ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

Nhập môn lập trình - CO1003

Bài tập lớn

MẬT MÃ DNA CỦA NHÀ KHOA HỌC LẬP DỊ

TP. HỒ CHÍ MINH, THÁNG 10/2025

ĐẶC TẢ BÀI TẬP LỚN

Phiên bản 1.0

1 Chuẩn đầu ra

Sau khi hoàn thành bài tập lớn này, sinh viên ôn lại và sử dụng thành thục:

- Các cấu trúc rẽ nhánh
- Các cấu trúc lặp
- Mảng 1 chiều và mảng 2 chiều
- Xử lý chuỗi ký tự
- Hàm và lời gọi hàm
- Cấu trúc do người dùng tự định nghĩa (Struct)

2 Dẫn nhập

Trong một thành phố tương lai nơi khoa học và công nghệ thống trị, xuất hiện một nhân vật bí ẩn được gọi là Bác Học X. Ông vốn là một nhà di truyền học thiên tài, nhưng sau một thí nghiệm thất bại, tâm trí trở nên méo mó và đầy tham vọng điên rồ. Tin rằng loài người quá yếu đuối, Bác Học X quyết định "nâng cấp" nhân loại bằng cách can thiệp trực tiếp vào bản đồ gene. Ông phát minh ra một hệ thống mã hóa codon lạ thường, biến các chuỗi DNA tự nhiên thành những gene đột biến, từ đó tạo ra các sinh vật quái vật nửa người, nửa dã thú.

Các vụ tấn công bí ẩn nhanh chóng gieo rắc nỗi kinh hoàng khắp thành phố. Tại hiện trường chỉ còn sót lại những mảnh DNA kỳ lạ, sắp xếp thành chuỗi base A, T, C, G (hoặc đôi khi thay bằng U – RNA). Người dân gọi đó là "mã gene của quái vật". Trước hiểm họa lan rộng, một nhóm sinh viên tài năng được triệu tập với nhiệm vụ giải mã hệ thống gene của Bác Học X, phát hiện các codon đột biến, và viết chương trình để phân tích – mô phỏng quá trình dịch mã gene thành protein.

Mục tiêu cuối cùng của họ là tìm ra gene gốc có thể đảo ngược đột biến, cứu lấy thành phố trước khi quá muộn. Thành công hay thất bại phụ thuộc hoàn toàn vào khả năng lập trình, tư duy giải thuật và sự am hiểu về cấu trúc DNA. Câu hỏi đặt ra: Liệu bạn có thể bẻ khóa di sản của Bác Học X trước khi những sinh vật quái vật chiếm lĩnh thế giới?

3 Yêu cầu

Để giải mã và ngăn chặn âm mưu của gã bác học điên, hệ thống cần có khả năng phân tích và xử lý các đoạn mã gene mà hắn để lại. Những đoạn mã này dựa trên nguyên lý DNA thật với bốn bazơ cơ bản (A, T, C, G), kết hợp thành các bộ ba codon tạo nên chuỗi gene và cấu trúc protein. Chỉ một đột biến nhỏ – ví dụ một codon bị thay đổi – cũng có thể dẫn đến những biến đổi lớn, gây hậu quả nghiêm trọng cho sinh vật mang gene đó.

Mục này đồng thời nêu ra các yêu cầu để sinh viên xây dựng hệ thống xử lý chuỗi gene bằng cách hiện thực các hàm được mô tả. Mỗi yêu cầu kèm số điểm, thể hiện mức tối đa có thể đạt nếu sinh viên cài đặt chính xác.

3.1 Kiểu của câu lệnh

Hệ thống sẽ tự động nhận các yêu cầu của người dùng và chuyển thành các câu lệnh để xử lý. Hệ thống sẽ bao gồm các lệnh sau, cùng với các giá trị thuộc `Enum` **commandType** tương ứng:

Yêu cầu (Keyword)	**Giá trị Enum tương ứng**
Sanitize the code	`SANITIZE`
Validation check	`CHECK`
Find the complement sequence	`COMPLEMENT`
Find the mRNA sequence	`MRNA`
Encode	`ENCODE`
Decode	`DECODE`
Create the table	`CREATE`
Print the gene table	`PRINT`
Detect a mutation	`MUTATION`
Repair the mutated gene	`REPAIR`
Reverse	`REVERSE`
Find the longest gene	`LONGEST`
Find the shortest gene	`SHORTEST`
Print the DNA pipeline	`PIPELINE`
Invalid command	`INVALID`

Bảng 1: Danh sách các chức năng xử lý chuỗi GENE

Yêu cầu 1 (0.5 điểm): Hiện thực hàm **getCommandType** để trả về loại của câu lệnh, thông tin cụ thể:

- **Khai báo hàm:** `enum commandType getCommandType(char* command)`
- **Tham số đầu vào:**

  - **command (kiểu char*)**: chuỗi chứa một lệnh.

- **Trả về:** giá trị trả về có kiểu là **commandType** tương ứng với kiểu của lệnh command được truyền vào.
- **Yêu cầu hàm:** Gọi w là từ đầu tiên xuất hiện trong câu lệnh. Nếu w trùng với một trong kiểu câu lệnh được mô tả ở trên thì trả về giá trị enum commandType tương ứng. Ngược lại, w là một từ không hợp lệ thì trả về giá trị INVALID. Lưu ý, các lệnh không biệt chữ hoa và chữ thường. Xem ví dụ 3.1 để hiểu rõ hơn.

