

Recursion, Josephus and Search Tools

Implement recursive functions, check a Josephus recurrence against a simulation, and process text with shell pipelines. Solo assignment 1 starts this week with ordinary and adjacent-peg Hanoi.

Due: Homework 3 · Tue 17 Nov, 23:59

Goals and submission routes

- State the input contract, base cases and progress of a recursive function.
- Compare recursive and iterative digit sums without changing the recursion limit.
- Check an algorithm using an independently implemented simulation.
- Search source files and combine text-processing commands.

Parts A, B and C are open together. Allow roughly 40, 35 and 15 minutes. Part A supplies 60% of this week's homework mark and Part B 40%; Part C records completion and is examinable. The judge reports raw problem points, not semester percentages. TAs apply the course's marking scheme when consolidating results.

Level 0

Do not use AI to generate or autocomplete code for this lab. Solo assignment 1 remains **Level 0 throughout weeks 7–8**, even after the general policy changes following Lecture 8.

Use [Homework 3](#) for this lab and [Solo assignment 1](#) for Hanoi. Their submission pages calculate the deadlines from the shared course calendar, in Beijing time (GMT+8). A lab's teaching week is not its submission deadline.

Part A · Recursive functions

Each of the six coding tasks is worth 4 raw points: 24 automatic points for Part A. The original five tasks retain their 4-point values. Adding task 7-5 adds raw homework points, not a new 4% semester component. Submit the observations for 7-5 on its separate text answer card.

The requirement to use recursion is also checked by code review. Syntax checks enforce the stated restrictions; they cannot establish that an implementation is recursive, terminates or is correct. Graded examples stay well below the usual recursion limit. Run the large-integer experiment separately on your own machine, without increasing that limit.

7-1 Count decimal digits

Implement `count_digits(n: int) -> int` recursively for a non-negative integer. Graded inputs have at most 16 decimal digits. This task's submitted file must contain no `for`, `while`, comprehensions or generator expressions.

```
count_digits(0)      → 1
count_digits(7)      → 1
count_digits(12345) → 5
```

The representation of zero

`0` has one decimal digit. Check what your base case returns before building the recursive step.

7-2 Reverse a string

Implement `reverse_string(s: str) -> str` recursively. In this task's submitted file, do not use `reversed()`, negative-step slicing (including `[::-1]`), `for`, `while`, comprehensions or generator expressions. Forward slices are allowed. Graded strings have length at most 60.

```
reverse_string("")      → ""
reverse_string("a")     → "a"
reverse_string("hello") → "olleh"
```

7-3 Test for a palindrome

Implement `is_palindrome(s: str) -> bool` recursively. Compare characters exactly: do not discard spaces or punctuation, or change case.

```
is_palindrome("")      → True
is_palindrome("a")     → True
is_palindrome("abba")  → True
is_palindrome("abc")   → False
```

Reduce the string

Matching endpoints leave a smaller substring to check. Decide separately what an empty string and a one-character string mean.

7-4 Flatten nested lists

Implement `flatten(items: list) -> list[int]`. Return a **new** list of integer elements in left-to-right order. Inputs contain only integers and lists and are finite, with no cyclic references. Do not change the outer list or any nested list. Graded nesting depth is at most 6; arbitrary finite depth is still subject to the interpreter's recursion limit.

```
flatten([1, [2, [3, [4]]]]) → [1, 2, 3, 4]
flatten([])                  → []
flatten([[], [1], [[2], [3]]]) → [1, 2, 3]
```

`isinstance(x, list)` is a type check: it returns `True` for a list and `False` for an integer. You may loop over the current list and recurse when an element is another list.

7-5 Digit sums: two implementations

Put both functions in one submitted source file. Both accept non-negative integers.

```
def sum_digits_rec(n: int) -> int:
    """Return the decimal digit sum recursively; the sum for 0 is 0."""
    pass

def sum_digits_loop(n: int) -> int:
    """Return the decimal digit sum using a loop; the sum for 0 is 0."""
    pass
```

```
sum_digits_rec(0)      → 0
sum_digits_loop(0)     → 0
sum_digits_rec(12034)  → 10
sum_digits_loop(12034) → 10
```

The recursive implementation must not use loops, comprehensions or generator expressions. The iterative implementation must not call the recursive function or a recursive helper. Automatic tests use at most 16 digits. Submit both functions to the “Digit sums: recursion and iteration” coding task in Homework 3.

