Finger Exercises 22

Thinking Recursively

≈ 2 sittings of 1 hour + stretch

Comp 102 — Programming Fundamentals

What this is. A full workout for Lecture 22, in six levels. Lecture 21 showed *how* a recursion runs. This sheet trains you to *design* one. It starts with cutting a job smaller and ends with code you write.

How to do it. On **paper**, **no computer**, in **two sittings**: Levels 1–3 (Parts A–K), then Levels 4–6 (Parts L–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 has the four WISE questions. Ask them, in order, every time you design a recursion.

LEVEL 1 • The Recipe

four questions, and the pieces you cut

Part A • Do You Remember?

RECALL ≈ 5 min

A1. The four WISE questions. **W**: What is the _____ case? **I**: If not easy, how do I make it _____? **S**: _____ the smaller job (trust it!). **E**: _____: use it to answer the big job.

A2. Five ways to make a job smaller. A number: _____ or _____. A string: drop the first letter, _____, or drop both ends, _____. A list: _____.

A3. Stuck on a design? Write the answers for _____ inputs, each one a bit bigger. Then look for the answer of the row _____ inside each answer.

A4. A tower of n disks needs _____ moves. One more disk gives _____ the moves. The family is _____.

A5. Circle **T** (true) or **F** (false): *(T / F)*

(a) A recursion must always use `n - 1`. T / F

(b) In **S** you trace the smaller call by hand. T / F

(c) The base case is where the shrinking stops. T / F

(d) `is_palindrome` can answer before it reaches its base case. T / F

(e) The same recipe works on numbers, strings and lists. T / F

(f) `find_max` could use `[]` as its base case. T / F

(g) A short program always runs fast. T / F

(h) `hanoi` calls itself twice. T / F

A6.

```
1 def digit_sum(n):
2     if n < 10:
3         return n
4     rest = n // 10
5     last = n % 10
6     return digit_sum(rest) + last
```

Write the WISE letter (or letters) of each line, the way the slide did.

lines 2–3: _____ line 4: _____

line 5: _____ line 6: _____

Part B • Cut It Smaller

TRACE ≈ 5 min

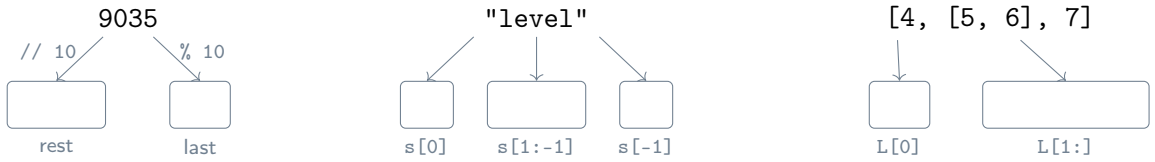
Every recursion cuts its input. Write the value. If the line stops with an error, write the error's name.

Expression	Value	Expression	Value	Expression	Value
`472 // 10`	_____	`"cat"[1:]`	_____	`"abba"[1:-1]`	_____
`472 % 10`	_____	`"cat"[0]`	_____	`"ab"[1:-1]`	_____
`7 // 10`	_____	`"cat"[-1]`	_____	`"a"[1:-1]`	_____
`7 % 10`	_____	`"c"[1:]`	_____	`[3, 7, 2][1:]`	_____
`50 // 10`	_____	`""[1:]`	_____	`[9][1:]`	_____
`50 % 10`	_____	`""[0]`	_____	`[][1:]`	_____
`0 // 10`	_____	`"cat"[:-1]`	_____	`[[2, 3], 4][0]`	_____

B1. How many times can you do `n = n // 10` before `n < 10`?

`n = 4096`: ____ `n = 7`: ____ `n = 50000`: ____ `n = 10`: ____

B2. Fill in the empty boxes. Then **circle** the box that goes into the smaller call.



Part C • The Smaller Copy

CLASSIFY ≈ 5 min

C1. Fill in the table, like the lecture's "Many Ways to Make a Job Smaller". In the last column, write the input where the shrinking **stops**.

the job	the smaller copy	how	stops at
`sum_to(5)`	`sum_to(4)`	`n - 1`	`0`
`count_digits(4096)`	_____	_____	_____
`stutter("cat")`	_____	_____	_____
`is_palindrome("level")`	_____	_____	_____
`is_sorted([2, 5, 5, 9])`	_____	_____	_____
`find_max([3, 7, 2])`	_____	_____	_____
`moves(5)`	_____	_____	_____
`mystery("banana")`	_____	_____	_____
`hanoi(3, "A", "B", "C")`	_____	_____	_____
	_____		

C2. Where does the shrinking stop? Write the input of the last call.

`is_palindrome("abba")` _____ `stutter("abcd")` _____

`is_palindrome("abcba")` _____ `count_digits(7)` _____

`reverse("abcd")` _____ `is_sorted([1, 2, 3])` _____

C3. Two lecture functions can call **themselves twice** in one call. One of them is in the table. Which two? _____ and _____

Part D • The Easiest Case

CLASSIFY ≈ 5 min

D1. For each job, write the easiest input and its answer. The first row is done.

the job	the easiest input	its answer
add up the numbers of a list	`[]`	`0`
multiply all the numbers of a list	_____	_____
join a list of words into one string (`["ab", "c"]` → `"abc"`)	_____	_____
count the vowels of a string	_____	_____
the biggest number of a list	_____	_____
reverse a string	_____	_____
are *all* the numbers of a list positive?	_____	_____
is *any* number of a list negative?	_____	_____
count the digits of a number	_____	_____
the list `[1, 2, ..., n]`	_____	_____
the moves for a tower of n disks	_____	_____

D2. One job in the table has **no** answer for the empty input. Which one, and why?

_____

D3. Fill the blank in line 3 of each function. Then answer the questions.

```
1 def all_positive(L):
2     if L == []:
3         return ______
4     if L[0] <= 0:
5         return False
6     return all_positive(L[1:])
```

```
1 def any_negative(L):
2     if L == []:
3         return ______
4     if L[0] < 0:
5         return True
6     return any_negative(L[1:])
```

Now someone swaps your two blanks.

`all_positive([3, 5])` now gives _____, and `any_negative([3, 5])` now gives _____.

What does each one give now, for *every* list? _____

LEVEL 2 • Trust and Extend

pretend the smaller job works, then finish the big one

Part E • Trust It

TRUST ≈ 7 min

All the functions are the lecture's. **Do not trace the smaller call.** You know what the function does, so just write its answer. Then finish the big job, the way the function's last line does. Row 1 is done.

	the big job	the smaller job	S: trust, it gives	E: so the answer is
1	`digit_sum(5821)`	`digit_sum(582)`	`15`	`15 + 1 = 16`
2	`count_digits(90210)`	_____	_____	_____
3	`reverse("python")`	`reverse("ython")`	_____	_____
4	`stutter("dog")`	_____	_____	_____
5	`mystery("idea")`	`mystery("dea")`	_____	_____
6	`mystery("bead")`	_____	_____	_____
7	`find_max([4, 11, 6, 2])`	`find_max([11, 6, 2])`	_____	_____
8	`find_max([20, 11, 6, 2])`	_____	_____	_____
9	`is_palindrome("refer")`	_____	_____	_____
10	`is_palindrome("robot")`	_____	_____	_____
11	`is_sorted([3, 8, 5])`	`is_sorted([8, 5])`	_____	_____
12	`moves(10)`	`moves(9)`	`511`	_____
13	`sum_to(100)`	`sum_to(99)`	`4950`	_____

E1. In row 7 you did not know *how* `find_max` finds the `11`. Why is that fine?