Ví dụ 3.1

- Với lời gọi hàm

```
getCommandType("Encode the first GENE")
```

Trả về giá trị

```
ENCODE
```

- Với lời gọi hàm

```
getCommandType("Check the validation of the gene is the first step!")
```

Trả về giá trị

```
CHECK
```

- Với lời gọi hàm

```
getCommandType("Print the full detailed table.")
```

Trả về giá trị

```
PRINT
```

- Với lời gọi hàm

```
getCommandType("Let's solve the gene's secret code.")
```

Trả về giá trị

```
INVALID
```

3.2 Kiểm tra tính hợp lệ của Gene

Sinh viên cần hiện thực hai hàm: **sanitizeCode** và **validCheck** nhằm kiểm tra cấu trúc và tính hợp lệ sinh học của một chuỗi DNA.

Cụ thể, hàm **sanitizeCode** sẽ tiếp nhận đầu vào `codes` là một chuỗi ký tự biểu diễn DNA của một gene nhưng có thể đã bị nhiễu bởi các ký tự không hợp lệ (gã bác học cố tình chèn thêm để gây khó khăn trong việc phân tích!). Nhiệm vụ của sinh viên là làm sạch chuỗi này trước khi tiến hành kiểm tra bằng hàm **validCheck**.

3.2.1 Chuẩn hoá DNA

Một chuỗi DNA hợp lệ cần tuân thủ các điều kiện sau:

- Chỉ bao gồm các ký tự hợp lệ, ngoài các ký tự dưới đây, tất cả ký tự khác được xem là đặc biệt và phải loại bỏ:
  - Các ký tự thường: [a, c, g, t] (cần được chuyển đổi thành ký tự in hoa tương ứng [A, C, G, T]).
  - Các ký tự in hoa: [A, C, G, T].
  - Các ký tự đặc biệt (ngoài chữ và số) có thể xuất hiện trong đoạn mã nhiễu: `! @ # $ % - _ & * ?`
- Độ dài chuỗi được đảm bảo nằm trong khoảng [1, 300] ký tự.

Yêu cầu 2 (0.4 điểm): Hiện thực hàm **sanitizeCode** để làm sạch chuỗi gene. Thông tin chi tiết:

- **Khai báo hàm:** `char* sanitizeCode(char * codes)`
- **Tham số đầu vào:**
  - **codes (kiểu char*)**: mã DNA có thể chứa ký tự không hợp lệ.
- **Trả về:**
  - Chuỗi DNA của gene sau khi đã được làm sạch, loại bỏ toàn bộ ký tự đặc biệt.

3.2.2 Xác minh gene hợp lệ

Một chuỗi gene chỉ được xem là hợp lệ khi đáp ứng các tiêu chí sau:

- Chuỗi có độ dài ≥ 9. Có ít nhất ba bộ codon.
- Độ dài của chuỗi chia hết cho 3.
- Gene hợp lệ phải bắt đầu bằng codon **ATG** và kết thúc bằng một trong ba codon sau:
  - **TAA**

- **TAG**
- **TGA**

Yêu cầu 3 (0.7 điểm): Hiện thực hàm **validCheck** để kiểm tra tính hợp lệ của gene. Thông tin chi tiết:

- **Khai báo hàm:** `bool validCheck(char * codes)`
- **Tham số đầu vào:**
  - **codes (kiểu char*)**: chuỗi DNA cần kiểm tra.
- **Yêu cầu:**
  - Trước hết sử dụng hàm **sanitizeCode** để loại bỏ ký tự không hợp lệ.
  - Tiếp theo kiểm tra cấu trúc của chuỗi theo các quy định nêu trên để xác định tính hợp lệ.
- **Trả về:**
  - Nếu chuỗi DNA hợp lệ, trả về giá trị `TRUE` và in ra:

    ```
    VALID GENE
    ```

  - Nếu độ dài của chuỗi == 1, trả về `FALSE` và in ra:

    ```
    INVALID
    TYPE: BASE
    ```

  - Nếu độ dài của chuỗi == 3, trả về `FALSE` và in ra:

    ```
    INVALID
    TYPE: CODON
    ```

  - Trong các trường hợp còn lại, trả về `FALSE` và in ra:

    ```
    INVALID CODE
    ```

Ví dụ 3.2

- Với mã `codes`

```
"ATGAAATAA"
```

`validCheck(codes)` trả về giá trị

```
VALID GENE
```

- Với mã `codes`

```
"A-TG@@ata–TAG??!!"
```

`validCheck(codes)` trả về giá trị

```
VALID GENE
```

- Với mã `codes`

```
"AAA@@ATG–TAG__!"
```

`validCheck(codes)` trả về giá trị

```
INVALID CODE
```

- Với mã `codes`

```
"??__A@@!!"
```

`validCheck(codes)` trả về giá trị

```
INVALID
TYPE: BASE
```

3.3 Biểu diễn DNA dưới dạng bổ sung và mRNA

Sinh viên cần hiện thực hai hàm: **generateComplement** và **generateMRNA** để biểu diễn các dạng chuyển đổi cơ bản của một chuỗi DNA. Đây là bước quan trọng giúp làm rõ cơ chế

sinh học tự nhiên trước khi tiến hành các phép biến đổi phức tạp hơn của gã bác học.

Cụ thể, hàm **generateComplement** tiếp nhận chuỗi DNA gốc `codes` và tạo ra chuỗi bổ sung dựa trên nguyên tắc bắt cặp bazơ. Sau đó, hàm **generateMRNA** thực hiện quá trình phiên mã bằng cách sử dụng chuỗi bổ sung (template strand) để sinh ra chuỗi mRNA tương ứng.