Next, run each version locally on `n = 10**1000`. This number has **1001 digits**, not 1000. Keep the default recursion limit. On the “7-5 Recursion and iteration notes” answer card, give:

1. A comparison of code length and readability, referring to your implementations.
2. Your Python version, the value of `sys.getrecursionlimit()`, and each function’s actual result or exception type. Explain the depth produced by repeated division by 10, why a finite computation can still raise `RecursionError`, and the iterative result.

Different environments can have different limits and observations. A `RecursionError` indicates the interpreter’s recursion restriction; it does not establish that physical memory has run out. Do not increase the limit to make this experiment pass.

7-6 Count coin combinations

Implement `count_ways(amount: int, coins: list[int]) -> int` recursively. The amount is a non-negative integer. Denominations are **distinct positive integers**, in any order; each can be used any number of times. Count combinations, not ordered arrangements. Do not modify the denomination list. Graded amounts are at most 40, with at most four denominations.

```
count_ways(5, [1, 2, 5]) → 4
count_ways(0, [1, 2, 5]) → 1
count_ways(3, [2])      → 0
count_ways(0, [])       → 1
count_ways(5, [])       → 0
```

The four combinations for 5 are `5`, `2+2+1`, `2+1+1+1` and `1+1+1+1+1`. In particular, `1+2` and `2+1` are the same combination.

Include or exclude a denomination

Including the current coin reduces the amount while retaining that denomination. Excluding it leaves the amount unchanged but removes that denomination from further choices. Add the two counts.

Amount zero has one combination: choose nothing. A recursive subproblem may have a negative amount and must return zero. A positive amount with no denominations also has zero combinations.

Part B · The Josephus problem

People numbered `1, 2, ..., n` form a circle. Counting starts at person 1. Every `k`th person leaves, and counting restarts at 1 with the next person. Stop when one person remains. Support `1 ≤ n ≤ 120` and `1 ≤ k ≤ 1000`.

B-1 Simulate the circle

Implement `josephus_sim(n: int, k: int) -> tuple[list[int], int]`. Return `(elimination_order, survivor)`. The survivor is not included in the elimination list.

```
josephus_sim(7, 3) -> ([3, 6, 2, 7, 5, 1], 4)
josephus_sim(1, 9) -> ([], 1)
josephus_sim(5, 1) -> ([1, 2, 3, 4], 5)
```

Track the index at which counting starts. After a deletion, which index represents the next person? Record the list and deletion index at every round for `n = 7, k = 3`.

B-2 Find the survivor recursively

Implement `josephus_rec(n: int, k: int) -> int`. Use recursion and return a **one-based** person number.

The recurrence is simplest with a zero-based survivor index `J`:

```
J(1, k) = 0
J(n, k) = (J(n - 1, k) + k) % n    for n > 1
```

Explain how renumbering the smaller circle maps its survivor back to the original circle. Identify the base case and the decreasing measure, then convert to the public one-based result.

B-3 Cross-check the results

Submit both functions in a single file to "Josephus: simulation and recurrence" (4 raw automatic points). Locally compare the survivor returned by each method for every pair `n = 1..30, k = 1..10`.

On the "Josephus notes" answer card, include your comparison code and a summary of the results. Also show elimination orders and survivors for `(1, 9)`, `(5, 1)`, `(7, 3)` and `(10, 20)`. One memorized survivor is not an adequate check.

B-4 Account for the work

What data does the simulation retain? What data does a survivor-only recurrence avoid storing, and how deep is its call chain? Discuss the costs of list deletion and recursive calls using your code. Large timing experiments and increased recursion limits are not required.

Part C · Search and pipelines

Use a macOS terminal, a Linux shell or Windows WSL. These examples are not PowerShell commands. If WSL is unavailable, tell a TA during the lab. Work only inside your own practice directory.