_____

E2. Row 10 needed no smaller job. Which other lecture function can also answer with no smaller call, when its input is long? _____

Part F • Trust a Stranger

TRUST ≈ 6 min

These functions have **no code** here. You only know their job. That is enough: trust the smaller call, then finish the big job.

the function and its job	S: the smaller job gives	E: so the big job gives
`count_e(s)`: how many 'e' in `s`	`count_e("leven")` → _____	`count_e("eleven")` → _____
`evens(L)`: the even numbers of `L`, in order	`evens([7, 2, 9])` → _____	`evens([4, 7, 2, 9])` → _____
`largest_digit(n)`: the biggest digit of `n`	`largest_digit(_____)` → ____	`largest_digit(3972)` → _____
`double_all(L)`: every number times 2	_____ → _____	`double_all([3, 1, 4])` → _____
`no_spaces(s)`: `s` without its spaces	`no_spaces("a b")` → _____	`no_spaces(" a b")` → _____
`power2(n)`: 2 to the power `n`	`power2(9)` → _____	`power2(10)` → _____
`count_big(L)`: how many numbers are over 10	_____ → _____	`count_big([12, 5, 30, 11])` → _____
`mirror(s)`: `s`, then `s` backwards	`mirror("bc")` → _____	`mirror("abc")` → _____

F1. Now write the last line of four of them. Use the smaller call.

`double_all`: return _____

`power2`: return _____

`mirror`: return _____

`count_e`, when `s[0] == "e"`: return _____

Part G • Front or Back? TRACE ≈ 6 min

E decides *where* the kept piece goes. Each row is a different last line for this function. Write what `f("abc")` gives.

```
1 def f(s):
2     if s == "":
3         return ""
4     return ______
```

Each row puts its own line 4 in the blank. Work from the **bottom**: `f("c")` first, then `f("bc")`, then `f("abc")`. Each one uses the answer before it.

line 4	`f("abc")`	line 4	`f("abc")`
(a) `s[0] + f(s[1:])`	_____	(g) `f(s[1:])`	_____
(b) `f(s[1:]) + s[0]`	_____	(h) `s[0] + "-" + f(s[1:])`	_____
(c) `s[0] + s[0] + f(s[1:])`	_____	(i) `s[-1] + f(s[:-1])`	_____
(d) `f(s[1:]) + s[0] + s[0]`	_____	(j) `f(s[:-1]) + s[-1]`	_____
(e) `s[0] + f(s[1:]) + s[0]`	_____	(k) `s[0]`	_____
(f) `"*" + f(s[1:])`	_____	(l) `s[0] + f(s[2:])`	_____

G1. Which two rows give back `s` unchanged? _____ Which two reverse it? _____ Which row is `stutter`? ____ Which one is not recursive? ____

G2. The same idea on a list. Write what `g([1, 2, 3])` gives, or the error.

```
1 def g(L):
2     if L == []:
3         return []
4     return ______
```

(a) `[L[0]] + g(L[1:])` _____

(b) `g(L[1:]) + [L[0]]` _____

(c) `[L[0] * 10] + g(L[1:])` _____

(d) `[L[0], L[0]] + g(L[1:])` _____

(e) `L[0] + g(L[1:])` _____

(f) `g(L[1:])` _____

Part H • Write Small Examples TRACE ≈ 6 min

Fill in each answer. Then **circle** the answer of the row above inside each answer, as the slide coloured it orange.

H1. `stutter`

`stutter("")` → _____

`stutter("t")` → _____

`stutter("at")` → _____

`stutter("cat")` → _____

H2. `upto(n)`: the list $1, 2, \ldots, n$

`upto(0)` → _____

`upto(1)` → _____

`upto(2)` → _____

`upto(3)` → _____

H3. `digit_sum`

`digit_sum(9)` → _____

`digit_sum(93)` → _____

`digit_sum(935)` → _____

`digit_sum(9351)` → _____

H4. `evens(L)`: the even numbers of `L`

`evens([])` → _____

`evens([8])` → _____

`evens([3, 8])` → _____

`evens([6, 3, 8])` → _____

`evens([5, 6, 3, 8])` → _____

H5. Now write the rule each ladder shows.

`stutter(s)` = _____ + `stutter(s[1:])`

`upto(n)` = `upto(n - 1)` + _____

`digit_sum(n)` = `digit_sum(`_____`)` + _____

`evens(L)` = _____ + `evens(L[1:])` if `L[0]` is even, else just _____

LEVEL 3 • Following a Design

unfold the calls, count them, and try odd inputs

Part I • Unfold the Calls

TRACE ≈ 8 min

Unfold each call, one line for each step, the way the slides did. The first line is given.

I1. `digit_sum(9035)`

= `digit_sum(`____`)` + ____

= `digit_sum(`____`)` + _____

= `digit_sum(`____`)` + _____

= _____ = ____

I2. `reverse("star")`

= `reverse(`_____`)` + ____

= `reverse(`_____`)` + _____

= `reverse(`_____`)` + _____

= _____ = _____

I3. `moves(4)`. Going **down**, each call waits for a smaller one. Coming back **up**, each one works out `2 * (smaller) + 1`.

call	going down: waits for	coming up: returns
`moves(4)`	`moves(3)`	`2 *` ____ `+ 1 =` ____
`moves(3)`	_____	`2 *` ____ `+ 1 =` ____
`moves(2)`	_____	`2 *` ____ `+ 1 =` ____
`moves(1)`	_____	`2 *` ____ `+ 1 =` ____
`moves(0)`	nothing: base case	____

Which row is worked out **first**? _____

I4. Trace `is_palindrome`. Write each string it checks and what it compares, until it answers.

`is_palindrome("level")`: _____

`is_palindrome("noon")`: _____

`is_palindrome("abcdba")`: _____

I5. `find_max([4, 9, 2, 6])`. Write the calls going **down**, then the "me or the best of the rest?" answers coming back **up**.

down:

`find_max([4, 9, 2, 6])`

`find_max(`_____`)`

`find_max(`_____`)`

`find_max(`_____`)` → base: ___

up:

___ vs ___ → ___

___ vs ___ → ___

___ vs ___ → ___ answer: ___

I6. `is_sorted([1, 3, 3, 2, 5])`. Write each list it checks and the pair it compares, until it answers.

_____

Does it ever look at the 5? _____

Part J • Count the Calls

COUNT ≈ 4 min

How many calls does each one make, the first one included? Every call of the same function counts.

call	calls	call	calls
`count_digits(50000)`	_____	`is_palindrome("abccba")`	_____
`digit_sum(7)`	_____	`is_palindrome("abcdefa")`	_____
`reverse("star")`	_____	`is_sorted([1, 2, 3, 0])`	_____
`reverse("")`	_____	`find_max([8, 1, 5, 2, 7])`	_____
`stutter("star")`	_____	`moves(5)`	_____
`mystery("banana")`	_____	`hanoi(2, "A", "B", "C")`	_____
`is_palindrome("racecar")`	_____	`nested_sum([1, [2, 3]])`	_____

J1. Most of these make one call for each _____. `is_palindrome` makes about one call for every _____ letters. Which two functions can stop **early**, before the base case?