Từ một chuỗi DNA đầu vào, hệ thống sẽ sinh ra hai dạng biểu diễn phục vụ phân tích:

- **Chuỗi bổ sung (Complement sequence):** được xác định dựa trên nguyên tắc bắt cặp bazơ:

$$\mathrm{A} \leftrightarrow \mathrm{T}, \quad \mathrm{C} \leftrightarrow \mathrm{G}$$

- **Chuỗi mRNA (mRNA sequence):** được phiên mã từ chuỗi bổ sung theo quy tắc:

$$\mathrm{A} \rightarrow \mathrm{U}, \quad \mathrm{T} \rightarrow \mathrm{A}, \quad \mathrm{C} \rightarrow \mathrm{G}, \quad \mathrm{G} \rightarrow \mathrm{C}$$

Hai phép biến đổi này cho phép quan sát DNA ban đầu dưới hai dạng song song: *Complement sequence* và *mRNA sequence*, từ đó hỗ trợ phân tích và giải mã gene một cách trực quan và hiệu quả hơn.

Yêu cầu 4 (0.5 điểm): Hiện thực hàm **generateComplement** để sinh ra chuỗi bổ sung của DNA. Thông tin chi tiết:

- **Khai báo hàm:** `char* generateComplement(char * codes)`
- **Tham số đầu vào:**
  - **codes (kiểu char*)**: chuỗi DNA gốc.
- **Yêu cầu:**
  - Trước hết sử dụng hàm **validCheck** để kiểm tra tính hợp lệ của chuỗi DNA.
  - Sau đó, tạo chuỗi bổ sung theo nguyên tắc bắt cặp bazơ.
- **Trả về:**
  - Nếu mã hợp lệ, trả về chuỗi bổ sung.
  - Nếu mã không hợp lệ, trả về chuỗi gốc.

Yêu cầu 5 (0.5 điểm): Hiện thực hàm **generateMRNA** để sinh ra chuỗi mRNA từ DNA gốc. Thông tin chi tiết:

- **Khai báo hàm:** `char* generateMRNA(char * codes)`

- **Tham số đầu vào:**
  - **codes (kiểu char*)**: chuỗi DNA hợp lệ cần được phiên mã.
- **Yêu cầu:**
  - Gọi hàm **generateComplement** để lấy chuỗi bổ sung.
  - Dựa trên chuỗi bổ sung, tạo ra chuỗi mRNA theo quy tắc ánh xạ.
- **Trả về:**
  - Chuỗi mRNA thu được sau phiên mã.

Ví dụ 3.3

- Với mã `codes`

```
"ATGAAATAA"
```

`generateComplement` trả về

```
"TACTTTATT"
```

`generateMRNA` trả về

```
"AUGAAAUAA"
```

- Với mã `codes`

```
"CGTACG"
```

`generateComplement` trả về

```
"GCATGC"
```

`generateMRNA` trả về

```
"CGUACG"
```

- Với mã `codes`

```
"ATGCGTACGTTTAAA"
```

`generateComplement` trả về

```
"TACGCATGCAAATTT"
```

`generateMRNA` trả về

```
"AUGCGUACGUUUAAA"
```

3.4 Mã hoá và Giải mã Gene

Sinh viên cần hiện thực hai hàm: **encode** và **decode** nhằm xử lý việc mã hoá và giải mã gene, qua đó hiểu rõ hơn cơ chế ẩn giấu thông tin mà gã bác học quái dị đã áp dụng.

Cụ thể, hàm **encode** tiếp nhận đầu vào `codes` là một mã gene thô (raw) chưa qua kiểm tra tính hợp lệ. Nhiệm vụ của sinh viên là biến đổi chuỗi gốc này thành mã đặc biệt theo quy tắc riêng của gã bác học. Ngược lại, hàm **decode** phục hồi chuỗi gốc ban đầu từ mã đặc biệt đó.

Dựa trên các tài liệu rời rạc thu thập được trong phòng thí nghiệm, ta có thể tái dựng tương đối chính xác cách gã bác học đã sử dụng để biến đổi DNA thường thành DNA đặc biệt. Quy tắc cụ thể như sau:

- Chuỗi DNA cần được làm sạch và kiểm tra hợp lệ bằng hàm **validCheck** trước khi thực hiện hai tác vụ này.
- Quá trình mã hoá được thực hiện theo từng **codon** (bộ 3 base) trong chuỗi DNA, lần lượt từ trái sang phải.
- Công thức mã hoá cho một codon (3 base) có dạng **XXYYYY**, trong đó:
  - a) **XX** được xác định dựa trên base đầu tiên của codon:
    - Base A → 65
    - Base C → 67
    - Base G → 71
    - Base T → 84
  - b) **YYYY** được tính từ hai base còn lại theo công thức:

    $$\text{YYYY} = [(\text{2nd base} * 31) + (\text{3rd base} * 37)] \ \% \ 10000$$

    trong đó `base` là mã ASCII của ký tự tương ứng.

Yêu cầu 6 (0.9 điểm): Hiện thực hàm **encode** để mã hoá gene. Thông tin chi tiết:

- **Khai báo hàm:** `char* encode(char * codes)`
- **Tham số đầu vào:**
  - **codes (kiểu char*)**: chuỗi DNA gốc.
- **Yêu cầu:**
  - Gọi hàm **validCheck** để kiểm tra tính hợp lệ của chuỗi DNA.

  - Nếu hợp lệ, tiến hành mã hoá từng bộ codon (3 base) theo quy tắc đã mô tả.

- **Trả về:**
  - Nếu chuỗi hợp lệ, trả về chuỗi gene đặc biệt sau khi mã hoá.
  - Nếu không hợp lệ, trả về chuỗi gốc.
- **Ví dụ:** Xem ví dụ 3.4.

