C++프로그래밍및실습

DNA 해독기

최종 보고서

제출일자:2023.12.24
제출자명:백성준
제출자학번:175989

1. 프로젝트 목표

1) 배경 및 필요성

"Chat gpt"와 같은 대화형 인공지능의 출현으로 세계의 기술적 환경은 크게 변화하였다. 평범한 사람들도 이제는 그 변화를 피부로 느끼고 있다. 당장 이러한 대화형 인공지능을 사용해본 사람들은 그 혁신성과 동시에 위기감을 함께 느낄 수 있다. 머지않아 대부분의 단순 노동은 대체될 것이며, 가장 인간적이라 생각한 부분도 대체될 것이다. 그리고 실제로 지금도 대체가 일어나고 있다.

이러한 변화 속에서 인공지능 또는 자동화 기술을 다양한 학문과 연계한다면 상당한 경쟁력 또는 필수적인 기술을 획득할 수 있을 것이다. 본 프로젝트에서는 생물학에서의 서열분석 자동화를 목표로 한다.

2) 프로젝트 목표

이 프로젝트에서는 생명의 가장 기초적이지만 근본적인 DNA 염기서열을 자동으로 처리하는 프로그램을 다루고자 한다. DNA 염기서열에서 ORF(Open Reading Frame)을 찾아내고 CDS(Coding sequence)를 최종적으로 얻어내는 것을 목표로 한다. 또한, 이 CDS를 단백질로 변환하고 kozak score를 매기는 작업까지 수행한다. 이 때, 사람의 DNA라 가정하며 가장 자주 발견되는 consensus sequence를 따른다고 가정한다. 또한, codon table은 보편적인 table을 사용한다.

3) 차별점

기존의 염기서열 정보를 단백질 서열로 변환하는 프로그램에는 단순 변환만 존재하였다. 여기에서 더 나아가 DNA에서 얻어낸 단백질의 정보가 저장된 ORF(Open Reading Frame)에서 intron을 제거해낼 수 있다. 또한, 얻어낸 서열 정보의 분석(kozak score 측정)을 진행할 수 있다.

3) 배경지식

사람의 유전자는 A,T,G,C 네 가지의 염기를 가지고 있다. 이 염기가 3 개씩(triplet) 모여서 단백질 정보를 이루는 것을 codon이라 하고, codon은 20 개 단

백질 정보를 담고 있다. 각 codon에 하나의 단백질이 상응하며 단백질의 시작이 되는 start codon(ATG)과 stop codon stop codon(TGA, TAA, TAG)이 각각 존재한다. Start codon에서 stop codon 까지를 ORF(Open Reading Frame)라 한다.

DNA는 RNA로 변환되어 splicing이라는 intron이 제거되는 과정을 거친다. Intron이 제거된 RNA가 ribosome에서 단백질로 변환된다. 이때의 RNA서열에 상응하는 DNA를 CDS(Coding Sequence)라 한다.

Kozak strength는 mRNA에서 단백질이 전사되는 속도에 영향을 미친다. 클수록 더 빠르게 단백질의 전사가 시작된다. 이는 곧, 단백질 양의 증가에도 영향을 끼친다. 따라서, Kozak strength는 서열의 특정 부분이 실제로 작동하는 CDS인지 확인할 수 있는 지표가 된다.

2. 기능 계획

2-1. 기능 계획

1) 사용자의 이름과 염기서열 정보, 찾을 ORF 범위 저장

- 사용자에게서 이름과 염기서열, 찾고자 하는 ORF의 크기 범위를 받는다. 이후, 확인을 위해 프로그램 내에 입력된 값들을 출력한다.

2) 염기서열의 정보를 분석

- 사용자에게서 받은 염기서열을 객체에 저장하여 단백질 형태로 가공하고 분석하여 최종적으로 파일로 출력한다.

(1) 염기서열에서 ORF를 저장

- 사용자의 서열을 6 가지의 reading frame으로 각각 객체에 하나씩 저장한다. 각 객체에 저장된 서열에서 start codon(ATG)로 시작하고 stop codon(TGA, TAA, TAG)로 끝나는 모든 서열(ORF)을 따로 저장한다. 진행 과정 확인을 위해 출력도 진행한다.

(2) 얻어낸 ORF에서 intron을 제거

- [GTA][ATG]로 시작하며 [TTT][TTT][TT][CAG]로 끝나는 서열(가장 흔한 intron)을 제거한다. 진행 과정 확인을 위해 출력도 진행한다.

(3) Intron이 제거된 CDS를 단백질 형태의 정보로 변환

- 인트론을 제거한 CDS를 DNA 형태에서 단백질 정보 형태로 변환한다. 진행 과정 확인을 위해 출력도 진행한다.

(4) Kozak sequence score를 계산

- 얻어낸 각 CDS의 Kozak sequence score를 계산하여 출력한다.

(5) 분석한 결과를 파일로 출력

- 얻어낸 단백질 정보로 바뀐 CDS를 원래의 서열과 비교할 수 있도록 파일로 출력한다.