_____

Part K • Value or Error

TRACE ≈ 7 min

All the functions are the lecture's. Some get inputs they were not made for. Write the value. If the call stops with an error, write the error's name.

Call	Value or error	Call	Value or error
`digit_sum(0), digit_sum(10)`	_____	`is_palindrome("Racecar")`	_____
`digit_sum(1000)`	_____	`is_palindrome("nurses run")`	_____
`digit_sum(-25)`	_____	`is_palindrome([1, 2, 1])`	_____
`count_digits(0)`	_____	`find_max([5, 5])`	_____
`count_digits(-512)`	_____	`find_max([-3, -8, -1])`	_____
`reverse(""), reverse("a")`	_____	`find_max("hello")`	_____
`reverse("noon")`	_____	`find_max([])`	_____
`reverse("a b")`	_____	`is_sorted([3, 3, 3])`	_____
`reverse([1, 2, 3])`	_____	`is_sorted("abc")`	_____
`reverse(123)`	_____	`mystery("AEIOU")`	_____
`stutter("ab ")`	_____	`mystery("rhythm")`	_____
`stutter([1, 2])`	_____	`moves(-1)`	_____
`is_palindrome("")`	_____	`nested_sum([1, "a"])`	_____

LEVEL 4 • Read It, Fix It

mysteries, bugs, and the inputs that hide them

Part L • Read & Explain

EXPLAIN ≈ 7 min

For each function, write the value of the call, then say in **one** sentence what the function does.

L1.

```
1 def g1(s):
2     if s == "":
3         return ""
4     if s[0] in "aeiou":
5         return g1(s[1:])
6     return s[0] + g1(s[1:])
```

`g1("banana")` is _____

It _____

L2.

```
1 def g2(n):
2     if n < 10:
3         return n
4     best = g2(n // 10)
5     last = n % 10
6     if last > best:
7         return last
8     return best
```

`g2(3972)` is _____

It _____

L3.

```
1 def g3(L):
2     if L == []:
3         return 0
4     if L[0] < 0:
5         return 1 + g3(L[1:])
6     return g3(L[1:])
```

`g3([3, -1, 0, -7])` is _____

It _____

L4.

```
1 def g4(L):
2     if L == []:
3         return []
4     return g4(L[1:]) + [L[0]]
```

`g4([1, 2, 3])` is _____

It _____

L5.

```
1 def g5(L, x):
2     if L == []:
3         return False
4     if L[0] == x:
5         return True
6     return g5(L[1:], x)
```

`g5([4, 8, 1], 8)` is _____

It _____

L6.

```
1 def g6(n):
2     if n < 10:
3         return n % 2
4     return g6(n // 10) + n % 2
```

`g6(3572)` is _____

It _____

Part M • Bug Hunt

FIX ≈ 10 min

Each function below is a lecture function with **one** change. Answer the questions. Then name the WISE step that went wrong (**W**, **I**, **S** or **E**) and write the fix.

M1.

```
1 def reverse(s):
2     if len(s) <= 1:
3         return s
4     return s[0] + reverse(s[1:])
```

`reverse("cat")` gives _____.

Step: ____ Fix line 4: _____

M2.

```
1 def digit_sum(n):
2     if n < 10:
3         return n
4     rest = n % 10
5     last = n // 10
6     return digit_sum(rest) + last
```

`digit_sum(472)` gives _____.
`digit_sum(47)` gives _____.
Step: ____

M3.

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

`stutter("cat")` gives _____.
`stutter("a")` gives _____.
Which step is missing? ____

M4.

```
1 def find_max(L):
2     if len(L) == 1:
3         return L[0]
4     best_of_rest = find_max(L[1:])
5     if L[0] < best_of_rest:
6         return L[0]
7     return best_of_rest
```

`find_max([3, 7, 2])` gives _____.
`find_max([4, 4])` gives _____.
What does it find now? _____
Step: ____

M5.

```
1 def moves(n):
2     if n == 0:
3         return 0
4     return 2 * moves(n - 1)
```

`moves(3)` gives _____. `moves(64)` gives _____.
Which move did it forget? _____
Step: ____

M6.

```
1 def count_vowels(s):
2     if s == "":
3         return 0
4     rest = count_vowels(s[1:])
5     if s[0] in "aeiou":
6         return 1 + rest
```

`count_vowels("a")` gives _____.
`count_vowels("hi")` gives _____.
`count_vowels("ah")` gives _____.
Step: ____ Fix: _____

M7.

```
1 def nested_sum(L):
2     if L == []:
3         return 0
4     first = L[0]
5     rest = nested_sum(L[1:])
6     if type(first) == list:
7         return first + rest
8     return first + rest
```

`nested_sum([1, 2])` gives _____.
`nested_sum([[1], 2])` gives _____.
Step: ____ Fix line 7: _____

M8.

```
1 def is_sorted(L):
2     if len(L) <= 1:
3         return True
4     if L[0] > L[1]:
5         return False
6     return is_sorted(L[2:])
```

Tick the lists where it gives the **right** answer:
☐ `[1, 2, 3, 4]` ☐ `[4, 3, 2, 1]`
☐ `[1, 3, 2, 4]` ☐ `[1, 2, 4, 3]`
Which pair is never compared? _____
Step: ____

M9.

```
1 def is_palindrome(s):
2     if len(s) == 1:
3         return True
4     if s[0] != s[-1]:
5         return False
6     return is_palindrome(s[1:-1])
```

Write the value or the error:
`"racecar"`: _____ `"ab"`: _____
`"abba"`: _____ `""`: _____
Step: ____

M10. Two changed versions of `hanoi`. For each one, write what `hanoi(2, "A", "B", "C")` prints. Then say which move is **impossible**, and why.

```
1 def hanoi(n, start, spare, goal):
2     if n == 0:
3         return
4     print("Move disk", n, "from",
5           start, "to", goal)
6     hanoi(n - 1, start, goal, spare)
7     hanoi(n - 1, spare, start, goal)
```

(a) _____

_____

```
1 def hanoi(n, start, spare, goal):
2     if n == 0:
3         return
4     hanoi(n - 1, start, goal, spare)
5     print("Move disk", n, "from",
6           start, "to", goal)
7     hanoi(n - 1, start, spare, goal)
```

(b) _____

_____

Part N • Lucky Inputs

FIX ≈ 6 min

A bug can hide. Some inputs give the right answer even with the bug. Each row has a buggy function. Tick every test that **catches** the bug (a wrong answer or an error).

	the buggy function	tick the tests that catch it			
1	`product` with base `if L == []: return 0`	☐ `[5, 0, 9]`	☐ `[2, 3]`	☐ `[]`	☐ `[0]`
2	`find_max` with base `if L == []: return 0`	☐ `[3, 7, 2]`	☐ `[-3, -8]`	☐ `[5]`	☐ `[0, -1]`
3	`count_digits` with base `if n == 0: return 0`	☐ `4096`	☐ `7`	☐ `0`	☐ `10`
4	`is_palindrome` with base `if len(s) == 1:`	☐ `"racecar"`	☐ `"abba"`	☐ `"ab"`	☐ `"aa"`
5	`stutter` with base `if len(s) == 1: return s + s`	☐ `"cat"`	☐ `"a"`	☐ `""`	☐ `"zz"`
6	`reverse` with last line `return s[-1] + reverse(s[1:-1]) + s[0]`	☐ `"cat"`	☐ `"ab"`	☐ `"abcd"`	☐ `""`

N1. Look at the tests that caught the bugs. Which inputs should you **always** try on a new recursive function? Name four.

