**Question 6:**

**1. Nucleotide Counting (Q1 & Q2)**

* **Initial Approach (count\_nucleotides):**
  + **Algorithm:** Iterate through the string of length N once. For each character, perform a dictionary lookup and increment, which is an O(1) operation.
  + **Time Complexity:** O(N).
  + **Space Complexity:** O(k), where k is the size of the alphabet (e.g., 4 for 'ATGC'). This is constant space, O(1).
* **Optimized Approach (in DNA class):**
  + **Algorithm:** collections.Counter(seq). This is a built-in, C-optimized hash-map implementation. It still iterates through the string once.
  + **Time Complexity:** O(N). While asymptotically the same, its C implementation makes it significantly faster in practice (lower constant factor) than a pure Python loop.
  + **Space Complexity:** O(k), constant space.
* **Alternative (Worse) Approach:**
  + **Algorithm:** seq.count('A'), seq.count('T'), etc.
  + **Time Complexity:** O(N . k). The .count() method must iterate through the entire string N for *each* nucleotide k. For an alphabet of 4, this is O(4N), which simplifies to O(N), but it's 4 full passes over the data instead of one.

**Conclusion:** collections.Counter is the most performant and "Pythonic" solution.

**2. Reverse Complement Generation (Q5)**

* **Initial Approach (reverse\_complement):**
  + **Algorithm:**
    1. Iterate through the original sequence of length N, building a *new* list of complement characters. (O(N) time, O(N) space for the new list).
    2. "".join() the list. This creates the final complement string. (O(N) time).
    3. Reverse the new string using slicing [::-1]. This creates *another* new string. (O(N) time, O(N) space).
  + **Time Complexity:** O(N) + O(N) + O(N) = O(N).
  + **Space Complexity:** O(N) to store the intermediate list and the intermediate complement string.
* **Optimized Approach (reverse\_complement\_optimized):**
  + **Algorithm:**
    1. Define a translation table using str.maketrans(). This is an O(k) operation, where k is alphabet size (constant time).
    2. Call seq.translate(table). This is a single-pass C-optimized operation to create the complement string. (O(N) time, O(N) space).
    3. Reverse the resulting string using [::-1]. (O(N) time, O(N) space).
  + **Time Complexity:** O(N) + O(N) = O(N).
  + **Space Complexity:** O(N).

Hence, both approaches are asymptotically O(N). However, the translate method is much faster. It performs the entire complement mapping in one C-optimized pass, avoiding the overhead of Python loops, dictionary lookups per-character, and intermediate list creation. For very large genomic sequences, this would be the preferred method.