

1) MUTATION - In-place Prefix Sums (no extra list)

def inplace_prefix_sum(nums: List[int]) -> None:

"""

Mutate nums so that nums[i] becomes the max of sum of nums[0..i] and
nums[i...-1](inclusive).

Do this IN PLACE. $O(n)$ time, $O(1)$ extra space.

Constraints:

- Works for empty list and negatives.
- May only read/write nums; no return value.

Examples:

nums = [1, 2, 3, 4] -> [10, 9, 7, 10]

nums = [] -> []

nums = [5] -> [5]

"""

TODO: implement in-place prefix sums

```

# 2) ALIASING - Replicate rows without shared inner lists
def replicate_rows(matrix: List[List[int]], times: int) -> List[List[int]]:
    """
    Return a NEW matrix where each original row is repeated 'times' times
    vertically. Ensure no aliasing: modifying one replicated row must NOT
    affect any other row.

    Constraints:
        - Do NOT mutate input 'matrix'.
        - Guarantee deep independence of produced rows (i.e., no shared inner
        lists).
        - times >= 0.

    Examples:
        matrix = [[1,2],[3,4]], times=2 ->
            [[1,2],[1,2],[3,4],[3,4]]
        If result[0][0] = 99, result[1][0] must remain 1 (no aliasing).
    """
    # TODO: build a new matrix with independent row copies

```

```
# 3) DICTIONARIES - Merge multiple count dicts safely (no input mutation)
def merge_counts(dict: List[Dict[str, int]]) -> Dict[str, int]:
```

```
    """
```

```
    Given a list of count dictionaries (e.g., word -> frequency),
    return a NEW dictionary summing counts per key across all inputs.
    Must not mutate any input dictionaries.
```

```
    Constraints:
```

- Keys are strings; counts are nonnegative ints.
- O(total number of key-value pairs) time.

```
    Examples:
```

```
    dicts = [{"a":2,"b":1}, {"b":3,"c":5}] -> {"a":2,"b":4,"c":5}
```

```
    dicts = [] -> {}
```

```
    """
```

```
    # TODO: aggregate into a new dict without touching inputs
```

```

# 4) TUPLES - Return argmin/argmax with indices
def argminmax(xs: List[int]) -> Tuple[Tuple[int,int], Tuple[int,int]]:
    """
    Return ((min_val, min_index), (max_val, max_index)) as TUPLES.
    If multiple occurrences, take the smallest index.

    Constraints:
        - xs non-empty.
        - Single pass preferred (O(n) time, O(1) extra space).
        - Do NOT sort.

    Examples:
        xs = [5, 2, 9, 2, 7] -> ((2,1), (9,2))
        xs = [3] -> ((3,0), (3,0))
    """
    # TODO: compute min/max values and first indices; return two tuples

```

5) INTEGRATION (Mutation + Aliasing + Dicts + Tuples)

```
def curved_gradebook(
    gradebook: Dict[str, Tuple[int, int]],
    curve: Tuple[int, int]) -> Tuple[Dict[str, Tuple[int, int]], Tuple[float,
float]]:
```

"""

Given:

- gradebook: mapping student->(midterm, final) as TUPLES (immutable),
- curve: (curve_midterm, curve_final) to add to each component,

Return:

(new_gradebook, (avg_midterm, avg_final))

Requirements:

- Do NOT mutate the input 'gradebook' or any of its tuple values.
- new_gradebook must map to NEW tuples reflecting curved scores, each component clamped to [0, 100].
- (avg_midterm, avg_final) are the arithmetic means of curved components.

Examples:

gb = {"alice": (92, 88), "bob": (81, 90)}, curve=(+3, -5)

-> new_gb = {"alice": (95, 83), "bob": (84, 85)}

avgs = ((95+84)/2, (83+85)/2) = (89.5, 84.0)

Edge cases:

- Empty gradebook -> return ({}, (0.0, 0.0))
- Scores after curve must be clamped to [0,100]

"""

TODO: produce a new dict of NEW tuples; compute averages; avoid
mutation/aliasing