The commands before the exercise

- `grep` finds matching lines in file contents. `-n` adds line numbers, `-F` matches literal text, and `-r` searches a directory recursively.
- `find` traverses directories. `-type f` selects regular files; `-name '*.py'` filters filenames. Quote the pattern so that the shell does not expand it first.
- `wc -l` counts newline characters. Our example has a newline after every line, so this is also the line count.
- `sort` sorts lines. `uniq` removes **adjacent** duplicate lines, not all duplicates anywhere in a file.
- `|` connects a command's standard output to the next command's standard input.

Create `words.txt` with an editor, including a newline after the last line:

```
pear
apple
pear
banana
apple
```

Try the commands separately before combining them:

```
grep -n -F 'pear' words.txt
wc -l < words.txt
sort words.txt
uniq words.txt
sort words.txt | uniq
sort words.txt | uniq | wc -l
find . -type f -name '*.py'
grep -rn -F 'print(' .
```

`find` searches for files; `grep` searches their contents. Direct `uniq words.txt` leaves the non-adjacent repetitions, while the sorted pipeline ends with a count of 3. Counting lines from `find` is not an accurate file count if a filename contains a newline; this exercise uses ordinary filenames without newlines.

C-1 Search and count

1. Paste the first six commands above with their actual output. Explain why sorting changes the effect of `uniq`.
2. Use `find` and a pipeline to count `.py` regular files in your Lab 1–7 source directory. State the directory you searched.
3. Search that directory for the literal text `print(` with `grep -rn -F`. Include a short output excerpt and explain why a matching line need not be an executed `print` call.

Submit commands, output and explanations on Homework 3's "Search and pipelines" text answer card. This records completion only, with no automatic points. No terminal screenshot or additional attachment upload is needed.

Solo assignment 1 · Ordinary and adjacent-peg Hanoi

There are **three solo assignments, each worth 5%** of the semester grade, totaling 15%. Work on this one in weeks 7–8. The [solo submission page](#) displays its calendar-derived deadline. The 100-point breakdown below is internal to this assignment, not an additional semester weighting.

7-7 Ordinary Hanoi

Pegs are named **A**, **B** and **C**. Support integer inputs $0 \leq n \leq 8$. Disk numbers are $1..n$, with larger numbers denoting larger disks. Initially A holds all disks, ordered $n, \dots, 2, 1$ from bottom to top. The target is the same tower on C.

A move transfers one **top** disk to a different peg. A larger disk may not be placed on a smaller one. In ordinary Hanoi, any pair of different pegs can be used.

```
def hanoi_moves(n: int) -> list[tuple[int, str, str]]:
    """Return an optimal ordinary-Hanoi sequence from A to C."""
    pass
```

Return a list of $(\text{disk_number}, \text{source_peg}, \text{target_peg})$ tuples, not printed instructions or a move count. For $n = 0$, return a fresh empty list. Every call returns an independent list. The moves must be legal, reach the target and be optimal: exactly $2^{*}n - 1$ moves.

```
hanoi_moves(1) -> [(1, "A", "C")]
hanoi_moves(2) -> [(1, "A", "B"), (2, "A", "C"), (1, "B", "C")]
```

7-8 Adjacent-peg Hanoi

Peg positions are fixed as **A–B–C**. Allowed moves are **A↔B** and **B↔C**; neither **A→C** nor **C→A** is a single legal move. The disk numbering, input range, initial state and target are unchanged.

```
def adjacent_hanoi_moves(n: int) -> list[tuple[int, str, str]]:
    """Return an optimal A-to-C sequence using adjacent pegs only."""
    pass
```

Return the full sequence, with a fresh empty list for $n = 0$. The sequence must obey the size and adjacency rules, finish on C, and use the minimum $3^{*}n - 1$ moves.

```
adjacent_hanoi_moves(1) → [(1, "A", "B"), (1, "B", "C")]
```

The largest disk must pass through B. Work out where all smaller disks have to be before each of its two moves. Explain the three end-to-end transfers of $n-1$ disks and the two largest-disk moves giving $T(0)=0$, $T(n)=3T(n-1)+2$. Establish the matching lower bound and solve for $T(n)=3*n-1$; quoting that formula alone is insufficient.

