Exercise 1. I approached the first task by turning "sum of the preceding n values" into a running total that I will have to update as my code moves through the list. In order to determine precisely what the value at index n should equal if the condition is true, I first initialised a variable that calculates the sum of the previous n values (rolling_sum = sum(lst[:n]). Then I loop from i = n up to the final value in the list. In the example provided in the exercise, for the list [1, 2, 3, 5, 8, 13], and n = 2, the rolling_sum should hold 3 (1 + 2) and at index 2, the value is 3. So, my function essentially checks lst[i] against rolling_sum on each loop to see if they do not match, rather than if they do, as this allows the function to efficiently halt early and return False if the rule has already been violated. However, if there is a match, I use rolling_sum += lst[i] – lst[i-n] to include the latest element that will belong to the following window and eliminate the previous value that we already checked. Without re-summing a slice, this maintains the invariant that rolling_sum equals the sum of the preceding n items prior to each subsequent comparison. The function doesn't require any additional lists or dictionaries; it just uses integers and a basic for loop. This makes the code ideal: minimum operations per element, early exit on failure, and one linear pass that scales cleanly to big inputs.

Exercise 2. Building a simple system that quickly counts and ranks bigrams after it traverses through a dictionary once was the goal of my code for this challenge. Since, collect_bigrams(word) utilises yield, bigrams are generated dynamically without the need for temporary lists, saving memory for lengthy files. The function does this by shifting a 2-character frame one position at a time along the word and yielding each pair. Instead of loading the whole file into memory at once, count_bigram(), reads and handles linuxwords row by row. Each word is lowercased, leading and trailing whitespace removed and for each captured bigram, a simple dictionary is updated with counts[bg] = counts.get(bg, 0) + 1 to ensure constant-time increments. The code also ensures words with less than two characters don't add anything since the loop never executes. Finally, main() displays the results using a straightforward 1-based rank after most_freq_bigrams(counts, k) sorts the dictionary keys by their counts in descending order and provides the top-k (top-25). This keeps the printing functionality distinct from counting. Overall, my approach eliminates additional passes or superfluous structures, goes over the letters in a single pass, simply saves the frequency table, and remains simple to understand and maintain.

Exercise 3. It was initially challenging to determine the most effective strategy for this exercise, but after reviewing the lab exercises, I realised that it all came down to converting the limitations into algebra that enables me to prune the search space. If i was to try every possible combination it would probably require at least four nested loops, the complexity would be quartic (O(n^4)), which is why replacing inner loops with algebra significantly improves scalability. Since it's not possible to have more scorpions than the total animals and each scorpion has eight legs, I begin by restricting the number of scorpions to the minimum of total_animals and total_legs//8. I calculate the left over animals and legs. I do a brief viability test to see if the remaining legs can be produced by the remaining animals (if (2 * anm_left_s <= legs_left_s) and (legs_left_s <= 6 * anm_left_s)). The reasoning behind this line: each of the other creatures must have at least two legs (parrots) and no more than six legs (ladybugs). The remaining legs must consequently fall within this range in order for any solution to be feasible at this point. After that, I loop ladybugs lb with strict constraints (legs_lb_limit = legs_left_s // 6) and (lb_limit = anm_left_s). At this point, only two animals remain. This gives us two simple equations ((p + l = animals_left_s_lb) and (2p + 4l = legs_left_s_lb)) and from these, I can solve algebraically for one lion (l = (legs_left_s_lb – 2 * animals_left_s_lb)/2). From this, it's evident the number must be an integer (as the numerator must be even, can't have half a lion) and between 0 and animals_left_s_lb. That leaves the number of parrots which is equal to animals_left_s_lb – l. Valid tuples are appended as small dictionaries. Overall, this solution shows by incorporating tight bounds to scorpions and ladybugs first, you can work out parrots and lions using just arithmetic, instead of having to loop over them. The use of only two loops makes the code simple, quick and scalable.