_____

LEVEL 5 • Towers and Lists Inside Lists

two calls, and shapes inside shapes

Part O • Towers by Hand

TRACE ≈ 7 min

```
1 def hanoi(n, start, spare, goal):
2     if n == 0:
3         return
4     hanoi(n - 1, start, goal, spare)
5     print("Move disk", n, "from",
6           start, "to", goal)
7     hanoi(n - 1, spare, start, goal)
```

O1. What does `hanoi(2, "A", "B", "C")` print?

`Move disk ___ from ___ to ___`

`Move disk ___ from ___ to ___`

`Move disk ___ from ___ to ___`

`hanoi(1, "X", "Y", "Z")` prints

_____

`hanoi(0, "X", "Y", "Z")` prints

_____

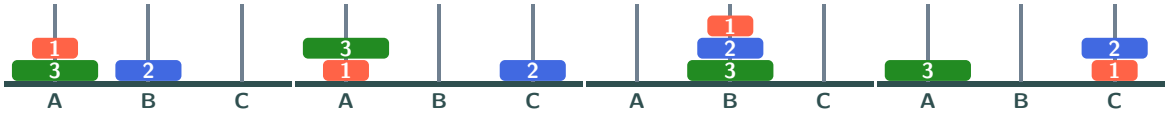
O2. Now the goal is **B**: `hanoi(3, "A", "C", "B")`. Write the 7 moves as "disk, from, to".

1. _____ 2. _____ 3. _____ 4. _____

5. _____ 6. _____ 7. _____

O3. Tick each picture that **can** happen during the game (no big disk on a smaller one):

(a) ☐ (b) ☐ (c) ☐ (d) ☐



O4. `hanoi(3, "A", "B", "C")` has started. **Draw** the disks (as numbered boxes) after move 4, and after move 6.

after move 4 after move 6



O5. With 3 disks, disk 1 moves ____ times, disk 2 ____ times, disk 3 ____ time(s). With 4 disks, disk 1 moves ____ times. The biggest disk moves on move number ____ (3 disks) and ____ (4 disks).

Part P • Inside `hanoi`

TRACE ≈ 5 min

P1. Use the listing in Part O. Write the arguments of the two smaller calls.

the call	line 4 calls	line 7 calls
`hanoi(3, "A", "B", "C")`	`hanoi(___, ___, ___, ___)`	`hanoi(___, ___, ___, ___)`
`hanoi(2, "A", "C", "B")`	`hanoi(___, ___, ___, ___)`	`hanoi(___, ___, ___, ___)`

P2. In one sentence: in line 4, why is the `goal` passed in the place of the `spare`?

_____

P3. `hanoi(4, "A", "B", "C")` prints 15 lines. Line 1 is `Move disk __ from __ to __`. Line 8 is `Move disk __ from __ to __`. Line 15 is `Move disk __ from __ to __`.

P4. A student makes **1** disk the base case: if `n == 1`, print the one move of disk 1 and return. Now `hanoi(3, "A", "B", "C")` prints _____ lines, and `hanoi(0, "A", "B", "C")` gives _____.

Part Q • Counting the Moves

COUNT ≈ 5 min

Q1. Fill in the table with `moves(n) = 2 * moves(n - 1) + 1`. Each box uses the box before it.

disks n	0	1	2	3	4	5	6	7	8	10
moves	0	___	___	___	___	___	___	___	___	____

Q2. One move a second. How long for each tower? Round to a nice unit.

10 disks: _____ 20 disks: _____ 30 disks: _____

Q3. A day has 86 400 seconds. One move a second, what is the biggest tower the priests can finish in **one day**? _____ disks. Hint: $2^{16} = 65\,536$ and $2^{17} = 131\,072$.

Q4. How many **calls** of `hanoi` does each one make, the first one included?

`hanoi(1, ...)`: ____ `hanoi(2, ...)`: ____ `hanoi(3, ...)`: ____

Q5. `moves(64)` gives its answer at once. `hanoi(64, "A", "B", "C")` would run for 585 billion years. Both are recursive. Why is one so fast?

_____

Part R • Lists Inside Lists

TRACE ≈ 6 min

```
1 def nested_sum(L):
2     if L == []:
3         return 0
4     first = L[0]
5     rest = nested_sum(L[1:])
6     if type(first) == list:
7         return nested_sum(first) + rest
8     return first + rest
```

`L = [1, [2, 3], [4, [5, 6]]]` is three boxes:

`1` | `[2, 3]` | `[4, [5, 6]]`

R1. Write the value or the error.

`len(L)`, `L[1]` _____

`L[2][1]`, `L[2][1][0]` _____

`type(L[0]) == list` _____

`type(L[2]) == list` _____

`nested_sum(L)` _____

R2. Draw `[[1, 2], [3, [4, [5]]]]` as boxes, the way `L` is drawn above. How many boxes? ____ Its `nested_sum`? ____

R3. Write the value or the error:

`nested_sum([])` _____ `nested_sum([1, [2], [[3]], [[[4]]])` _____

`nested_sum([[]])` _____ `nested_sum([[], [[]]])` _____

`nested_sum([[[7]]])` _____ `nested_sum(5)` _____

R4. List every call of `nested_sum` that `nested_sum([1, [2, 3]])` makes, the first one included.

_____

R5. Change line 8 to `return 1 + rest`. Now `nested_sum([1, [2, 3], [[4]]])` gives ____, and `nested_sum([10, 20])` gives ____.

What does the function count now? _____

Part S • Recursion or a Loop?

CLASSIFY ≈ 4 min

S1. Python stops at about **1000** calls deep. Tick the calls that stop with `RecursionError`:

☐ `sum_to(500)` ☐ `sum_to(5000)` ☐ `reverse("a" * 2000)`

☐ `is_palindrome("a" * 1500)` ☐ `count_digits(10 ** 500)` ☐ `is_palindrome("ab" * 1000)`

☐ `moves(64)` ☐ `find_max([])` ☐ `mystery("a" * 3000)`

S2. Recursion or a loop? Write **R** or **L**.

(a) add the numbers 1 to 100,000 ___ (b) add up a list with lists inside, any depth ___

(c) print the moves for 5 disks ___ (d) count the letters of a 3000-letter string ___

(e) count the files in folders inside folders ___ (f) find the biggest of a million numbers ___

LEVEL 6 • Design It Yourself

from the four questions to the code

Part T • WISE Grids

DESIGN ≈ 8 min

Four new jobs. Answer the four questions in words, the way the slides did ("**W**: one digit: give it back."). Do not write code yet: that is Part Y.

`count_a(s)`: how many 'a' in `s`?

`count_a("banana")` is 3

W _____

I _____

S _____

E _____

`product(L)`: all the numbers multiplied

`product([2, 3, 4])` is 24

_____

_____

_____

_____

`all_even(L)`: are all the numbers even?

