)`: _____

Which one is like a `print` *before* the call? _____

In `stars`, is `"*" + stars(n - 1)` different from `stars(n - 1) + "*"`? _____ Why? _____

LEVEL 5 • Bugs *find it, name it, fix it*

Part S • Bug Hunt FIX ≈ 8 min

Each function has **one** bug. Answer the questions, then write the fix. Name the bug: **N** = not getting smaller, **J** = base case jumped over, **R** = missing `return`, or **O** = other.

S1.

```
1 def factorial(n):
2     if n <= 1:
3         return 1
4     return n * factorial(n)
```

`factorial(3)` gives _____

Bug: ___ Fix: _____

S2.

```
1 def threes(n):
2     if n == 0:
3         print("Blastoff!")
4         return
5     print(n)
6     threes(n - 3)
```

Tick each call that ends with `Blastoff!`:

☐ `threes(9)` ☐ `threes(10)` ☐ `threes(12)`

☐ `threes(0)` ☐ `threes(1)`

Bug: ___ Fix: _____

S3.

```
1 def length(s):
2     if s == "":
3         return 0
4     1 + length(s[1:])
```

`length("")`: _____ `length("a")`: _____

`length("ab")`: _____

Bug: ___ Fix: _____

S4.

```
1 def sum_to(n):
2     if n <= 0:
3         return 1
4     return n + sum_to(n - 1)
```

`sum_to(4)`: _____ `sum_to(0)`: _____

Bug: ___ Fix: _____

S5.

```
1 def countdown_list(n):
2     if n == 0:
3         return []
4     return n + countdown_list(n - 1)
```

`countdown_list(0)`: _____

`countdown_list(2)`: _____

Bug: ___ Fix: _____

S6.

```
1 def countdown(n):
2     print(n)
3     countdown(n - 1)
4     if n <= 0:
5         print("Blastoff!")
6         return
```

`countdown(2)` prints _____

and then _____

Fix: _____

S7.

```
1 def multiply(a, b):
2     if b == 1:
3         return a
4     return a + multiply(a, b - 1)
```

`multiply(3, 4)`: _____ `multiply(3, 1)`: _____

`multiply(3, 0)`: _____

Fix: _____

Part T • More Bugs: Return, Print and Other Slips

FIX ≈ 8 min

One bug each again. These are the slips beginners make most often.

T1.

```
1 def total(n):
2     if n <= 0:
3         return 0
4     print(n + total(n - 1))
```

`x = total(1)` prints _____ and `x` is _____
`total(2)` prints _____, then _____
Fix: _____

T2.

```
1 def countdown(n):
2     if n <= 0:
3         print("Blastoff!")
4     print(n)
5     countdown(n - 1)
```

`countdown(1)` prints its first five lines:
_____
and it ends with _____ Fix: _____

T3.

```
1 def spell(s):
2     print(s[0])
3     if s == "":
4         return
5     spell(s[1:])
```

`spell("hi")` prints _____, then stops with _____ (not a `RecursionError`!)
Fix: _____

T4.

```
1 def length(s):
2     return 1 + length(s[1:])
```

`""[1:]` is _____. So `length("ab")` gives _____ Fix: _____

T5.

```
1 def multiply(a, b):
2     if b == 0:
3         return 0
4     return a + multiply(a - 1, b)
```

`multiply(3, 0)`: _____ `multiply(3, 2)`: _____
Something gets smaller. Why does it not stop?
_____

T6.

```
1 def factorial(n):
2     if n <= 1:
3         return
4     return n * factorial(n - 1)
```

`factorial(1)`: _____ `factorial(3)`: _____
Fix: _____

T7.

```
1 def fact(n):
2     if n <= 1:
3         return 1
4     return n * factorial(n - 1)
```

The file holds only this function. It was renamed.
`fact(1)`: _____ `fact(3)`: _____
Fix: _____

T8.

```
1 def countup(n):
2     if n == 0:
3         return
4     return countup(n - 1)
5     print(n)
```

`countup(3)` prints _____
Which line can never run? _____
Fix: _____

LEVEL 6 • Read It, Write It

from reading a function to writing one

Part U • Finish the Function

FILL IN ≈ 5 min

Fill in every blank. Each answer is short.