Yêu cầu 7 (0.5 điểm): Hiện thực hàm **decode** để giải mã gene. Thông tin chi tiết:

- **Khai báo hàm:** `char* decode(char * codes)`
- **Tham số đầu vào:**
  - **codes (kiểu char*)**: chuỗi gene đặc biệt cần được giải mã. Chuỗi đầu vào đảm bảo hợp lệ (chỉ chứa ký tự số, chiều dài hợp lý, v.v.).
- **Trả về:**
  - Chuỗi DNA gốc được phục hồi theo quy tắc giải mã.
- **Ví dụ:** Xem ví dụ 3.5.

Ví dụ 3.4

- Với `codes`

```
"ATG"
```

`validCheck` và `encode` trả về

```
INVALID
TYPE: CODON
ATG
```

- Với `codes`

```
"A-TG@@ata–TAG??!!"
```

`validCheck` và `encode` trả về

```
VALID GENE
"6552316550098446 42"
```

- Với `codes`

```
"AAA@@ATG–TAG__!"
```

`validCheck` và `encode` trả về

```
INVALID CODE
AAAATGTAG
```

- Với `codes`

```
"??__A@@!!"
```

`validCheck` và `encode` trả về

```
INVALID
TYPE: BASE
A
```

Ví dụ 3.5

- Với `codes`

```
"6552316544208444 20"
```

`decode` trả về

```
"ATGAAATAA"
```

- Với `codes`

```
"655231655009844642"
```

`decode` trả về

```
"ATGATATAG"
```

- Với `codes`

```
"65523167448271571265470484442 0"
```

`decode` trả về

```
"ATGCCAGTTACGTAA"
```

- Với `codes`

```
"6552317146806550096751858446 42"
```

`decode` trả về

```
"ATGGGCATACCTTAG"
```

Lưu ý: Khi in ra kết quả, sinh viên cần đảm bảo **CHÍNH XÁC TUYỆT ĐỐI** từng ký tự như mô tả. Sai khác chỉ một ký tự (kể cả khoảng trắng thừa) cũng sẽ bị tính là sai testcase.

3.5 Cấu trúc Gene

Sau khi giải mã được một phần các chuỗi gene kỳ lạ trong phòng thí nghiệm bí ẩn, nhóm nghiên cứu phát hiện rằng ngoài thông tin về cơ chế mã hoá – giải mã, gã bác học quái dị còn để lại một kho dữ liệu chứa nhiều gene đột biến đặc biệt. Những gene này được hắn dự định tiêm vào các thí nghiệm của mình nhằm tạo ra những sinh vật vượt ngoài tầm kiểm soát.

Nhiệm vụ của chúng ta không chỉ dừng lại ở việc giải mã gene, mà còn cần phát hiện và khắc phục các gene đột biến này trước khi chúng kịp được sử dụng. Để làm được điều đó, trước tiên sinh viên cần xây dựng một cấu trúc dữ liệu `struct Gene` để lưu trữ thông tin gene.

Yêu cầu 8 (0.5 điểm): Hiện thực cấu trúc **`struct` Gene**, chi tiết như sau:

- **Khai báo cấu trúc:** `struct Gene{}`
- **Các trường trong cấu trúc:**
  - **name (kiểu char[])**: tên gene.
  - **org_seq (kiểu char[])**: chuỗi DNA gốc của gene.
  - **length (kiểu int)**: độ dài chuỗi DNA gốc.
  - **encoded_seq (kiểu char[])**: chuỗi DNA đã được mã hoá đặc biệt.
  - **complement_seq (kiểu char[])**: chuỗi DNA bổ sung.
  - **mrna_seq (kiểu char[])**: chuỗi mRNA tương ứng.

Ví dụ 3.6

Ví dụ về cách sử dụng đối tượng **struct Gene**:

```
1    struct Gene gene;
2    char *encoded_gene = encode(gene.org_seq);
```

3.6 Bảng tổng hợp các Gene

Bảng này lưu trữ thông tin của tất cả các gene đã được nghiên cứu cho đến thời điểm hiện tại. Giả sử ta có N gene đã biết, khi đó những phiên bản gene còn lại (không nằm trong danh sách) sẽ được xem là gene đột biến do gã bác học tạo ra. Nhiệm vụ tiếp theo là đưa danh sách gene này vào hệ thống lưu trữ.

3.6.1 Đọc bảng Gene

Yêu cầu 9 (0.5 điểm): Hiện thực hàm **createTable** để đọc thông tin của toàn bộ gene hiện có vào một bảng dữ liệu chung. Thông tin chi tiết:

- **Khai báo hàm:** `int createTable(struct Gene * emptyTable, char ** geneInfo)`
- **Tham số đầu vào:**
  - **emptyTable (mảng struct Gene)**: danh sách rỗng, nơi sẽ được ghi dữ liệu gene từ `geneInfo`.
  - **geneInfo (kiểu char**)**: mảng hai chiều chứa chuỗi DNA của các gene. Mỗi phần tử ứng với một gene. Phần tử cuối cùng trong danh sách sẽ là **NULL** để đánh dấu kết thúc.
  - **Lưu ý**: Mã gene đầu vào luôn đảm bảo hợp lệ. Tuy nhiên, có thể xuất hiện cả dạng gốc (ACGT) lẫn dạng đặc biệt (format `XXYYYY`). Nếu gặp dạng đặc biệt, cần giải mã về dạng gốc trước khi lưu.
- **Yêu cầu:**
  - Với mỗi chuỗi DNA trong `geneInfo`, tạo một đối tượng `Gene` và thêm vào bảng.
  - Tên gene sẽ được đặt lần lượt từ `"G1"` đến `"GN"`, với N là số lượng gene trong danh sách.
  - Ngoài chuỗi DNA gốc, cần bổ sung đầy đủ các thuộc tính còn lại: độ dài gene, chuỗi mã hoá đặc biệt, chuỗi bổ sung, và chuỗi mRNA.
  - **Lưu ý**: Mã gene đầu vào có thể ở dạng gốc hoặc dạng đặc biệt. Nếu là dạng đặc biệt, bắt buộc phải được chuyển đổi về chuỗi DNA gốc trước khi lưu.
- **Trả về:**
  - Trả về số lượng gene được thêm vào danh sách.