`all_even([2, 8, 4])` is `True`

W _____

I _____

S _____

E _____

`last(L)`: the last item of a list

`last([4, 9, 7])` is 7

_____

_____

_____

_____

Part U • Pick the Base Case

CLASSIFY ≈ 6 min

The last lines of each function are fixed. Tick every base case that makes it right for **every** input it is made for. For each one you do not tick, write an input that goes wrong.

U1. `reverse`, made for every string. The last line is `return reverse(s[1:]) + s[0]`.

- ☐ `if s == "": return ""` _____
- ☐ `if len(s) == 1: return s` _____
- ☐ `if len(s) <= 1: return ""` _____
- ☐ `if len(s) <= 1: return s` _____

U2. `find_max`, made for lists with **at least one** number. Lines 4–7 are the lecture's.

- ☐ `if len(L) == 1: return L[0]` _____
- ☐ `if L == []: return 0` _____
- ☐ `if len(L) == 1: return L` _____
- ☐ `if L == []: return L[0]` _____

U3. `count_digits`, made for `n >= 0`. The last line is `return count_digits(n // 10) + 1`.

- ☐ `if n < 10: return 1` _____
- ☐ `if n == 0: return 0` _____
- ☐ `if n < 10: return 0` _____
- ☐ `if n <= 9: return 1` _____

U4. `is_sorted`, made for every list. The last lines are `if L[0] > L[1]: return False` and `return is_sorted(L[1:])`.

- ☐ `if len(L) <= 1: return True` _____
- ☐ `if len(L) == 1: return True` _____
- ☐ `if L == []: return True` _____
- ☐ `if len(L) <= 1: return False` _____

Part V • Match the Pieces

CLASSIFY ≈ 4 min

Each job needs a base case answer and a last line. Here `f` stands for the function itself. Write a letter and a number for each job.

base case returns

P `0` **Q** `1` **R** `""`
S `[]` **T** `n` **U** `s`

last line

1 `return f(s[1:]) + s[0]`
2 `return 2 * f(n - 1) + 1`
3 `return L[0] + f(L[1:])`
4 `return f(n // 10) + 1`
5 `return s[0] + s[0] + f(s[1:])`
6 `return f(n - 1) + [n]`
7 `return L[0] * f(L[1:])`
8 `return f(n // 10) + n % 10`

the job	base	last line
`count_digits(n)`	____	____
`digit_sum(n)`	____	____
`stutter(s)`	____	____
`reverse(s)`	____	____
`product(L)`	____	____
`moves(n)`	____	____
`upto(n)`	____	____
`total(L)`: add up a list	____	____

Part W • Same Job, Two Designs

CLASSIFY ≈ 6 min

Each box is a **new** design of a lecture function. Most of them cut from the other end. Write **same** if it gives the lecture's answer for every input the function is made for. If not, write one call that shows the difference.

(a) `reverse`

```
def reverse(s):
    if len(s) <= 1:
        return s
    return s[-1] + reverse(s[:-1])
```

_____

(b) `find_max`

```
def find_max(L):
    if len(L) == 1:
        return L[0]
    best = find_max(L[:-1])
    if L[-1] > best:
        return L[-1]
    return best
```

_____

(c) `is_palindrome`

```
def is_palindrome(s):
    if len(s) <= 1:
        return True
    ok = s[0] == s[-1]
    return ok and is_palindrome(s[1:-1])
```

_____

(d) `upto`

```
def upto(n):
    if n == 0:
        return []
    return [n] + upto(n - 1)
```

_____

(e) `digit_sum`, for `n >= 0`

```
def digit_sum(n):
    if n == 0:
        return 0
    return digit_sum(n // 10) + n % 10
```

_____

(f) `stutter`

```
def stutter(s):
    if s == "":
        return ""
    return s[0] + stutter(s[1:]) + s[0]
```

_____

Part X • Two Inputs: Which One Shrinks?

CLASSIFY ≈ 5 min

These functions take **two** inputs. Only one of them gets smaller.

```
def count_char(s, c):
    if s == "":
        return 0
    rest = count_char(s[1:], c)
    if s[0] == c:
        return 1 + rest
    return rest
```

```
def contains(L, x):
    if L == []:
        return False
    if L[0] == x:
        return True
    return contains(L[1:], x)
```

```
def repeat(s, n):
    if n == 0:
        return ""
    return s + repeat(s, n - 1)
```

X1. Fill in the table. For `hanoi`, look back at Part O.

function	the input that gets smaller	the inputs that do not
`count_char`	_____	_____
`contains`	_____	_____
`repeat`	_____	_____
`hanoi`	_____	_____

X2. Write the value or the error.

`count_char("banana", "a")` _____ `repeat("ab", 3)` _____

`count_char("banana", "A")` _____ `repeat("ab", 0)` _____

`contains([4, 8, 1], 8)` _____ `repeat("", 5)` _____

`contains([], 5)` _____ `repeat("ha", -1)` _____

X3. How many calls does each one make, the first one included?

`contains([4, 8, 1], 8)`: ____ `contains([4, 8, 1], 9)`: ____

Part Y • Finish the Design

FILL IN ≈ 7 min

Fill in every blank. Use your WISE answers from Part T.

Y1.

```
def count_a(s):
    if s == "":
        return ______
    rest = count_a(______)
    if ______ == "a":
        return ____________
    return rest
```

```
def product(L):
    if L == []:
        return ______
    return ______ * product(______)
```

`count_a("Aa")` gives _____.

`product([5, 0, 9])` gives _____.

Y2.

```
def all_even(L):
    if L == []:
        return ______
    if L[0] % 2 == ______:
        return False
    return ______
```

```
def last(L):
    if len(L) == ______:
        return ______
    return ______
```

`all_even([2, 5, 4])`: _____

`all_even([])`: _____

`last([7])`: _____ `last([])`: _____

Y3.

```
def upto(n):
    if n == ______:
        return ______
    return upto(______) + ______
```

`upto(3)` is `[1, 2, 3]` (your ladder in Part H). What does your `upto(-2)` give? _____

Part Z • Write the Code

STRETCH ≈ 15 min, optional

For each job, write the four WISE answers first, then the code. Then check your code on the edge inputs.

Z1. `digit_product(n)`: all the digits multiplied. `digit_product(472)` is `56`, `digit_product(7)` is `7`. What does `digit_product(305)` give?

Z2. `is_power_of_two(n)`, for `n >= 1`: `is_power_of_two(8)` is `True`, `is_power_of_two(12)` is `False`. Hint: cut with `n // 2`, and an odd number over 1 answers `False` at once.

Z3. `flatten(L)`: one flat list of all the numbers. `flatten([1, [2, [3]], 4])` is `[1, 2, 3, 4]`. Hint: start from `nested_sum`, with lists in place of numbers. What is `flatten([[], [[]]])`?

Z4. `depth(L)`: how deep the lists go. `depth([1, 2])` is `1`, `depth([1, [2, [3]]])` is `3`, `depth([])` is `1`. What is `depth([[]])`?

Console Quests

tonight · 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 `lec22.py`.