U1.

```
1 def power(b, e):
2     if e == ____:
3         return ____
4     return ____ * power(b, _____)
```

U2.

```
1 def count_up_list(n):
2     if n == 0:
3         return ______
4     return count_up_list(______) + ______
```

`count_up_list(3)` is `[1, 2, 3]`.

U3.

```
def repeat(s, n):
    if n ____ 0:
        return ______
    return ___ + repeat(s, _____)
```

`repeat("ab", 3)` is `"ababab"`. Use only `+`.

U4.

```
def stairs(n):
    if n == 0:
        return
    ----------------------
    ----------------------
```

`stairs(3)` prints `*`, then `**`, then `***`, one on each line.

Part V • Same Job, Two Ways

CLASSIFY ≈ 3 min

V1. Each loop does the same job as one function of the lecture. Write its name. Then write what the loop's `while` test does in the recursive version.

(a) function: _____

```
def f(n):
    t = 0
    while n > 0:
        t = t + n
        n = n - 1
    return t
```

(b) function: _____

```
def g(n):
    while n > 0:
        print(n)
        n = n - 1
    print("Blastoff!")
```

(c) function: _____

```
def h(a, b):
    t = 0
    while b > 0:
        t = t + a
        b = b - 1
    return t
```

(d) function: _____

```
def k(s):
    t = 0
    while s != "":
        t = t + 1
        s = s[1:]
    return t
```

V2. What does `f(-5)` return? _____ What does `h(3, -1)` return? _____ Which one gives a different answer from the recursive version of Part N? _____

Part W • Read the Function

TRACE ≈ 7 min

Write what each call returns. Then say in a few words what the function does.

W1.

```
def f(n):
    if n == 0:
        return 0
    return n * 2 + f(n - 1)
```

`f(3)`: _____ `f(0)`: _____ `f(5)`: _____

It works out: _____

W2.

```
def g(n):
    if n == 0:
        return 1
    return 2 * g(n - 1)
```

`g(3)`: _____ `g(0)`: _____ `g(10)`: _____

It works out: _____

W3.

```
def h(a, b):
    if b == 0:
        return a
    return h(a + 1, b - 1)
```

`h(3, 4)`: _____ `h(9, 0)`: _____

It works out: _____

`a` gets bigger. Can it still stop? _____

W4.

```
1 def p(n):
2     if n <= 1:
3         return n
4     return p(n - 2)
```

`p(7)`: _____ `p(10)`: _____ `p(1)`: _____

For `n` ≥ 0 it works out: _____

W5.

```
1 def e(n):
2     if n == 0:
3         return 0
4     if n % 2 == 0:
5         return 1 + e(n - 1)
6     return e(n - 1)
```

`e(6)`: _____ `e(5)`: _____ `e(1)`: _____

It counts: _____

How many calls does `e(6)` make in all? _____

W6.

```
1 def w(a, b):
2     if a > b:
3         return 0
4     return a + w(a + 1, b)
```

`w(3, 6)`: _____ `w(5, 5)`: _____ `w(6, 3)`: _____

It works out: _____

`a` goes *up*. What does it get closer to? _____

Part X • Calls Inside Calls

TRACE ≈ 4 min

X1. Write each value. The functions are in Parts C and N.