Ví dụ 3.7

Ví dụ về danh sách `geneInfo`:

```
char *geneInfo[] = {
    "ATGCGTACGTAA",
    "ATGTTTCCCTAA",
    "ATGAAATGA",
    "ATGCGTGGGCTAATAG",
    "ATGCGCTGA",
    NULL
};
```

3.6.2 In bảng Gene

Yêu cầu 10 (0.4 điểm): Hiện thực hàm **printTable** để in toàn bộ thông tin của các gene trong bảng. Thông tin chi tiết:

- **Khai báo hàm:** `void printTable(struct Gene * geneTable, int count)`
- **Tham số đầu vào:**
  - **geneTable (mảng struct Gene)**: danh sách các gene cần in.
  - **count (kiểu int)**: số lượng gene trong danh sách.
- **Yêu cầu:**
  - In ra bảng thông tin gene theo đúng định dạng được chỉ định.
  - Cần đảm bảo căn chỉnh chính xác theo format mẫu.
- **Format output (mẫu):**

No.	Gene Name	DNA Sequence
1	G1	ATGCGTACGTAA
2	G2	ATGTTTCCCTAA
3	G3	ATGAAATGA
4	G4	ATGCGTGGCTAATAG
5	G5	ATGCGCTGA

Bảng 2: Sample gene table

3.7 Đột biến Gene

Nhiệm vụ tiếp theo là phát hiện những gene đột biến trong dữ liệu thí nghiệm hiện tại, từ đó điều chỉnh và khôi phục chúng về dạng gốc, nhằm ngăn chặn nguy cơ lan truyền các chuỗi gene độc hại trong thí nghiệm của gã bác học.

3.7.1 Kiểm tra Gene đột biến

Yêu cầu 11 (0.5 điểm): Hiện thực hàm **isMutated** để kiểm tra một gene có phải là gene đột biến hay không. Thông tin chi tiết:

- **Khai báo hàm:** `int isMutated(struct Gene gene, struct Gene * geneTable, int count)`
- **Tham số đầu vào:**
  - **gene (struct Gene)**: đối tượng gene cần kiểm tra.
  - **geneTable (mảng struct Gene)**: danh sách các gene bình thường đã biết.
  - **count (int)**: số lượng gene trong danh sách.
- **Yêu cầu:**
  - Gene đột biến được định nghĩa là gene không xuất hiện trong bảng thông tin các gene bình thường.
  - Hàm cần so sánh chuỗi `org_seq` của gene cần kiểm tra với danh sách `geneTable`.
- **Trả về:**
  - Trả về `1` nếu gene là gene đột biến.
  - Trả về `0` nếu gene thuộc danh sách gene bình thường.

Ví dụ 3.8

- Với chuỗi gene

```
"ATGCGTACGTAA"
```

Dựa vào Bảng 2, `isMutated` trả về

```
"0"
```

- Với chuỗi gene

```
"ATGCGTACGAAATTACTATAA"
```

Dựa vào Bảng 2, `isMutated` trả về

```
"1"
```

3.7.2 Sửa đổi Gene đột biến

Sinh viên cần hiện thực hai hàm **findMutatedCodon** và **repairMutation** để phát hiện và khôi phục các codon bị đột biến trong một gene. Các hàm nhận vào một đối tượng `Gene` và chỉnh sửa trực tiếp chuỗi `sequence` theo gene gốc cùng các quy tắc sau:

- Cần xác định vị trí codon bị đột biến (khác với gene gốc). Nếu codon đột biến thuộc:
  - **Codon đầu (start codon):** luôn sửa thành `ATG`.
  - **Codon cuối (stop codon):** luôn sửa thành `TAG`.
  - **Codon thân:**
    - Vị trí lẻ (1, 3, 5,...): thay toàn bộ codon bằng base có số lần xuất hiện ít nhất trong gene hiện tại.
    - Vị trí chẵn (2, 4, 6,...): loại bỏ codon khỏi chuỗi. Gene được đảm bảo độ dài ≥ 9.

Yêu cầu 12 (0.4 điểm): Hiện thực hàm **findMutatedCodon** để xác định vị trí codon bị đột biến trong gene. Thông tin chi tiết:

- **Khai báo hàm:** `int findMutatedCodons(struct Gene mutatedGene, char * originalGene)`

- **Tham số đầu vào:**
  - **mutatedGene (struct Gene)**: gene đã bị đột biến.
  - **originalGene (char*)**: gene gốc dùng để so sánh và phát hiện codon sai khác.
- **Trả về:**
  - Vị trí (số thứ tự) của codon bị đột biến.