7-9 Validate a sequence

```
def is_valid_sequence(
    n: int,
    moves: list[tuple[int, str, str]],
    adjacent: bool = False,
) -> bool:
    """Check move legality and the final state, not optimality."""
    pass
```

Simulate the peg states without modifying `moves`. With `adjacent=False`, check ordinary rules; with `adjacent=True`, also enforce adjacency. Return `True` only if:

- Every move is a three-item tuple. Its disk number is an integer in $1..n$, not a Boolean. Peg names are A, B or C, with distinct source and target.
- The named disk is the source peg's current top disk. The destination is empty or has a larger top disk.
- No direct endpoint-to-endpoint move occurs in adjacent mode.
- The final tower is on C in size order, with A and B empty.

Return `False` if any condition fails. Only `n` in the stated range and Boolean `adjacent` are required. `moves` is a list, but its elements may be malformed. With `n = 0` and an empty list, return `True`.

Legality is not optimality. For one disk, `[(1, "A", "B"), (1, "B", "A"), (1, "A", "C")]` is a legal completed ordinary transfer and must return `True`, despite using too many moves. The first two generation tasks check optimal move counts separately; the validator must not substitute a length check for simulation.

7-10 Recurrences and test evidence

Use the “Hanoi explanations” answer card for four responses:

1. Explain the lower bound $2T(n-1)+1$ for ordinary Hanoi, a construction attaining it, the base case and the resulting minimum.
2. Describe the three smaller-tower transfers and two largest-disk moves in adjacent Hanoi. Explain why the lower bound and construction agree, and derive $3**n-1$ from the recurrence.
3. Report local counts, legality checks and final-state checks for both methods at $n = 0..5$. Include at least four illegal-sequence tests: an invalid disk number, a non-top disk move, a larger-on-smaller move and a direct A→C move in adjacent mode. Also test a legal but non-optimal sequence and explain the validator’s result.
4. Explain function decomposition, type annotations, and the difference between call depth and output length. Discuss time and storage from small experiments; do not try to construct enormous move lists.

Optional exploration: investigate divide-and-conquer tiling with L-shaped trominoes and discuss your approach with a TA. This release has no automatic grading or bonus submission slot for that extension; it does not add a semester component.

Marks and file handling

Component	Internal raw points	Assessment
Complete, legal, optimal ordinary sequence	30	Automatic tests; recursive design reviewed separately
Complete, legal, optimal adjacent sequence	40	Automatic tests; recursive design reviewed separately
Validator: rules, final state and unchanged input	20	Automatic tests
Recurrence arguments, test evidence and code quality	10	Text answer card, marked by a TA

Write all three functions in **one** `solution.py`. There is no `render` requirement and no visualization deliverable. Submit the entire same source to each of the three solo coding cards, then submit the explanation card. The existing judge accepts one source per card, not a multi-file upload or archive.

All three skeletons and public tests use module name `hanoi_submission`. To run them locally, copy `solution.py` to `hanoi_submission.py` and keep each task's public tests in a separate directory so their filenames do not overwrite one another.

Full automatic points mean the tests passed; they do not replace the recursion review or minimum-move argument. A TA may ask you to explain your code. The whole assignment remains Level 0.

Submission checklist

For Homework 3, submit six Part A coding tasks, the 7-5 notes, both Josephus functions and their notes, and the Part C card. For Solo assignment 1, submit the full shared source on three coding cards and complete the Hanoi explanations. Once every task in an assignment has been submitted, download its evaluation report and submit that report to Canvas. Reports retain both source and text answers; no new attachment workflow is required.

- ☐ Zero, empty strings, empty lists and empty denominations are covered.
- ☐ Flattening and coin counting leave their inputs unchanged; signatures match the skeletons.
- ☐ The 1001-digit experiment includes the actual environment and observations.
- ☐ Josephus counting positions, index conversion and cross-checks are explained.
- ☐ The shell card contains actual commands and output.
- ☐ Hanoi generators return move lists; the validator distinguishes legality from optimality.
- ☐ Solo code and explanations agree, and no AI-generated or autocompleted code was used.