`sum_to(2) + sum_to(3)`	_____	`length("ab" * 3)`	_____
`sum_to(sum_to(2))`	_____	`length("abc"[1:])`	_____
`factorial(factorial(3))`	_____	`power(2, length("hey"))`	_____
`multiply(sum_to(3), 2)`	_____	`countdown_list(length("hi"))`	_____
`sum_to(4) - sum_to(3)`	_____	`factorial(5) // factorial(4)`	_____

X2. In `sum_to(sum_to(2))`, which call runs first: the inner or the outer? _____ Are frames of both ever on the stack at the same time? _____ The most `sum_to` frames at once: ____

Part Y • One Sentence Each

EXPLAIN ≈ 3 min

Y1. Why does `countup(3)` print `1` first, not `3`?

_____

Y2. Why is `n <= 0` a safer base case than `n == 0`?

_____

Y3. `sum_to` without the `return` on line 6 gives `None` for 1 but crashes for 2. Why?

_____

Y4. Why is the website called Stack Overflow?

_____

Part Z • Write the Code

STRETCH ≈ 15 min, optional

Use recursion, not a loop. Check the Checklist: base case, smaller job, odd inputs stop, every path returns.

Z1. Write `sum_squares(n)`: $1^2 + 2^2 + \ldots + n^2$. `sum_squares(3)` is 14, and `sum_squares(0)` is 0.

Z2. Write `countdown_by(n, step)`: it prints `n`, `n - step`, … while the number is above 0, then

`Blastoff!`. `countdown_by(10, 3)` prints `10 7 4 1 Blastoff!` (one on each line). It must stop for every `n`, and every `step` of 1 or more.

Z3. Write `zeros(n)`: a list of `n` zeros. `zeros(3)` is `[0, 0, 0]`; `zeros(0)` is `[]`.

Z4. Write `print_range(a, b)`: it prints `a`, `a + 1`, . . . , `b`, one on each line. `print_range(3, 6)` prints `3 4 5 6`, and `print_range(6, 3)` prints nothing.

Z5. Write `count_down_up(n)`: `count_down_up(3)` prints `3 2 1 0 1 2 3`, one on each line (the `0` only once).

Console Quests

after the sheet · 5 min · at a computer

When unsure, just try it in the console. These are deliberately **not** answered anywhere on this sheet — the console is the answer key. Predict first, then type. Start from `lec21.py`.

1. Debug `sum_to(3)` in Thonny (Ctrl+F5) and press **Step into** (F7) again and again. How many `sum_to` windows are open at the deepest moment? Does that match Part H?
2. Type `import sys` and `sys.getrecursionlimit()`. Then try `sum_to(900)` and `sum_to(1000)`. Which one stops with a `RecursionError`?
3. Remove the `#` in front of `infinite(5)` and run it. What is the *last* number it prints before the error? So how many `infinite` frames were there? Why not exactly 1000?

Stuck on more than two items? Re-read the Lecture 21 slides "Every Call Gets a Frame", "Tracing `sum_to(4)`: Going Down" and "Coming Back Up", "The Big Three Recursion Bugs" and "Checklist" before the next class — then redo Parts E, H, I and S cold.

Answer Key

Check your work *after* attempting everything. If an answer surprises you, that item is worth re-reading in the slides. Console Quests have no key — the console is the key.

A1. **calls itself**; **smaller**.

A2. **base**; **recursive**.

A3. **frame**; `n` (its own variables); **top**; **removed**. Last on, first off.

A4. **1000**; **RecursionError**; **stack overflow** (the website's name).

A5. **trust**; **base case**; **smaller**.

A6. (a) **False** — the calls do the repeating. (b) **True**. (c) **False** — it is the *last* to end: it waits for all the others. (d) **False** — it prints `1` first: the top frame prints first. (e) **True** (or a `TypeError` one call later). (f) **True** — it also stops negatives, and odd numbers that jump over 0. (g) **False** — the code runs; it just never reaches the base case. (h) **True** — with **Step into**; each window is a frame.