Ví dụ 3.9

- Với `mutatedGene` và `originalGene`:

```
"ATGCGTACGTCC" & "ATGCGTACGTAA"
```

`findMutatedCodon` trả về

```
"4"
```

- Với `mutatedGene` và `originalGene`:

```
"ATGCAAGGGCTAATAG" & "ATGCGTGGGCTAATAG"
```

`findMutatedCodon` trả về

```
"2"
```

Yêu cầu 13 (0.6 điểm): Hiện thực hàm **repairMutation** để phục hồi gene dựa trên vị trí codon đột biến đã phát hiện. Thông tin chi tiết:

- **Khai báo hàm:** `char* repairMutation(struct Gene gene, int pos)`
- **Tham số đầu vào:**
  - **gene (struct Gene)**: đối tượng gene cần điều chỉnh.
  - **pos (int)**: vị trí codon đột biến cần sửa.
- **Trả về:**
  - Chuỗi DNA của gene sau khi đã được sửa.

Ví dụ 3.10

- Với gene và vị trí `pos`:

```
"ATGCGTACGTAA" & "2"
```

`repairMutation` trả về

```
"ATGACGTAA"
```

- Với gene và vị trí `pos`:

```
"ATGCGTACGAAATTACTATAA" & "3"
```

`repairMutation` trả về

```
"ATGCGTCCCAAATTCTATAA"
```

3.8 Tác vụ khác

3.8.1 Đảo ngược Gene

Yêu cầu 14 (0.5 điểm): Hiện thực hàm **reverseGene** để đảo ngược chuỗi gene hiện tại. Thông tin chi tiết:

- **Khai báo hàm:** `char* reverseGene(struct Gene gene)`
- **Tham số đầu vào:**
  - **gene (struct Gene)**: đối tượng gene cần được đảo ngược.
- **Trả về:**
  - Chuỗi DNA đã được đảo ngược từ `org_seq`.

Ví dụ 3.11

- Với chuỗi gene

```
"ATGCGTACGTAA"
```

`reverseGene` trả về

```
"AATGCATGCGTA"
```

- Với chuỗi gene

```
"ATGCGTACGAAATTACTATAA"
```

`reverseGene` trả về

```
"AATATCATTAAAGCATGCGTA"
```

3.8.2 Gene dài nhất/ngắn nhất

Yêu cầu 15 (0.4 điểm): Hiện thực hàm **longestGene** để in ra thông tin gene có độ dài lớn nhất trong bảng. Thông tin chi tiết:

- **Khai báo hàm:** `void longestGene(struct Gene * geneTable, int count)`
- **Tham số đầu vào:**
  - **geneTable (mảng struct Gene)**: bảng thông tin các gene.
  - **count (int)**: số lượng gene trong bảng.
- **Yêu cầu:**
  - Tìm gene có độ dài lớn nhất trong bảng và in thông tin ra theo format:

```
NAME: GENE_NAME
DNA: GENE_SEQ
```

  - **Lưu ý:** Nếu có nhiều hơn một gene có cùng độ dài lớn nhất, cần in ra toàn bộ.

Yêu cầu 16 (0.4 điểm): Hiện thực hàm **shortestGene** để in ra thông tin gene có độ dài ngắn nhất trong bảng. Thông tin chi tiết:

- **Khai báo hàm:** `void shortestGene(struct Gene * geneTable, int count)`
- **Tham số đầu vào:**
  - **geneTable (mảng struct Gene)**: bảng thông tin các gene.
  - **count (int)**: số lượng gene trong bảng.
- **Yêu cầu:**
  - Tìm gene có độ dài ngắn nhất trong bảng và in thông tin ra theo format:

    ```
    NAME: GENE_NAME
    DNA: GENE_SEQ
    ```

  - **Lưu ý:** Nếu có nhiều hơn một gene có cùng độ dài ngắn nhất, cần in ra toàn bộ.

3.8.3 In quá trình biến đổi của DNA

Yêu cầu 17 (0.4 điểm): Hiện thực hàm **DNAPipeline** để in ra toàn bộ quy trình phiên mã DNA của một gene. Thông tin chi tiết:

- **Khai báo hàm:** `void DNAPipeline(struct Gene gene)`
- **Tham số đầu vào:**
  - **gene (struct Gene)**: đối tượng gene cần in.
- **Yêu cầu:**
  - In ra chuỗi DNA gốc, chuỗi bổ sung và chuỗi mRNA theo format:

    ```
    DNA: DNA's original sequence
    DNA-COM: DNA's complement sequence
    mRNA: DNA's transcription sequence
    ```

    ```
    DNA: ATGCGTACGTAA
    DNA-COM: TACGCATGCATT
    mRNA: AUGCGUACGUAA
    ```

3.9 Lệnh chương trình

Yêu cầu 18 (0.5 điểm): Hiện thực hàm **getActionByCommand** để thực hiện các tác vụ cụ thể được yêu cầu trên bảng gene. Thông tin chi tiết:

- **Khai báo hàm:** `void getActionByCommand(enum commandType command, struct Gene * geneTable, int count)`
- **Tham số đầu vào:**
  - **command (enum commandType)**: loại tác vụ cần thực hiện.
  - **geneTable (mảng struct Gene)**: bảng thông tin các gene.
  - **count (int)**: số lượng gene trong bảng.
- **Yêu cầu:**
  - Xác định tác vụ cần thực hiện dựa trên giá trị của `command`, sau đó gọi hàm tương ứng để áp dụng lên toàn bộ danh sách gene đầu vào.
  - Các trường chuỗi trong `struct Gene`, ngoại trừ chuỗi DNA gốc, mặc định có giá trị `""` (rỗng) trước khi thực hiện các thao tác.
  - Đối với lệnh `MUTATION`, sử dụng bảng Gene cố định được cung cấp để so sánh với các gene trong `geneTable`. Tham khảo ví dụ 3.12 và thêm nguyên mã minh họa trong ví dụ đó vào mã chương trình.