1. Type `for n in range(1, 6): hanoi(n, "A", "B", "C")`, then count the lines each tower prints. Do you see $2^n - 1$?
2. Type `reverse([1, 2, 3])`. The function was written for strings. Which line fails, and why does the error mention `list` and `int`? Then try `find_max("hello")`.
3. Type your `count_a`, `product`, `all_even` and `last` from Part Y. Test each one on its edge inputs first: `""`, `[]`, one item.

Stuck on more than two items? Re-read the Lecture 22 slides "WISE: Four Questions", "Think: Write Small Examples", "Many Ways to Make a Job Smaller", "The Smaller Copy Inside" and "Nested Sum, in Python" before the next class — then redo Parts E, G and U 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. **easiest** (the base case); **smaller**; **Solve**; **Extend**.

A2. `n - 1`, `n // 10`; `s[1:]`, `s[1:-1]`; `L[1:]`.

A3. **small**; **above**.

A4. $\mathbf{2^n - 1}$; **twice** (plus one); $\mathbf{O(2^n)}$, the exponential family.

A5. (a) **F** — `n // 10`, `s[1:]`, `s[1:-1]`, `L[1:]` all work. You choose. (b) **F** — you *trust* it. (c) **T**. (d) **T** — ends that differ give `False` at once. (e) **T**. (f) **F** — an empty list has no biggest number. (g) **F** — `hanoi(64)` is 6 lines and 585 billion years. (h) **T**.

A6. Lines 2–3: **W** (one digit: give it back). Line 4: **I** (cut off the last digit). Line 5: **E** (the piece we add at the end). Line 6: **S** and **E** (solve the smaller job, then add `last`).

B. `47`, `2`, `0`, `7`, `5`, `0`, `0`. `"at"`, `"c"`, `"t"`, `""` (empty), `""` (a slice never crashes), **IndexError** (an empty string has no place 0), `"ca"`. `"bb"`, `""`, `""`, `[7, 2]`, `[]`, `[]`, `[2, 3]` (the first item is itself a list).

B1. **3** (409, 40, 4); **0**; **4** (5000, 500, 50, 5); **1** (1). Always one less than the number of digits.

B2. `903` and `5`: circle `903`. 'l', `"eve"`, 'l': circle `"eve"` (if the two 'l' match). `4` and `[[5, 6], 7]`: circle `[[5, 6], 7]`.

C1. `count_digits(409)`, `n // 10`, one digit (4). `stutter("at")`, `s[1:]`, `""`. `is_palindrome("eve")`, `s[1:-1]`, one letter (`"v"`). `is_sorted([5, 5, 9])`, `L[1:]`, one item (`[9]`). `find_max([7, 2])`, `L[1:]`, one item (`[2]`). `moves(4)`, `n - 1`, `0`. `mystery("anana")`, `s[1:]`, `""`. `hanoi(2, "A", "C", "B")` *and* `hanoi(2, "B", "A", "C")`: two smaller copies, `n - 1` both times, 0 disks.

C2. `""` (even length: both ends go, down to nothing); `"c"` (odd length: one letter is left); `"d"` (`reverse` stops at one letter); `""` (`stutter` stops only at the empty string); `7` (it is already the base case: no smaller call); `[3]`.

C3. `hanoi` (always, when $n > 0$) and `nested_sum` (when the first item is a list: one call for the rest, one for the list inside).

D1. `[]`, `1` (not 0: with 0, every answer is 0). `[]`, `""`. `""`, `0`. A list of **one** number `[x]`, `x`. `""` or one letter, the string itself. `[]`, `True`. `[]`, `False`. One digit (`n < 10`), `1`. `n = 0`, `[]`. 0 disks, `0`.

D2. The biggest number: an empty list has no biggest number. So `find_max` starts at one number.

D3. `all_positive`: `True`. `any_negative`: `False`. Swapped: `False` and `True` — wrong. Every list ends at `[]`, so `all_positive` now says `False` for every list and `any_negative` says `True` for every list. The base answer must not change the result.

E. 2: `count_digits(9021)`, 4, `4 + 1 = 5`. 3: `"nohty"`, `"nohty" + "p" = "nohtyp"`. 4: `stutter("og")`, `"oogg"`, `"dd" + "oogg" = "ddoogg"`. 5: 2, `1 + 2 = 3` ('i' is a vowel). 6: `mystery("ead")`, 2, 2 ('b' is not a vowel: add nothing). 7: 11; 4 vs 11 → 11. 8: `find_max([11, 6, 2])`, 11; 20 vs 11 → 20. 9: `is_palindrome("efe")`, `True`; r = r, so `True`. 10: **no smaller job**: r ≠ t, so `False` at once. 11: `False`; $3 \leq 8$, but the rest is not sorted: `False`. 12: `2 * 511 + 1 = 1023`. 13: `4950 + 100 = 5050`.

E1. The smaller call is the same function on a smaller list. If the base case and the extend step are right, it gives the right answer. That is the leap of faith.

E2. `is_sorted`: if the first two are in the wrong order, `False` at once (`is_sorted([9, 1, 2, 3])`).

F. 2; 3. `[2]`; `[4, 2]` (4 is even: put it in front). 397, 9; 9 (the cut digit 2 is smaller). `double_all([1, 4])` → `[2, 8]`; `[6, 2, 8]`. `"ab"`; `"ab"` (the first letter is a space: add nothing). 512; 1024. `count_big([5, 30, 11])` → 2; 3. `"bccb"`; `"abccba"`.

F1. `[2 * L[0]] + double_all(L[1:])`; `2 * power2(n - 1)`; `s[0] + mirror(s[1:]) + s[0]`; `1 + count_e(s[1:])`.

G. (a) `"abc"` (b) `"cba"` (c) `"aabbcc"` (d) `"ccbbaa"` (e) `"abccba"` (f) `"***"` (g) `""` — every letter is dropped, nothing is ever added (h) `"a-b-c-"` (i) `"cba"` (j) `"abc"` (k) `"a"` — no smaller call at all (l) `"ac"` — every other letter.

G1. (a) and (j); (b) and (i); (c); (k).

G2. (a) `[1, 2, 3]` (b) `[3, 2, 1]` (c) `[10, 20, 30]` (d) `[1, 1, 2, 2, 3, 3]` (e) **TypeError**: the last call tries `3 + []`, a number plus a list. The kept piece must be a list too: `[L[0]]`. (f) `[]`.

H. H1: `""`, `"tt"`, `"aatt"`, `"ccaatt"`. H2: `[]`, `[1]`, `[1, 2]`, `[1, 2, 3]`: the row above is at the **front**. H3: 9, 12, 17, 18: the row above is the number with the last digit cut off. H4: `[]`, `[8]`, `[8]`, `[6, 8]`, `[6, 8]`: an odd first number adds nothing. H5: `s[0] + s[0]`; `[n]`; `n // 10` and `n % 10`; `[L[0]]`, `evens(L[1:])`.

I1. `digit_sum(903) + 5`; `digit_sum(90) + 3 + 5`; `digit_sum(9) + 0 + 3 + 5`; `9 + 0 + 3 + 5 = 17`. The 0 is a digit too.

I2. `reverse("tar") + "s"`; `reverse("ar") + "t" + "s"`; `reverse("r") + "a" + "t" + "s"`; `"r" + "a" + "t" + "s" = "rats"`.

