## **Kedar Oza 1032240040**

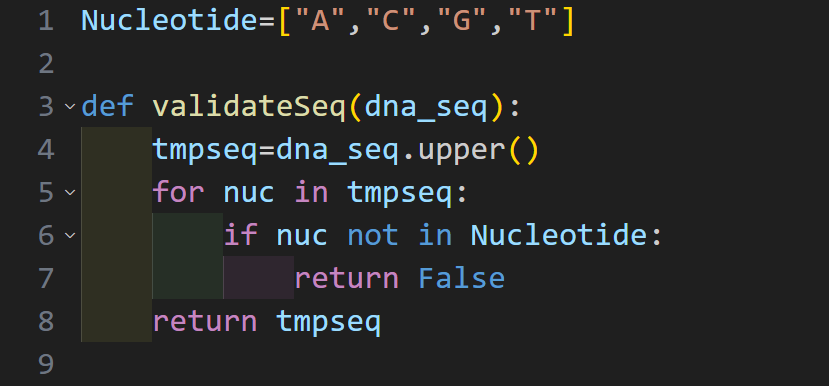
## **Assignment 1: DNA Fundamentals and Basic Tools (CCA3)**

## **Part A: DNA Representation and Basic Operations (40 marks)**

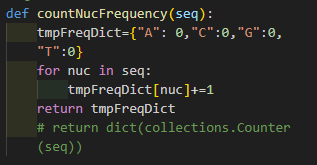
Question 1: DNA Data Structures (15 marks)  
Write a Python program that:

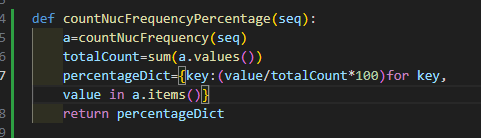
* Implements a DNA class using Python lists and dictionaries to store nucleotide sequences
* Validates DNA sequences (ensures only A, T, G, C nucleotides)
* Implements methods for sequence length, nucleotide counting, and basic statistics
* Handles invalid nucleotides with appropriate error messages

Ans ->



Question 2: Nucleotide Counting and Analysis (12 marks)  
Create functions to:

* Count individual nucleotides in a DNA sequence
* 
* Calculate nucleotide frequencies as percentages

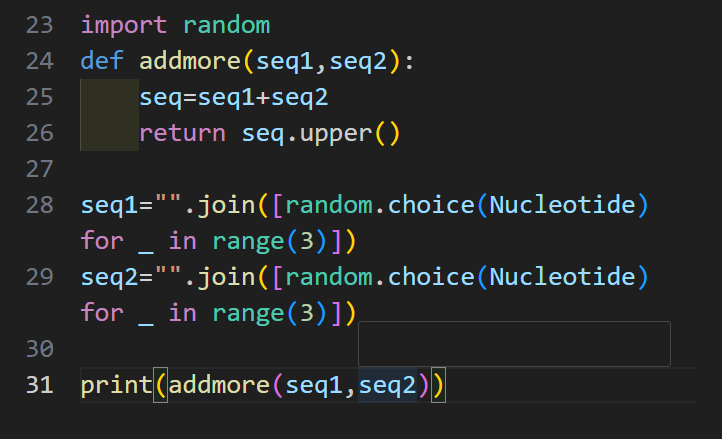


* Generate a comprehensive nucleotide analysis report
* Compare nucleotide composition between two sequences

Question 3: String Manipulation for Genomics (13 marks)  
Implement algorithms for:

* Converting DNA sequences to uppercase/lowercase
* Removing non-nucleotide characters from sequences
* Splitting long sequences into codons (groups of 3)
* Merging multiple DNA fragments into a single sequence

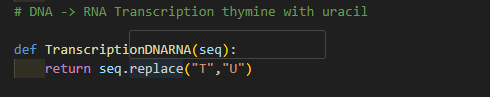
Ans->



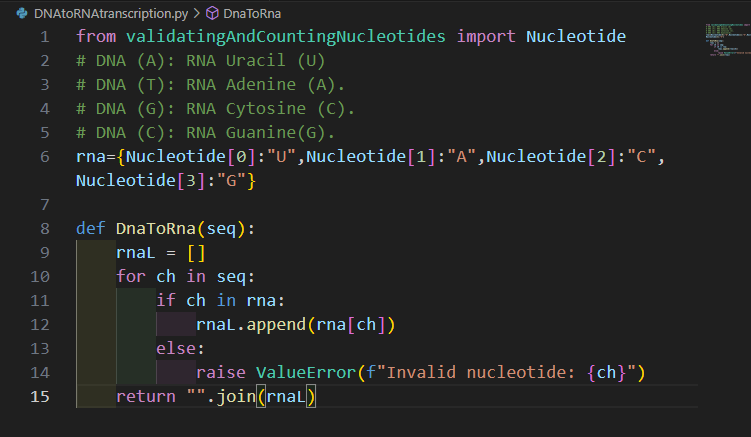
## **Part B: Essential DNA Algorithms (35 marks)**

Question 4: DNA Transcription (15 marks)  
Develop a comprehensive transcription system:

* Convert DNA to RNA (T→U substitution)

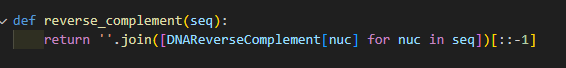


* Handle both coding and template strands
* Implement error checking for invalid sequences
* Create batch processing for multiple sequences

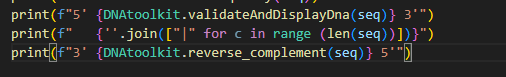


Question 5: Reverse Complement Generation (20 marks)  
Build robust reverse complement functionality:

* Generate reverse complement using dictionary mapping



* Handle degenerate nucleotides (R, Y, S, W, K, M)
* Implement both 5'-3' and 3'-5' orientations



* Optimize for large sequences using efficient algorithms

## **Part C: Code Optimization and Testing (25 marks)**

Question 6: Algorithm Optimization (15 marks)  
Optimize your DNA processing functions:

* Compare time complexity of different approaches
* Implement memory-efficient solutions for large sequences
* Use Python's built-in functions for performance improvement
* Profile and benchmark your implementations

Ans->

Question 7: Comprehensive Testing Suite (10 marks)  
Create a testing framework:

* Write unit tests for all DNA manipulation functions
* Include edge cases (empty sequences, single nucleotides, very long sequences)

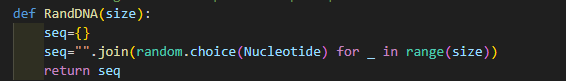
 **Empty Sequences:** Test get\_complement("") to ensure it returns "". For get\_gc\_content(""), ensure it handles the **ZeroDivisionError** by returning 0.0.

 **Single Nucleotides:** Test single characters ("A", "G", etc.) to confirm correct character mapping.

 **Invalid Characters:** Test sequences containing non-standard IUPAC characters (e.g., "ATZGC") to ensure the function raises a ValueError or handles the character as a clearly defined placeholder.

 **Performance Cases:** Use very long sequences (105 to 106 nucleotides) as performance tests to ensure functions complete within an acceptable time limit.

* Implement test data generators for random DNA sequences



* Document test coverage and results

 **Test Coverage Tool (pytest-cov)**: Use this tool to automatically measure the percentage of code lines executed by the tests. The target is **≥95% coverage** for all core manipulation functions.

 **Reporting**: Output the coverage results to a summary report and an HTML file.

 **Test Status Documentation**: Include the testing results in the project documentation (e.g., README.md or a formal report), clearly stating: "All unit tests passed. Total code coverage: **X%**."