Bảng gene sau được dùng cho ví dụ của **Yêu cầu 18**:

No.	Gene Name	DNA Sequence
1	G1	ATGCGTTAA
2	G2	ATGTTTCCCTAA
3	G3	ATGAAACTAGTTTGA

Bảng 3: A gene table with N = 3

Ví dụ 3.12

```
void getActionByCommand(enum commandType command, struct Gene *
    geneTable, int count) {

    struct Gene com_mutation_geneTable[10] = {
        {"G_1", "ATGCGTTAA", 9, "", "", ""},
        {"G_2", "ATGAAACTAGTTTGA", 15, "", "", ""},
        {"G_3", "ATGAAACTATTTGGGTGA", 18, "", "", ""},
        {"G_4", "ATGCGTGGCTAATAG", 15, "", "", ""},
        {"G_5", "ATGCGCTGA", 9, "", "", ""},
        {"G_6", "ATGCCGTAATAG", 12, "", "", ""},
        {"G_7", "ATGTTTAAACCCGGGTAA", 18, "", "", ""},
        {"G_8", "ATGCGATCGCTAG", 13, "", "", ""},
        {"G_9", "ATGAAATTTCCCGGGTGA", 18, "", "", ""},
        {"G_10", "ATGACTGACTAA", 12, "", "", ""}
    };

    //STUDENT CODE
    ...
}
```

Ví dụ 3.13

- Với lệnh `SANITIZE`, `getActionByCommand(SANITIZE, geneTable, 3)` sẽ in ra:

```
ATGCGTTAA
ATGTTTCCCTAA
ATGAAACTAGTTTGA
```

- Với lệnh `CHECK`, `getActionByCommand(CHECK, geneTable, 3)` sẽ in ra:

```
VALID GENE
VALID GENE
VALID GENE
```

- Với lệnh `COMPLEMENT`, `getActionByCommand(COMPLEMENT, geneTable, 3)` sẽ in ra:

```
VALID GENE
TACGCAATT
VALID GENE
TACAAAGGGATT
VALID GENE
TACTTTGATCAAACT
```

- Với lệnh `MRNA`, `getActionByCommand(MRNA, geneTable, 3)` sẽ in ra:

```
VALID GENE
AUGCGUUAA
VALID GENE
AUGUUUCCCUAA
VALID GENE
AUGAAACUAGUUUGA
```

Ví dụ 3.14

- Với lệnh `ENCODE`, `getActionByCommand(ENCODE, geneTable, 3)` sẽ in ra:

```
VALID GENE
"655231674420844420"
VALID GENE
"6552318457126745566745568444 20"
VALID GENE
"655231654420674556655009845712844606"
```

- Với lệnh `DECODE`, `getActionByCommand(DECODE, geneTable, 3)` sẽ in ra:

```
ATGCGTTAA
ATGTTTCCCTAA
ATGAAACTAGTTTGA
```

- Với lệnh `PRINT`, `getActionByCommand(PRINT, geneTable, 3)` sẽ in ra:

No.	Gene Name	DNA Sequence
1	G1	ATGCGTTAA
2	G2	ATGTTTCCCTAA
3	G3	ATGAAACTAGTTTGA

- Với lệnh `MUTATION`, `getActionByCommand(MUTATION, geneTable, 3)` sẽ in ra:

```
Gene G1: NORMAL
Gene G2: MUTATED
Gene G3: NORMAL
```

Note: Nếu là gene đột biến, in ra "Gene GX: MUTATED"

- Với lệnh `REPAIR`, `getActionByCommand(REPAIR, geneTable, 3)` sẽ in ra:

```
Gene G1: NORMAL
Gene G2: NORMAL
Gene G3: NORMAL
```

Note: Nếu gene được chỉnh sửa, in ra "Gene GX: chuỗi sau khi chỉnh sửa"

Ví dụ 3.15

- Với lệnh `REVERSE`, `getActionByCommand(REVERSE, geneTable, 3)` sẽ in ra:

```
AATTGCGTA
AATCCTTTGACA
AGTTTGATCTTTGTA
```

- Với lệnh `LONGEST`, `getActionByCommand(LONGEST, geneTable, 3)` sẽ in ra:

```
NAME: G3
DNA: ATGAAACTAGTTTGA
```

- Với lệnh `SHORTEST`, `getActionByCommand(SHORTEST, geneTable, 3)` sẽ in ra:

```
NAME: G1
DNA: ATGCGTTAA
```

- Với lệnh `PIPELINE`, `getActionByCommand(PIPELINE, geneTable, 3)` sẽ in ra:

```
DNA: ATGCGTTAA
DNA-COM: TACGCAATT
mRNA: AUGCGUUAA

DNA: ATGTTTCCCTAA
DNA-COM: TACAAAGGGATT
mRNA: AUGUUUCCCUAA

DNA: ATGAAACTAGTTTGA
DNA-COM: TACTTTGATCAAACT
mRNA: AUGAAACUAGUUUGA
```

- Với lệnh `INVALID`, `getActionByCommand(INVALID, geneTable, 3)` sẽ in ra:

```
INVALID COMMAND
```

3.10 Kiểm tra tất cả các hàm

Yêu cầu 19 (0.9 điểm): Một số testcase sẽ kiểm tra tổng hợp các hàm ở phía trên.