I3. Down: `moves(2)`, `moves(1)`, `moves(0)`. Up, from the bottom: `0`; `2 * 0 + 1 = 1`; `2 * 1 + 1 = 3`; `2 * 3 + 1 = 7`; `2 * 7 + 1 = 15`. The `moves(0)` row is worked out first (the base case), `moves(4)` last.

I4. `"level"`: l = l → `"eve"`: e = e → `"v"`: one letter → `True`. `"noon"`: n = n → `"oo"`: o = o → `""`: no letters → `True`. `"abcdba"`: a = a → `"bcdb"`: b = b → `"cd"`: c ≠ d → `False`, two levels deep.

I5. Down: `[9, 2, 6]`, `[2, 6]`, `[6]` → base: 6. Up: 2 vs 6 → 6; 9 vs 6 → 9; 4 vs 9 → 9. Answer **9**.

I6. `[1, 3, 3, 2, 5]`: $1 \leq 3$ → `[3, 3, 2, 5]`: $3 \leq 3$ → `[3, 2, 5]`: $3 > 2$ → `False`. **No**: it stops before the 5.

J. Left: **5** (one per digit); **1** (base case at once); **4** (star, tar, ar, r: `reverse` stops at one letter); **1**; **5** (star, tar, ar, r, and `""`); **7** (six letters, then `""`); **4** (racecar, aceca, cec, e). Right: **4** (abccba, bccb, cc, `""`); **2** (a = a, then b ≠ f); **3** (it stops at $3 > 0$); **5** (one per number); **6** (5, 4, 3, 2, 1, 0); **7** (one with 2 disks, two with 1, four with 0); **6** (`[1, [2, 3]]`, `[[2, 3]]`, `[]`, `[2, 3]`, `[3]`, `[]`).

J1. Digit, letter or item (one call per piece cut off). **Two** (it drops both ends). `is_palindrome` (ends that differ) and `is_sorted` (a pair in the wrong order). `find_max` cannot: it must see every number.

K. Left: `0` and `1`; `1`; `-25` (it is < 10, so it comes straight back: the design is for $n \geq 0$); `1` (0 has one digit); `1` — wrong, -512 is also < 10; `""` and `"a"`; `"noon"`; `"b a"` (a space is a letter too); **TypeError**: the last step is `[3] + 2`, a list plus a number; **TypeError**: a number has no `len`; `"aabb "` (the space is doubled too); **IndexError**: a list is never equal to `""`, so `[][0]` is tried; `True` (0 letters). Right: `False` ('R' is not 'r'); `False`: n = n, u = u, r = r, then `"ses "` compares 's' with the space; `True` — it works on lists too; `5`; `-1`; 'o' (it works on strings too: 'o' comes last in the alphabet); **RecursionError**: `len([])` is never 1, and `[][1:]` is `[]` again; `True` (`3 > 3` is false); `True`; `0` (capitals are not in `"aeiou"`); `0` (no vowels); **RecursionError** ($-1, -2, \ldots$ never meets 0); **TypeError**: `"a" + 0`.

L. L1: `"bnn"`. It removes the vowels. L2: `9`. It gives the biggest digit (`find_max` for digits, shrinking with `// 10`). L3: `2`. It counts the negative numbers (0 is not negative). L4: `[3, 2, 1]`. It reverses a list, the way `reverse` does a string. L5: `True`. It tells whether `x` is in `L`, like `x in L`. L6: `3`. It counts the odd digits (`n % 2` is 1 for an odd digit).

M1. `"cat"`, unchanged: the first letter goes back in **front**. Step E. Fix: `return reverse(s[1:]) + s[0]`.

M2. `49` ($2 + 47$). `11`: right, by luck — with two digits, `//` and `%` both give one digit. Step I: `rest` must be `n // 10` and `last` must be `n % 10`. Test with 3 or more digits.

M3. `"ccat"`. `"aa"` — right, by luck. Step S: the smaller job is never solved. Fix: `return double + stutter(s[1:])`.

M4. `2`; `4` (right, by luck). It finds the **smallest** number. Step E: "me or the best of the rest" must keep the bigger one. Fix: `>`.

M5. `0` and `0`: twice 0 is 0, all the way up. It forgot the **one move of the biggest disk**. Step E. Fix: `2 * moves(n - 1) + 1`.

M6. `1`. `None`: 'h' is not a vowel, and that case has no `return`. **TypeError**: the 'a' call tries `1 + None`. Step E: the "not a vowel" case was forgotten. Fix: add `return rest` as line 7.

M7. `3`. **TypeError**: `[1] + 2` adds a list and a number. Step S: the list inside is never solved (added up). Fix: `return nested_sum(first) + rest`.

M8. Right on `[1, 2, 3, 4]` (`True`), `[4, 3, 2, 1]` and `[1, 2, 4, 3]` (`False`). Wrong on `[1, 3, 2, 4]`: it says `True`. Dropping two items skips the pair in the middle, `L[1]` and `L[2]`. Step I. Fix: `L[1:]`.

M9. `True`; `False`; **IndexError**; **IndexError**. An even length shrinks to `""`, which this base case misses, and `""[0]` crashes. Odd lengths stop at one letter, so they hide the bug. Step W. Fix: `len(s) <= 1`.

M10. (a) disk 2 A→C, disk 1 A→B, disk 1 B→C. The **first** move is impossible: disk 1 is still on top of disk 2. The print must come *between* the calls. (b) disk 1 A→B, disk 2 A→C, disk 1 A→C. The **last** move is impossible: peg A is empty (disk 1 is on B). The second call must start from the `spare`.

N. 1: `[2, 3]` (gives 0, not 6) and `[]` (0, not 1). `[5, 0, 9]` and `[0]` are lucky: their answer *is* 0. 2: only `[-3, -8]` (gives 0, a number not in the list). `[0, -1]` is lucky: 0 is the right answer. 3: only `0` (gives 0, not 1). 4: `"abba"` and `"aa"` (**IndexError**: even lengths reach `""`). `"ab"` is lucky: a ≠ b answers `False` before the base case. 5: only `""` (**IndexError**). 6: **none** — this is not a bug. It swaps the two ends and reverses the middle: a correct second design.

N1. The edge inputs: `0`, the empty string `""`, the empty list `[]`, and an even length (for functions that drop two letters). Also negatives, one letter or one item, and an input with 3 or more digits.

O1. Disk 1 from A to **B**; disk 2 from A to C; disk 1 from B to C (the spare is B). `Move disk 1 from X to Z`. Nothing at all: 0 disks, nothing to do.

O2. 1 A→B, 2 A→C, 1 B→C, **3 A→B**, 1 C→A, 2 C→B, 1 A→B. The slides' moves with B and C swapped.

O3. (a) and (c). (b) has disk 3 on disk 1, and (d) has disk 2 on disk 1.

O4. After move 4 (disk 3 A→C): A empty, B holds 2 with 1 on top, C holds 3. After move 6 (disk 1 B→A, then disk 2 B→C): A holds 1, B empty, C holds 3 with 2 on top. One move later, all three are on C.

O5. **4**, **2**, **1** ($4+2+1=7$). **8**. Move **4** and move **8**: right after a tower of $n-1$ is out of the way ($7+1$).

