1. List Processing & Control Flow

Problem 1 — Filter Increasing Adjacent Pairs

Write a function `adjacent_increasing(nums)` that:

- takes a list of integers
- returns a **new list containing only the integers that are strictly larger than the number immediately before them** in the list.

Example:

```
adjacent_increasing([5, 7, 3, 9, 9, 12])
→ [7, 9, 12]
```

---

2. Dictionaries & Counting

Problem 2 — Character Frequency Dictionary

Write a function `char_freq(s)` that:

- takes a string
- returns a dictionary mapping each character → the number of times it appears

Example:

```
char_freq("banana")
→ {'b': 1, 'a': 3, 'n': 2}
```

3. Sets & Nested Structures

Problem 3 — Unique Coordinates

You are given a list of coordinate pairs (tuples), possibly with duplicates.

Write a function `unique_coords(coords)` that:

- takes a list like `[(1,2), (3,4), (1,2), (3,4), (5,6)]`
- returns a **set** of all unique coordinates.

Example:

```
unique_coords([(1,2), (3,4), (3,4), (1,2)])
→ {(1,2), (3,4)}
```

4. Recursion

Problem 4 — Recursive Digit Sum

Write a **recursive** function `digit_sum(n)` that:

- takes a non-negative integer `n`

- returns the sum of its digits
- **must use recursion**, not loops

Example:

```
digit_sum(5029)
→ 16
```

---

5. Iterators & Generators

Problem 5 — Generator of Running Totals

Write a generator function `running_total(nums)` that:

- yields the cumulative total as you iterate through the list

Example:

```
g = running_total([2, 5, 3])
next(g) → 2
next(g) → 7
next(g) → 10
```

---

6. Nested Data Structures + Logic

Problem 6 — Students Who Passed All Courses

You are given a dictionary mapping student names → list of their grades in multiple courses:

```
{
    "Alice": [70, 85, 90],
    "Bob": [50, 40, 62],
    "Cara": [88, 72, 91]
}
```

Write a function `students_passing(data)` that:

- returns a **list of names** of students who scored **≥ 60 in every course**
- order of returned names does NOT matter

Example:

```
students_passing(data)
→ ["Alice", "Cara"]
```

PART 2: MORE PROBLEMS

Topic 1 — Basic Python Syntax & Control Flow

Q1.1 — Remove Consecutive Duplicates

Write a function `remove_consecutive_dups(lst)` that returns a new list with all **consecutive duplicate elements removed**, but keeps non-consecutive duplicates.

Example:
`[1,1,2,2,2,3,1,1] → [1,2,3,1]`

Q1.2 — Count Values Below Average

Write `count_below_avg(nums)` that returns how many integers in the list are **strictly below the average** of the list.

Q1.3 — First Index of Decline

Write `first_decline(nums)` that returns the **first index** `i` such that `nums[i] < nums[i-1]`.
If no such index exists, return `-1`.

Q1.4 — Flatten Only One Level

Write `flatten_once(nested)` that flattens **only one level** of a list:

`[[1,2], 3, [4], 5] → [1,2,3,4,5]`

Do *not* use recursion.

Q1.5 — Pair Elements Into Tuples

Write `pair_up(lst)` that groups elements into pairs of 2:

`[1,2,3,4,5] → [(1,2), (3,4), (5, None)]`

If the list has an odd number of values, pair the last element with `None`.

Topic 2 — Dictionaries

Q2.1 — Invert a Dictionary

Write `invert_dict(d)` that swaps keys and values **assuming values are unique**.

`{'a': 1, 'b': 2} → {1: 'a', 2: 'b'}`

Q2.2 — Group Words by Length

Write `group_by_length(words)` returning a dictionary:

key = length
value = list of all words with that length

Q2.3 — Sum Dictionary Values with Matching Keys

Write `sum_dicts(d1, d2)` that returns a new dictionary where:

- each key that appears in *either* dictionary appears in the result
- values are summed (missing values treated as 0)

Q2.4 — Most Frequent Value

Write `most_frequent_value(d)` that returns the value that appears most frequently in the dictionary's **values**.
Assume at least 1 value.

Q2.5 — Filter Dictionary by Key Condition

Write `filter_keys(d, fn)` where `fn` is a function taking a key and returning True/False.

Return a **new dictionary** containing only entries where `fn(key)` is True.

Topic 3 — Sets

Q3.1 — Common Characters in All Words

Write `common_chars(words)` that returns the **set of characters** that appear in **every** word in the given list.

Q3.2 — Remove All Values Appearing in Another List

Write `remove_values(lst, forbidden)` that returns a set of all elements in `lst` that are **not** in `forbidden`.

Q3.3 — Unique Sorted Pairs

Write `unique_pairs(lst)` that:

- forms ALL 2-element unordered pairs `(a,b)` where `a ≠ b`

- returns them as a **set of tuples**
- ensure `(a,b)` and `(b,a)` count as the same tuple (store with smaller first)

Example:
`[1,2,3] → {(1,2), (1,3), (2,3)}`

Q3.4 — Symmetric Difference of Lists

Write `list_sym_diff(a, b)` returning the values that appear in **exactly one** of the lists.

Q3.5 — Detect Duplicates Using Only a Set

Write `has_duplicate(lst)` returning True if any element appears more than once, otherwise False.

Topic 4 — Recursion

Q4.1 — Count Occurrences Recursively

Write `count_occ(lst, x)` that counts how many times `x` appears in a list using **recursion** only (no loops).

Q4.2 — Recursive Max of List

Write `rec_max(lst)` that returns the maximum element using recursion only.

Q4.3 — Reverse String Recursively

Write `rev(s)` that returns the reversed version of s using recursion.

Q4.4 — Check if List is Palindrome Recursively

Write `is_pal(lst)` that returns True if the list is a palindrome using recursion.

Q4.5 — Sum of Nested List (Recursive Nesting)

Write `sum_nested(nested)` where elements may be integers or lists containing more integers/lists.

Example:
`[1,[2,[3,4],5],6] → 21`

Topic 5 — Iterators & Generators

Q5.1 — Generator of Even Numbers in a List

Write a generator `evens(lst)` that yields only the even integers from the list.

Q5.2 — Infinite Alternator

Write a generator `alternator(a, b)` that yields:

`a, b, a, b, a, b, ...` forever.

Q5.3 — Generator for All Prefixes of a String

For string `"boat"` yields:

`"b"`, `"bo"`, `"boa"`, `"boat"`

Q5.4 — Chunk Generator

Write `chunks(lst, size)` that yields slices of the list of length `size`.

Example:
`chunks([1,2,3,4,5], 2)` yields:

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

Q5.5 — Enumerate-But-Backwards Generator

Write `reverse_enumerate(lst)` that behaves like built-in `enumerate`, but goes from the end to the start.

Example:
`reverse_enumerate(['a','b','c'])` yields:

```
(2,'c')
(1,'b')
(0,'a')
```