B1. Calls itself: `a` N, `b` **Y**, `c` N (a loop repeats, but nothing calls itself), `d` **Y**, `e` **Y**, `g` N (it calls `a`, another function). Base case: `b` Y (`n == 0`), `d` **N**, `e` Y (`s == ""`); a dash for the others. Ends: all Y but `d`: it prints `3` about 1000 times, then a `RecursionError`.

B2. Lines **7**, **14** and **20**. (Line 23 calls `a` from inside `g`: not recursion.)

B3. `0`; `0` (every call only hands on what the base case gave); `8`; `h` then `i`.

C1. `countdown`: `n <= 0`; prints `Blastoff!`; `n - 1`. `factorial`: `n <= 1`; `1`; `n - 1`. `length`: `s == ""`; `0`; `s[1:]` (one letter shorter). `multiply`: `b == 0`; `0`; `b - 1` (`a` stays the same). `countdown_list`: `n == 0`; `[]`; `n - 1`.

C2. `"hello"`, `"ello"`, `"llo"`, `"lo"`, `"o"`, `""`: six calls.

C3. `b`; `a`. Only one thing has to get smaller to reach the base case.

D1. $10 + 45 = 55$. $7 * 720 = 5040$. $1 + 8 = 9$. $6 + 42 = 48$. $3 * 81 = 243$. $1 + 5 = 6$. `[5] + [4, 3, 2, 1] = [5, 4, 3, 2, 1]`. This is the leap of faith: trust the smaller call, do one step.

D2. `190` ($210 - 20$); `24` ($120/5$).

D3. `6`; `50`. The smaller job is whatever the call's argument works out to.

E1. `down`: `3 2 1 Go`; `1 Go`; `Go`; `Go`. `up`: `Go 1 2 3`; `Go 1`; `Go`; **RecursionError**, and it prints *nothing*: from -2, `n` goes $-3, -4, \ldots$ and never meets 0, and every `print` waits for a call that never ends.

E2. `n` = **1** prints first (the top frame, just above the base case). `up(3)`, the first call, ends last.

E3. `3 2 1 Go`: a `print` before the call runs going down.

F1. `down 3`, `down 2`, `down 1`, `base`, `up 1`, `up 2`, `up 3`. The downs print going down; the ups print coming back up.

F2. **4** (`n` = 3, 2, 1, 0). **21** (10 downs, `base`, 10 ups). **1**: just `base`.

F3. `down 3`, `down 1`, `base`, `up 1`, `up 3`. From 1, `n` goes to -1; `<=` still catches it.

G1. `1 2 3`: `q`. `3 2 1`: `p`. `3 2 1 1 2 3`: `r`. `1 1 2 2 3 3`: `t`. `3 3 2 2 1 1`: **none** (it needs both prints *before* the call). `1 2 3 3 2 1`: **none**: prints after the call can only go up, 1 to 3.

G2. **down**; **up**.

G3. `w(n - 1)`, then `print(n * 2)`: the smallest number prints first, so the print is after the call.

G4. `print(n)` (or `print(0)`) in the base case, before its `return`.

H1. Top to bottom: `sum_to n: 0` — running, `line 4: return 0`; `sum_to n: 1` — `line 6: 1 + ?`; `sum_to n: 2` — `line 6: 2 + ?`; `sum_to n: 3` — `line 6: 3 + ?`; Global frame — `line 8: result = ?`. The newest frame is on top.

H2. `n: 1`: `?` = 0, returns $1 + 0 = 1$. `n: 2`: `?` = 1, returns $2 + 1 = 3$. `n: 3`: `?` = 3, returns $3 + 3 = 6$. `result` is **6**.

H3. `n: 0` first (the newest); `n: 3` last. **101**: `n` = 100 down to 0.

I1. Lines: 1, 8, 3, 6, **3, 6, 3, 4, 6, 6, 8**. Frames: G, G, 2, 2, **1, 1, 0, 0, 1, 2, G**. Steps 9 and 10 go *back* into line 6 to finish the `+`; step 11 goes back into line 8 to store 3.

