ACADEMIC CITY UNIVERSITY

GE 1206 PROGRAMMING IN C

END of SEMESTER EXAMINATION - 2024/2025

Question Paper

Allocated Time: 21 days
Examination Date: August, 2025

Maximum mark: 20 marks

Practical Exam.

Click me for same but cleaner question format https://dub.sh/cHkkFVx

Question: Readability Analyzer

Problem Statement

You are to implement a program that calculates the approximate U.S. school grade level required to comprehend a given body of English text. The program uses the *Coleman–Liau Index*, which is a readability formula based on the number of letters, words, and sentences in the text.

Background

Books are often graded for reading level based on features such as word length and sentence complexity. For instance, *Charlotte's Web* is readable by students in grades 2 to 4, while *The Giver* is intended for grades 8 to 12. The Coleman–Liau Index provides a numerical score indicating the grade level required to understand a text.

The Coleman–Liau index is calculated using the formula:

- $\text{Index} = 0.0588 \times L - 0.296 \times S - 15.8$

Where:

- L is the average number of **letters per 100 words**, and
- S is the average number of **sentences per 100 words**.

Your task is to write a program that:

- Prompts the user to enter a block of English text.
- Computes the number of letters, words, and sentences in the text.
- Calculates and prints the reading grade level of the text using the Coleman–Liau index.

Specification

1. **Input**

   The program should prompt the user with:

   Text:

   and then accept a single line of input text from the user.

2. **Processing**

   The program should perform the following:

   a. Count all alphabetical characters (A–Z and a–z) as **letters**.
   b. Count all sequences of characters separated by spaces as **words**.
   c. Count all occurrences of `.`, `!`, or `?` as **sentence terminators**.
   d. Compute the values of `L` and `S`.
   e. Apply the Coleman–Liau formula.
   f. Round the result to the nearest whole number.

3. **Output**

   Based on the result of the index:

- If the result is less than 1, print:

  Before Grade 1

- If the result is 16 or more, print:

  Grade 16+

- Otherwise, print:

  Grade X

  Where `X` is the rounded grade level.

Question: DNA Complement Generator

Problem Statement

Write a C program that takes a DNA sequence from the user and prints its **complementary sequence**. In DNA, each base pairs with another specific base:

- `A` pairs with `T`
- `C` pairs with `G`

Your program should replace each base in the user's input with its complementary base and display the final result.

Background

DNA (short for *Deoxyribonucleic Acid*) is like the instruction manual for living things. It tells cells how to grow, function, and reproduce. DNA is made up of **bases** — the "letters" of the DNA alphabet: `A`, `T`, `C`, and `G`.

These bases always pair up in a specific way:

- `A` (adenine) pairs with `T` (thymine)
- `C` (cytosine) pairs with `G` (guanine)

Scientists often need to find the **complement** of a DNA sequence — that is, the matching sequence based on base pairing rules. For example, if the original strand is `ATCG`, the complement is `TAGC`.

This idea is important in **DNA analysis**, **genetic testing**, **forensics**, and even in making medicines. For example, to read or copy DNA in a lab, computers first generate the complement.

In this task, you'll simulate part of that process by writing a simple C program that produces the complement of any DNA strand entered by the user.

Specification

1. **Input**

   The program should ask the user to enter a DNA sequence like this:

   Enter a DNA sequence:

   The input will be a string made up of the characters `A`, `T`, `C`, and `G`. (Optionally, you can also handle lowercase letters.)

2. **Processing**

   Your program should:

   a. Go through each base in the input
   b. Replace:
      i. `A` with `T`
      ii. `T` with `A`
      iii. `C` with `G`
      iv. `G` with `C`
   c. (Bonus: Handle lowercase versions like `a` → `t`, etc.)

3. **Output**

   Print the resulting complementary sequence, like this:

   Complement: TAGC

Example Runs

```
$ ./dna_complement
Enter a DNA sequence: ATCG
Complement: TAGC

$ ./dna_complement
Enter a DNA sequence: gcat
Complement: cgtA
```

Question: Caesar Cipher Encryption

Problem Statement

You are to implement a program that encrypts a message using **Caesar's Cipher**, a simple encryption technique where each letter in the plaintext is shifted by a fixed number of positions in the alphabet. The key to this cipher is the number of positions used to shift each character.

Background

Caesar's Cipher is one of the earliest known encryption techniques. It replaces each letter in the plaintext with another letter a fixed number of positions down the alphabet. For example, with a key of 1, "A" becomes "B," "B" becomes "C," ..., and "Z" wraps around to become "A."

To decrypt, the recipient shifts each letter in the opposite direction using the same key.

This cipher is not secure by modern standards, but it provides a useful introduction to encryption.

- Unencrypted text is called **plaintext**
- Encrypted text is called **ciphertext**
- The number used to shift the letters is the **key**

Example:

Encrypting `HELLO` with a key of 1 results in:

Plaintext	H	E	L	L	O
+ key	1	1	1	1	1
Ciphertext	I	F	M	M	P

The formula for encryption is:

$$c_i = (p_i + k)$$

Where:

- p_i is the position of the $i-th$ character in the alphabet (A/a = 0, B/b = 1, ..., Z/z = 25)
- k is the key
- c_i is the position of the encrypted character

For example, to encrypt "Hi" with a key of 3:

- $H \rightarrow 7 \rightarrow (7+3) \bmod 26 = 10 \rightarrow K$
- $i \rightarrow 8 \rightarrow (8+3) \bmod 26 = 11 \rightarrow L$

So “Hi” becomes “Kl.”

Specification

1. **Input**

- The program should prompt the user with:

  plaintext:

  and accept a single line of input as the message to be encrypted.

2. **Processing**

   - Each alphabetical character in the plaintext should be **rotated by the given key**, wrapping around the alphabet when needed.
   - The case of each letter must be **preserved** (i.e., lowercase stays lowercase, uppercase stays uppercase).
   - All non-alphabetic characters should remain unchanged.
3. **Output**

- The program must output:

  ciphertext: <result>
  - followed by a newline.

  - After successfully printing the ciphertext, the program should return `0`.

-

NOTE: You are required to upload your project on GitHub.

Good Luck

End of Semester Exams;