Exercise 4. My approach to locating patterns in this task involved creating a purposeful function that scans each straight line in four directions precisely once. To gather results, I save n = len(board) and a matches list. To make bounds checks understandable for myself, I used a tiny is_valid(r,c) function.

The fundamental function, scan_lines_direction(starts, row_step, col_step), uses a straightforward while loop to traverse each line, progressing row += row_step and col += col_step, given a list of initial coordinates and the direction to move each time. The code maintains two bits of information while it's traversing: current_run (the coordinates for that run) and run_type (the symbol being recorded at the moment). I add [row, col] if the next piece matches the previous symbol; if it differs, I finish the prior run (only appending it to matches if its length is >=3) and begin a fresh run with the latest symbol. Before the end of the line, it also checks if the current run is long enough and, if so, stores it. To ensure that each list is covered correctly and without repetition, the final part of this code creates the starting points for rows (all (r,0)), columns (all (0,c)), and both diagonals (top row and left column; top row and right column). For clarity, I express matches as simple lists of [row, col] pairs. Due to its $O(n^2)$ complexity and modest, set number of visits (one per direction) per object, this design is efficient.

Exercise 5. Without loading the entire file into memory at once, I handled ratings.csv row by row, in order, and retained just the data required to determine the earliest rating. I used csv.reader to load the file, omitted the header, and initialised two variables using the first data row: earliest_user_id = first_row[0] (userId is the first column) and earliest_timestamp = int(first_row[-1]) (timestamp is the final column). The logic of the code's only loop is straightforward and iterates over the rest of the rows. But first, it converts the data entries for the timestamp column from strings to integers (t_stamp = int(row[-1])) before beginning comparisons to the current earliest timestamp. The two variables, earliest_timestamp and earliest_user_id, are only updated if it was smaller. I chose this method once again because it is simple to understand and update, eliminates unnecessary passes or data structures, and has a linear in the number of rows.

Exercise 6. The aim of my code was to create functions that distinguish between aggregate and lookup. movieId_title() reads movies.json one line at a time, gathering just the information the solution requires and creating a dictionary titles[id] = title; this prevents loading additional, unnecessary data. Due to this dictionary, the time to later find the title is basically the same (one quick lookup instead of a search through all entries). The same concept can be seen in the next function as movieId_rating() returns each id row by row by opening ratings.csv and reading the headers once to get the movieId. In main(), I maintain two variables (most_id, most_count) to monitor the current leader and one counts dictionary to record the number of ratings each movie gets. On each yielded movieId, I update its count with counts.get(movieId, 0) + 1 and, I update the leader if it surpasses the current leader. Finally, I print the title of the movie with the largest number of ratings by indexing titles. With this method, every file is processed precisely once, just the essential data structures (dicts for O(1) updates/lookups) are utilised, unnecessary checks are avoided, and memory consumption is kept to a minimum using streaming. Essentially, the code is simple to reuse and upkeep due to its tiny, targeted functions.

Exercise 7. Once again, I've used multiple functions to break down this task. title_language() reads movies.json one row at a time and creates a dictionary movie lookup so that language and titles may be resolved by its id in subsequent phases. In similar fashion, iterate_rows() streams ratings.csv and yields (userId, movieId, rating) tuples. To find French-exclusive users, I scan ratings once: add every userId to all_users, and if that user ever rates a non-French film (movies[movieId]['language'] != 'fr'), add them to non_french_users. The set difference all_users - non_french_users gives exactly the users who rated only French films. With that filtered set, highest_rated_movie() makes a second pass over ratings and, for French-exclusive users only, updates three dictionaries per movie: sum_ratings (running total), count_ratings (number of ratings), and distinct_users (a set to count unique raters). After the pass, I consider only movies with at least 4 distinct raters, calculate the average (sum_ratings[movieId]/count_ratings[movieId]) and track the best (average, movieId). As is the theme of my solutions, this code is optimal since it employs clean loops and little memory.