I2. **3** times (once in each `sum_to` frame); **once**; lines 2, 5 (comments) and 7 (blank).

I3. First: it makes the call `sum_to(1)` and waits (`2 + ?`). Second: it gets the 1 back, works out `2 + 1` and returns 3.

I4. Line **6** of the frame `n: 1`. **True**: a frame carries on from the place where it waited.

J1. Line 3: `3, 2, 1, 0` — **4**. Prints 1: `3, 2, 1` — **3** (the frame `n: 0` is already removed). Prints 2: `3, 2` — **2**. Prints 3: `3` — **1**.

J2. **1**; **3**; **4**. It prints while the stack grows; `countup` prints while it shrinks.

J3. Heights 1, 2, 3, 4, 3, 2, 1: a **mountain**. The stack grows going down the calls and shrinks coming back up.

J4. `1 11`, `2 21`, `3 31`. **3**: one in each of the frames `n:` 3, 2, 1. Still **20**: each frame has its own `x`, and no call can change another frame's. **No**: it returns on line 3, before line 4.

K1. `4 * 3 * factorial(2)`, `= 4 * 3 * 2 * factorial(1) = 4 * 3 * 2 * 1` (base case: `n <= 1`) `= 24`.

K2. `5 + 5 + multiply(5, 1)`, `= 5 + 5 + 5 + multiply(5, 0) = 5 + 5 + 5 + 0 = 15`.

K3. `= 1 + 1 + length("g")`, `= 1 + 1 + 1 + length("") = 1 + 1 + 1 + 0 = 3`.

K4. `= [3] + [2] + countdown_list(1)`, `= [3] + [2] + [1] + countdown_list(0) = [3] + [2] + [1] + [] = [3, 2, 1]`.

L1. `sum_to(4)`: **5**, 10. `sum_to(0)`: **1**, 0. `factorial(5)`: **5**, 120 (it stops at 1, not 0). `factorial(1)`: **1**, 1. `length("hello")`: **6**, 5. `multiply(3, 4)`: **5**, 12. `power(2, 10)`: **11**, 1024. `halvings(8)`: **4** (8, 4, 2, 1), 3. `halvings(100)`: **7** (100, 50, 25, 12, 6, 3, 1), 6. `countdown_list(4)`: **5**, `[4, 3, 2, 1]`.

L2. **one**; **waits**. So no frame is removed until the base case is reached: all the calls are on the stack at the deepest moment.

M1. `sum_to(50)`: 51, OK. `sum_to(3000)`: 3001, **E**. `factorial(2000)`: 2000, **E**. `halvings(1000000)`: only **20**, OK (halving gets small very fast). `length("ha" * 40)`: 81, OK. `power(2, 5000)`: 5001, **E**. `power(1, 5000)`: 5001, **E** (the answer is just 1, but the frames are still needed). `multiply(5000, 3)`: 4, OK. `multiply(3, 5000)`: 5001, **E**. `countdown(100)`: 101, OK.

M2. How **many** calls wait at once (the depth). `multiply(5000, 3)`: only `b` is counted down.

M3. Line **34**. One **frame**. **No**: the error is about too many frames ("depth"), not about the size of a number.

N. `0`; `1`; `1` (`-3 <= 1` stops at once); `0`; `0`; **RecursionError** (`b` goes $-2, -3, \ldots$ and never meets 0); `1`; `5`. **RecursionError** (the same reason); `[]`; **RecursionError**; `0`; `0`; **RecursionError** (`0 // 2` is 0, so `n` stays 0 forever); `5050`; **RecursionError**: 5000 calls need about 5000 frames, far past the limit of about 1000 — even though each call is smaller.

N1. "Odd inputs (0, negatives, `""`) still stop." Fix: `if n <= 0:` (`<=`, not `==`).