2-2. 함수 계획

3. 기능구현

3-1. 기능 구현

(1) 사용자의 이름과 염기서열 정보, 찾을 ORF 범위 저장

- 입출력

string name : 사용자의 이름을 최초로 입력 받는 변수.

string sequence : 사용자의 서열을 입력 받는 변수.

ifstream sequence_file : 사용자가 입력한 저장한 파일에서 서열을 가져오기 위함.

User user : 사용자의 이름, 서열, 찾고자 하는 ORF의 크기 범위를 저장하는 객체.

user.range1 : 유저가 찾고자 하는 ORF를 받는 변수. 범위의 작은 수를 넣기 위함.

user.range2 : 유저가 찾고자 하는 ORF를 받는 변수. 범위의 큰 수를 넣기 위함.

void User::SetName(string name) : user 객체 내의 이름변수를 초기화

void User::SetSequence(string sequence) : user 객체 내의 원본 서열을 초기화

string GetName() : user 객체에 저장된 이름을 반환

string GetSequence() : user 객체 내에 저장된 서열을 반환

- 설명

사용자가 입력한 이름, 서열, 범위를 user 객체 내에 저장하고 확인을 위해 출력한다. 이 때, 서열 파일이 없으면 에러 메시지를 출력하고 범위를 지정할 때에는 range1에 작은 수가 들어가게 된다.

- 적용된 배운 내용

클래스, 예외처리, if 문, get/set 메소드, string, 파일 입출력, 비교연산자, cin/cout, 객체 함수/변수 접근

- 코드 스크린샷

```
string name;                              // 유저의 이름
string sequence;                          // 유저의 서열
ifstream sequence_file("sequence.txt");   // 유저의 서열 파일
User user;  // User의 이름, 서열, ORF범위, 다른 reading frame서열을
// 저장하는 클래스

// 이름입력, 테스트를 위해 임의의 이름과 서열, 범위를 사용했습니다.
cout << "이름: " << endl;
cin >> name;
// name = "김철수";//테스트용

// 서열 입력
cout << "DNA서열:" << endl;
try {
    if (!sequence_file)  // 오류 발생 시 메세지//이상이 없으면 서열 입력
    {
        throw invalid_argument("파일을 불러오지 못했습니다.");
    } else {
        sequence_file >> sequence;
    }
    sequence_file.close();
} catch (invalid_argument& e) {
    cout << "에러: " << e.what() << endl;
}
```

```
// ORF 범위 입력
cout << "찾고자하는 ORF의 크기 : " << endl;
cin >> user.range1 >> user.range2;  // ORF의 범위를 입력
// user.range1 = 1;
// user.range2 = 100;
//  range1에 더 큰 수를 입력한 경우, 둘을 바꿔준다.
if (user.range2 < user.range1) {
    int temp_length = user.range1;
    user.range1 = user.range2;
    user.range2 = temp_length;
}
user.SetName(name);           // 객체내의 이름초기화
user.SetSequence(sequence);   // 객체내의 서열 초기화
// 저장한 값들을 출력
cout << "이름 :" << user.GetName() << endl;
cout << "서열 : " << user.GetSequence() << endl;
cout << "범위 : " << user.range1 << "," << user.range2 << endl;
```

(2) 염기서열에서 ORF를 저장

- 입출력

void User::FrameSetting() : 6 가지의 가능한 reading frame으로 서열을 변환하여 user 객체 내의 멤버 변수에 저장하는 함수.

Orf user_seq1, user_seq2, user_seq3, user_seq4, user_seq5, user_seq6 : reading frame 6 가지 경우를 각각 넣어 주기 위한 객체. 각 객체 에서 서열에 대한 분석을 진행

string User::GetDna1(), string User::GetDna2(), string User::GetDna3(), string User::GetRDna1(), string User::GetRDna2(),string User::GetRDna3() : 6 가지 가능한 reading frame으로 변환되어 저장된 string 변수를 출력하는 함수. Orf 클래스의 생성을 할 때 생성자를 사용하여 객체 변수 클래스를 초기화 하기 위해 사용.

void Orf::TransferSeq() : 받아온 reading frame이 지정된 서열을 분석하기 쉽게 3 개씩 묶어서 스트링 벡터화 하는 함수.

vector<string> orf1 : 3 개씩 묶인 서열을 각 string에 저장하고 있는 Orf 클래스의 멤버 변수.
void Orf::IndexFinder() : ORF를 탐색하는 데에 필요한 start codon과 stop codon을 객체의 멤버 변수에 저장하는 함수.

void Orf::OrfFinder(User user) : user 객체에서 저장된 찾고자 하는 범위를 가져와서 ORF를 찾은 후 이를 스트링 이중 벡터에 저장하는 함수. 이때, 이중 벡터에 저장된 스트링 벡터마다의 인덱스도 함께 저장.

vector<vector<string>> complete_orf : 찾은 ORF를 스트링 이중 벡터에 저장해 놓은 Orf의 클래스 멤버 변수.
- 설명

6 가지의 가능한 reading frame으로 서열을 변환하여 user 객체 내의 멤버 변수에 저장하고 확인을 위해 이들을 출력한다.

