**1. Problem Understanding**

The program needs to:

* Accept a DNA sequence (made of A, T, G, C) and validate it.
* Calculate the complementary DNA sequence.
* Find the most frequent base (A, T, G, or C) in the sequence.
* Let the user insert a new sequence into the original at a given position.
* Remove a specified pattern from the sequence.
* Decompress a compressed DNA sequence.

So, the program should work with DNA sequences step by step while ensuring user input is valid and providing meaningful output.

**2. User-Friendly Input/Output**

The first goal is to make it easy for the user to interact with the program:

* Ask for input clearly.
* Validate that the DNA sequence only contains valid bases (A, T, G, C).
* Repeat prompts until the user provides valid input.

**3. Functions for Each Task**

To keep the code clean and modular, I decided to write separate methods for each functionality:

1. **Validation of DNA Sequence**:
   * Loops through each character to check if it's valid (A, T, G, or C).
   * If it's not valid, inform the user and ask for a new sequence.
2. **Complementary Sequence**:
   * For every base in the DNA sequence, find its complement:
     + A ↔ T, G ↔ C.
   * Use a StringBuilder for efficient string manipulation.
3. **Most Frequent Base**:
   * Check how many times each base appears in the sequence.
   * Keep track of the base with the highest count.
4. **Insert Sequence**:
   * Insert a new sequence at the user-specified position.
   * Validate the position to ensure it's within bounds.
5. **Remove Pattern**:
   * Remove the first occurrence of a pattern from the sequence.
   * If the pattern doesn’t exist, return the original sequence unchanged.
6. **Decompress DNA**:
   * Handle compression formats like "A3(TG)2" and expand them into full sequences (e.g., ATGTGTG).
   * This part uses logic to parse and repeat patterns based on numbers.

**4. Thought Process for Each Step**

1. **Validation**:
   * DNA sequences only have A, T, G, and C. Anything else is invalid.
   * Using a loop to check each character keeps things simple.
   * If something’s wrong, print a message and ask again.
2. **Complementary Sequence**:
   * DNA strands are complementary, so for each base:
     + A ↔ T and G ↔ C.
   * I used a switch statement because it’s clean and easy to read.
3. **Most Frequent Base**:
   * Count occurrences of each base (A, T, G, C).
   * Compare counts to find the most frequent one.
   * Straightforward logic that loops through the sequence twice.
4. **Insertion**:
   * Use StringBuilder.insert() to insert a sequence at a specific position.
   * Throw an error if the position is invalid to prevent mistakes.
5. **Remove Pattern**:
   * Find the first occurrence of a pattern using indexOf().
   * Rebuild the sequence by skipping the pattern.
6. **Decompression**:
   * This is the tricky part. It involves parsing compressed strings like A3(TG)2.
   * Use loops to process letters, numbers, and nested patterns.
   * Repeat patterns based on numbers and handle parentheses for grouping.

**5. How It Runs**

Here’s how the program flows:

1. It asks the user to enter a DNA sequence and validates it.
2. It generates the complementary sequence and prints it.
3. It finds the most frequent base and prints it.
4. It lets the user:
   * Insert a sequence into the DNA at a chosen position.
   * Remove a pattern from the DNA.
5. Finally, it takes a compressed sequence and decompresses it.