4 Nộp bài

Sinh viên tải các các tập tin sau từ trang site của môn học:

genevilx.c	Mã nguồn khởi tạo
NMLT_GENEVILX_Assignment.pdf	File mô tả nội dung bài tập lớn

File `genevilx.c` là mã nguồn khởi tạo. Sinh viên phải sử dụng mã nguồn này để viết phần hiện thực nằm giữa 2 dòng sau:

- // —— **Begin: Student Answer** ——
- // —— **End: Student Answer** ——

Khi nộp bài, sinh viên nộp bài trên site LMS của môn học. Sinh viên điền code bài tập lớn giống như các bài thực hành khác. Nội dung điền vào sẽ là phần code sinh viên hiện thực nằm giữa 2 dòng trên. Sinh viên không được phép include bất kỳ thư viện nào ngoài các thư viện đã có sẵn trong mã nguồn khởi tạo. Sinh viên được cung cấp các nơi nộp bài:

- **Nơi nộp bài thử**: Sinh viên nộp bài làm và được chấm trên 5 testcases để kiểm tra các lỗi cú pháp, lỗi logic cơ bản có thể có của bài làm sinh viên. Sinh viên được phép nộp bài vô số lần ở đây
- **Nơi nộp bài chính thức**: sẽ thông báo sau.

Trong mỗi phần trên, ngoại trừ **Nơi nộp bài thử**, sinh viên có tối đa **10** lần làm bài. Đối với mỗi lần nộp bài, sinh viên có **10 phút** để nộp code và kiểm tra. Chỉ có lần nhấn "Kiểm tra" đầu tiên là được tính điểm, các lần sau sẽ không được lấy điểm. Kết quả bài làm chỉ hiển thị sau khi bạn nhấn nút "Hoàn thành bài làm". Điểm cao nhất trong các lần làm bài sẽ được lấy làm điểm cho phần đó.

Thời hạn nộp bài được công bố tại nơi nộp bài trong site nêu trên. Đến thời hạn nộp bài, đường liên kết sẽ tự động khoá nên sinh viên sẽ không thể nộp chậm. Sinh viên nên hoàn thành và nộp bài sớm. Nếu sinh viên đợi gần đến thời hạn mới nộp mà hệ thống bị quá tải thì sinh viên sẽ tự chịu trách nhiệm.

Sinh viên phải kiểm tra chương trình của mình trên MinGW và nơi nộp bài thử trước khi nộp.

5 Harmony cho Bài tập lớn

Bài kiểm tra cuối kì của môn học sẽ có một số câu hỏi Harmony với nội dung của BTL. Giả sử điểm BTL mà sinh viên đạt được là **a** (theo thang điểm 10), tổng điểm các câu hỏi Harmony **b** (theo thang điểm 5). Gọi **x** là điểm của BTL sau khi Harmony, cũng là điểm BTL cuối cùng của sinh viên. Các câu hỏi cuối kì sẽ được Harmony với 50% điểm của BTL theo công thức sau:

- Nếu $a = 0$ hoặc $b = 0$ thì $x = 0$
- Nếu a và b đều khác 0 thì

$$x = \frac{a}{2} + HARM(\frac{a}{2}, b)$$

Trong đó:

$$HARM(x, y) = \frac{2xy}{x + y}$$

Sinh viên phải giải quyết BTL bằng khả năng của chính mình. Nếu sinh viên gian lận trong BTL, sinh viên sẽ không thể trả lời câu hỏi Harmony và nhận điểm 0 cho BTL.

Sinh viên **phải** chú ý làm câu hỏi Harmony trong bài kiểm tra cuối kỳ. Các trường hợp không làm sẽ tính là 0 điểm cho BTL, và bị không đạt cho môn học. **Không chấp nhận giải thích và không có ngoại lệ.**

6 Quy định và xử lý gian lận

Bài tập lớn phải được sinh viên TỰ LÀM. Sinh viên sẽ bị coi là gian lận nếu:

- Có sự giống nhau bất thường giữa mã nguồn của các bài nộp. Trong trường hợp này, TẤT CẢ các bài nộp đều bị coi là gian lận. Do vậy sinh viên phải bảo vệ mã nguồn bài tập lớn của mình.
- Sinh viên không hiểu mã nguồn do chính mình viết, trừ những phần mã được cung cấp sẵn trong chương trình khởi tạo. Sinh viên có thể tham khảo từ bất kỳ nguồn tài liệu nào, tuy nhiên phải đảm bảo rằng mình hiểu rõ ý nghĩa của tất cả những dòng lệnh mà mình viết. Trong trường hợp không hiểu rõ mã nguồn của nơi mình tham khảo, sinh viên được đặc biệt cảnh báo là KHÔNG ĐƯỢC sử dụng mã nguồn này; thay vào đó nên sử dụng những gì đã được học để viết chương trình.

- Nộp nhầm bài của sinh viên khác trên tài khoản cá nhân của mình.

Trong trường hợp bị kết luận là gian lận, sinh viên sẽ bị điểm 0 cho toàn bộ môn học (không chỉ bài tập lớn).

KHÔNG CHẤP NHẬN BẤT KỲ GIẢI THÍCH NÀO VÀ KHÔNG CÓ BẤT KỲ NGOẠI LỆ NÀO!

Sau mỗi bài tập lớn được nộp, sẽ có một số sinh viên được gọi phỏng vấn ngẫu nhiên để chứng minh rằng bài tập lớn vừa được nộp là do chính mình làm.

Một số quy định khác:

- Mọi quyết định của giảng viên phụ trách bài tập lớn là quyết định cuối cùng.
- Sinh viên không được cung cấp testcase sau khi chấm bài.
- Nội dung Bài tập lớn sẽ được Harmony với một câu hỏi trong bài kiểm tra với nội dung tương tự.

7 Thay đổi so với phiên bản trước

- Trong Mục 3.9, bổ sung thêm thông tin liên quan đến lệnh MUTATION, đồng thời chỉnh sửa lại một số ví dụ hiện có cho phù hợp.

——————HẾT——————