O. (a) ✓ ✓ ✗ (-2 moves away from 0). (b) ✓ ✗ ✓ (5, 3, 1, -1: it jumps over 0). (c) ✓ ✓ ✓. (d) ✓ ✓ ✗ (6, 3, 1 is fine; `0 // 2` stays 0). (e) ✗ ✓ ✓ (3 goes up, away from 0; 0 and -2 stop at once). (f) ✓ ✓ ✓ (a space is a letter too). (g) ✓ ✓ ✗ (going *up* is fine when the base case is above; 12 is past it). Always ask: does every call get closer to the base case?

P1. (a) 20, 10, 10. (b) 10, 0, -3. (c) 10, 0, 0 (one more call: `0 + sum_to(-1)`). (d) 10, 0, **E**. (e) 7 ($4+3+0$), 0, 0. (f) 6 ($4+2+0$), 0, 0. (g) 0, 0, 0 (everything is multiplied by the base case's 0). (h) 4, 0, 0. (i) 0, 0, 0 (nothing is added on the way up). (j) **E**, 0, 0 (it moves away from the base case). (k) **E**, 0, 0: `sum_to(n) - 1` calls with the same `n`, then takes 1 off the *answer*.

P2. **(c)**. **(b)**. **(h)**: it adds 1 for each call, so it counts.

Q. (a) same. (b) same: the test is turned round, and the base case is the last line. (c) same: `rest` holds the smaller answer. (d) **not the same**: `s(0)` (or any negative) is a `RecursionError`; from 1 up it agrees. (e) same. (f) **not the same**: `s(4)` is 4. The calls are made, but their answers are thrown away.

Q1. It is what happens when `n > 0` is `False`: no call is made, and 0 is returned ((b): the last line; (e): `total` stays 0).

R1. `0`; `1`; `0`; `[]`; `""`. Adding 0, multiplying by 1, and joining an empty list or string change nothing.

R2. `0`: the 0 wipes out every product. `0`; **TypeError**: `"*" + 0` joins a string and a number.

R3. `"4321"`; `"1234"`; `""`. `d1`: its own piece goes in front. **No**: all the pieces are the same star, so the order does not show.

S1. **RecursionError**. **N**: `n` never changes. Fix: `factorial(n - 1)`.

S2. `threes(9)`, `threes(12)` and `threes(0)`. `threes(10)` goes 10, 7, 4, 1, -2, ... and `threes(1)` goes 1, -2: both `RecursionError`. **J**. Fix: `if n <= 0:`. Lucky inputs hide this bug.

S3. `0`; `None` (the `1 + 0` is worked out, then thrown away); **TypeError**: it tries `1 + None`. **R**. Fix: `return 1 + length(s[1:])`.

S4. `11`; `1`. No crash, just a wrong answer: **O**. Fix: `return 0`. Test with an input whose answer you know.

S5. `[]`; **TypeError**: `1 + []` adds a number to a list. **O**. Fix: `[n] + countdown_list(n - 1)`.

S6. `2 1 0 -1 -2` ..., then a **RecursionError**: line 3 calls again before line 4 can test `n`. Fix: move the base case (lines 4–6) to the top, before the call.

S7. `12`; `3`; **RecursionError**: from 0, `b` goes $-1, -2, \ldots$ and never meets 1. Fix: `if b == 0: return 0`. Test the odd input 0.

T1. Prints `1`; `x` is `None`. `total(2)` prints `1`, then a **TypeError** (`2 + None`). A `print` shows a value but hands nothing back to the caller. Fix: `return n + total(n - 1)`.

T2. `1`, `Blastoff!`, `0`, `Blastoff!`, `-1`, ..., then a **RecursionError**. The base case prints but does not stop. Fix: a `return` after line 3.

T3. `h`, `i`, then an **IndexError**: `s[0]` of the empty string. Fix: test the base case first (move line 2 below line 4).