P1. `hanoi(2, "A", "C", "B")` and `hanoi(2, "B", "A", "C")`. Then `hanoi(1, "A", "B", "C")` and `hanoi(1, "C", "A", "B")`. The `n` always shrinks by one; the three pegs swap places.

P2. The top $n-1$ disks must go to the spare peg, out of the way. So, for that smaller tower, the spare is the goal.

P3. `Move disk 1 from A to B` (with an even number of disks, disk 1 goes to the spare first). `Move disk 4 from A to C` (the middle move: 7 moves, then the big one). `Move disk 1 from B to C`.

P4. **7** (the same moves). **RecursionError**: from 0, n goes $-1, -2, \ldots$ and never meets 1. The lecture's base case, 0 disks, is safer.

Q1. **1, 3, 7, 15, 31, 63, 127, 255**, and **1023** for 10 (9 gives 511). Each one is $2^n - 1$.

Q2. 1023 s: about **17 minutes**. 1,048,575 s: about **12 days**. 1,073,741,823 s: about **34 years**. Ten more disks: about 1000 times longer.

Q3. **16** disks: 65,535 moves fit in 86,400 seconds; 17 disks need 131,071.

Q4. **3** (itself and two calls with 0 disks). **7** ($1+3+3$). **15** ($1+7+7$). Twice the calls, plus one, again.

Q5. `moves` makes one call per disk (65 calls) and only *counts*. `hanoi` calls itself twice and prints *every* move: $2^{64}-1$ of them.

R1. `3`, `[2, 3]`; `[5, 6]`, `5`; `False` (1 is a number); `True`; `21`.

R2. **2** boxes: `[1, 2]` and `[3, [4, [5]]]`. `nested_sum` is **15**.

R3. `0`; `10` (recursion does not care how deep); `0` (one box, an empty list); `0`; `7`; **TypeError**: `5` is not a list, so `L[0]` fails.

R4. `[1, [2, 3]]`, `[[2, 3]]`, `[]`, then (for the first item of `[[2, 3]]`) `[2, 3]`, `[3]`, `[]`: 6 calls.

R5. `4`; `2`. It counts the **numbers**, at any depth (each number adds 1, not itself).

S1. `sum_to(5000)`, `reverse("a" * 2000)`, `find_max([])` (it never shrinks) and `mystery("a" * 3000)`. `is_palindrome` drops **two** letters a call (750 calls). `10 ** 500` has 501 digits (501 calls). `"abab...ab"` gives `False` on the first call (a vs b). `moves(64)` is only 65 calls deep.

S2. (a) **L**: simple adding, and far too deep for recursion. (b) **R**: shapes inside shapes. (c) **R**: the towers. (d) **L**: 3000 calls is too deep (or just `len`). (e) **R**: folders inside folders, like lists inside lists. (f) **L**: a million calls deep is far over the limit.

T. `count_a`: **W**: `""` has 0. **I**: drop the first letter. **S**: `count_a(s[1:])`. **E**: add 1 if `s[0]` is 'a'. `product`: **W**: `[]` gives **1**, not 0 (with 0, every answer would be 0). **I**: drop the first number. **S**: `product(L[1:])`. **E**: multiply by `L[0]`. `all_even`: **W**: `[]` gives `True`. **I**: drop the first number. **S**: `all_even(L[1:])`. **E**: if `L[0]` is odd, `False` at once; if not, the smaller answer. `last`: **W**: one item: it is the last one. **I**: drop the first item. **S**: `last(L[1:])`. **E**: nothing to add — the smaller answer *is* the answer.

U1. Tick the first and the last. `len(s) == 1`: `""` goes wrong (`RecursionError`: `""[1:]` is `""` again). `return ""` for one letter: `"ab"` gives `"a"` (the last letter is lost).

U2. Tick only the first. `return 0`: `[-3, -8]` gives 0. `return L`: `[3, 7]` gives a **TypeError** (`3 > [7]`: a number and a list), and `[5]` gives `[5]`, a list. `return L[0]` on `[]`: **IndexError** for every list.

U3. Tick the first and the last (`n <= 9` is the same test). `n == 0: return 0`: only `0` goes wrong (0, not 1). `n < 10: return 0`: every input is one short (`7` gives 0).

U4. Tick only the first. `len(L) == 1`: `[]` gives an **IndexError**. `L == []`: a list with one item, like `[5]`, has no `L[1]`: **IndexError** (every sorted list ends there). `return False`: `[1, 2]` gives `False`.

V. `count_digits`: Q, 4. `digit_sum`: T, 8. `stutter`: R, 5. `reverse`: U, 1. `product`: Q, 7. `moves`: P, 2. `upto`: S, 6. `total`: P, 3. (Q and P are used twice.)

W. (a) **same**: the last letter goes in front, and the rest is reversed. (b) **same**: "the best of the rest" now looks at the front part, and the last number is "me". (c) **same**: `and` gives `False` as soon as the ends differ. (d) **different**: `upto(3)` gives `[3, 2, 1]` (Lecture 21's `countdown_list`). (e) **same**: from 0, the base case gives 0, and `digit_sum(7)` is `digit_sum(0) + 7 = 7`. One more call per number, the same answers. (f) **different**: `stutter("ab")` gives `"abba"`, not `"aabb"`.

X1. `count_char`: `s`; `c` stays the same. `contains`: `L`; `x` stays the same. `repeat`: `n`; `s` stays the same. `hanoi`: `n`; the three pegs do not shrink, but they **swap places** in each call.

X2. `3`; `0` (capital 'A' is not 'a'); `True`; `False`. `"abababab"`; `""`; `""` (five empty strings joined); **RecursionError** (`n` goes $-1, -2, \ldots$ and never meets 0).

X3. **2**: it stops as soon as it finds 8. **4**: 4, 8, 1, then `[]`, where it answers `False`.

Y1. `0`; `s[1:]`; `s[0]`; `rest + 1`. `1`; `L[0]`; `L[1:]`. `count_a("Aa")` is `1` ('A' is not 'a'); `product([5, 0, 9])` is `0`.

Y2. `True`; `1`; `all_even(L[1:])`. `1`; `L[0]`; `last(L[1:])`. `False` (5 is odd); `True`; `7`; **RecursionError**: `[]` never has length 1 (an empty list has no last item).

Y3. `0`; `[]`; `n - 1`; `[n]`. **RecursionError**: from -2, `n` never meets 0. (With `n <= 0` in the base case, `upto(-2)` gives `[]`.)

Z1. W: one digit: give it back. I: `n // 10`. S: `digit_product(n // 10)`. E: multiply by `n % 10`. Code: `if n < 10: return n`, then `return digit_product(n // 10) * (n % 10)`. 305 gives `0`.

Z2. W: `n == 1` gives `True`. E first: if `n % 2 == 1`, `False` at once. I: `n // 2`. S: `return is_power_of_two(n // 2)`. $12 \to 6 \to 3$ (odd): `False`.

Z3. W: `[]` gives `[]`. I: first item and the rest. S: `rest = flatten(L[1:])`. E: if the first item is a list, `flatten(first) + rest`; if not, `[first] + rest`. `flatten([[], [[]]])` is `[]`.

Z4. W: `[]` gives 1. I: first item and the rest. S: `rest = depth(L[1:])`. E: if the first item is a list, `inner = 1 + depth(first)`; return the bigger of `inner` and `rest`. `depth([[]])` is 2.