T4. `""`: slicing the empty string is no error, it just stays empty. So the calls never stop: **RecursionError**. Fix: add the base case `if s == "": return 0`.

T5. `0`; **RecursionError**. `a` gets smaller, but the base case tests `b`, which never changes. Fix: `multiply(a, b - 1)`.

T6. `None`; **TypeError** (`2 * None`). A bare `return` gives `None`. Fix: `return 1`.

T7. `1`; **NameError**: no function is called `factorial` now. Fix: `fact(n - 1)`. The base case hides the bug, so test a bigger input too.

T8. **Nothing**. Line **5**: the `return` on line 4 leaves the function first. Fix: take the word `return` off line 4.

U1. `0`; `1`; `b`; `e - 1`.

U2. `[]`; `n - 1`; `[n]`. The new number goes at the *end* this time.

U3. `<=` (or `==`); `""`; `s`; `n - 1`. `repeat("ab", 0)` is `""`.

U4. `stairs(n - 1)`, then `print("*" * n)`: the print comes *after* the call, so the smallest line prints first. The other order prints `***`, `**`, `*`.

V1. (a) `sum_to`. (b) `countdown`. (c) `multiply`. (d) `length`. When the `while` test is `False`, the loop stops: that is the **base case**. Each pass of the loop is one **smaller call** (`n - 1`, `b - 1`, `s[1:]`).

V2. `0`; `0`. `h`: `multiply(3, -1)` was a `RecursionError`, but `while b > 0` stops at once. (`b > 0` works like `<=` in a base case.)

W1. `12` $(6+4+2+0)$; `0`; `30`. Twice `sum_to(n)`: $2+4+\ldots+2n$.

W2. `8`; `1`; `1024`. 2 to the power `n`: it is `power(2, n)`.

W3. `7`; `9`. `a + b`. **Yes**: only `b` has to get closer to its base case.

W4. `1`; `0`; `1`. `n % 2`: 1 for an odd number, 0 for an even one.

W5. `3`; `2`; `0`. The even numbers from 1 to `n`. **7** calls: each call makes one more, whichever `return` it takes.

W6. `18` $(3+4+5+6)$; `5`; `0`. The sum of the whole numbers from `a` to `b`. It gets closer to `b`: "smaller job" means closer to the base case.

X1. `9` $(3+6)$; `6` (`sum_to(3)`); `720` (`factorial(6)`); `12`; `4`. `6`; `2`; `8`; `[2, 1]`; `5`.

X2. The **inner**: Python needs its value (3) before it can make the outer call. **No**: every inner frame is removed before `sum_to(3)` starts. **4** (during `sum_to(3)`).

Y1. `countup(3)` must wait for `countup(2)` to finish before it prints 3; the top frame prints first.

Y2. It also stops negative inputs, and calls that jump over 0 (like 1 to -1).

Y3. `sum_to(1)` works out `1 + 0` and throws it away, so it gives `None`; then `sum_to(2)` tries `2 + None`: `TypeError`.

Y4. Too many frames pile up on the call stack (past about 1000), usually when there is no base case: a stack overflow, the bug every programmer meets.

Z1. `if n <= 0: return 0`, then `return n * n + sum_squares(n - 1)`. Like `sum_to`, with `n * n` in place of `n`.

Z2. `if n <= 0`: print `Blastoff!` and `return`; else `print(n)` and `countdown_by(n - step, step)`. With `n == 0` it would jump over 0 (10, 7, 4, 1, -2).

Z3. `if n <= 0: return []`, then `return [0] + zeros(n - 1)`. Like `countdown_list`, with `[0]` in place of `[n]`.

Z4. `if a > b: return`, then `print(a)` and `print_range(a + 1, b)`. The job gets smaller by `a` going *up* to `b`.

Z5. `if n <= 0: print(0)` and `return`; then `print(n)`, `count_down_up(n - 1)`, `print(n)`. It is `echo`: one print going down, one coming back up.