ORF를 찾기 위해 다음과 같은 과정을 거친다. 6 가지의 reading frame으로 나눈 6 개의 서열을 각각 Orf 객체에 하나씩 할당한다. 이 서열을 string 벡터로 저장하여 3 개씩 한 string에 넣어서 분석을 용이하게 한다. 이를 가지고 Orf의 멤버 함수를 사용하여 각 서열의 start codon과 stop codon을 찾아서 저장하고 이를 바탕으로 ORF를 찾는다. 찾아진 ORF를 Orf 클래스의 complete_orf 멤버 변수에 저장한다. 이때, 저장된 각 ORF의 인덱스도 저장하여 놓는다(최종 출력에 사용하기 위함).

마지막으로 확인을 위해 저장된 ORF(complete_orf 멤버변수)를 출력한다. 탐색된 ORF가 없으면 메시지를 출력한다.

- 적용된 배운 내용

클래스, cin/cout, get/set 메소드, 벡터, 배열에의 접근, for 문, 비교연산자, 증감연산자, 초기화 if else문, 생성자, 객체 함수/변수 접근

- 코드 스크린샷

```
// 6가지의 가능한 readingframe으로 서열을 저장하는 함수
user.FrameSetting();
// 저장이 되었는지 확인하가 위해 출력하는 코드
cout << "dna1: " << user.GetDna1() << endl;
cout << "dna2: " << user.GetDna2() << endl;
cout << "dna3: " << user.GetDna3() << endl;
cout << "dna4: " << user.GetRDna1() << endl;
cout << "dna5: " << user.GetRDna2() << endl;
cout << "dna6: " << user.GetRDna3() << endl << endl;
```

```
// 각 6개의 서열에 대해 같은 방법으로 분석, 6번 반복
Orf user_seq1(user.GetDna1());  // Orf객체에 분석할 서열 전달및 Orf 객체 생성
user_seq1.TransferSeq();  // 서열의 스트링 벡터화
cout << "벡터화dna1: " << endl;
for (int i = 0; i < user_seq1.orf1.size();
    i++)  // 스트링 벡터화된 서열을 출력.
{
 cout << user_seq1.orf1[i] << " ";
}
cout << endl;

user_seq1.IndexFinder();  // start codon과 stop codon에 해당하는 인덱스 찾기
user_seq1.OrfFinder(user);  // orf 찾기

// 완료한 orf 출력
cout << "찾은 ORF: " << endl;
if (user_seq1.complete_orf.empty())  // 벡터비어있으면 관련 안내 출력
{
 cout << "ORF가 탐색되지 않았습니다." << endl;
} else  // 벡터가 차있다면 ORF를 출력
{       // 스트링 이중벡터에 존재하는 모든것을 출력.
   //찾은 ORF를 모두 출력
 for (int k = 0; k < user_seq1.complete_orf.size(); k++) {
  for (int i = 0; i < user_seq1.complete_orf[k].size(); i++) {
   cout << user_seq1.complete_orf[k][i] << " ";
  }
  cout << endl;
 }
}
cout << endl << endl;
```

```
Orf user_seq2(user.GetDna2());  // Orf객체에 분석할 서열 전달
user_seq2.TransferSeq();        // 서열의 스트링 벡터화
cout << "벡터화dna2: " << endl;
for (int i = 0; i < user_seq2.orf1.size();
    i++)  // 스트링 벡터화된 서열을 출력.
{
 cout << user_seq2.orf1[i] << " ";
}
cout << endl;

user_seq2.IndexFinder();  // start codon과 stop codon에 해당하는 인덱스 찾기
user_seq2.OrfFinder(user);  // orf 찾기
cout << "찾은 ORF: " << endl;
if (user_seq2.complete_orf.empty()) {
 cout << "ORF가 탐색되지 않았습니다." << endl;
} else  // 벡터가 차있다면 ORF를 출력
{
 for (int k = 0; k < user_seq2.complete_orf.size(); k++) {
  for (int i = 0; i < user_seq2.complete_orf[k].size(); i++) {
   cout << user_seq2.complete_orf[k][i] << " ";
  }
  cout << endl;
 }
}
cout << endl << endl;
```

```
Orf user_seq3(user.GetDna3());  // Orf객체에 분석할 서열 전달
user_seq3.TransferSeq();        // 서열의 스트링 벡터화
cout << "벡터화dna3: " << endl;
for (int i = 0; i < user_seq3.orf1.size();
    i++)  // 스트링 벡터화된 서열을 출력.
{
 cout << user_seq3.orf1[i] << " ";
}
cout << endl;

user_seq3.IndexFinder();  // start codon과 stop codon에 해당하는 인덱스 찾기
user_seq3.OrfFinder(user);  // orf 찾기
cout << "찾은 ORF: " << endl;
if (user_seq3.complete_orf.empty()) {
 cout << "ORF가 탐색되지 않았습니다." << endl;
} else  // 벡터가 차있다면 ORF를 출력
{
 for (int k = 0; k < user_seq3.complete_orf.size(); k++) {
  for (int i = 0; i < user_seq3.complete_orf[k].size(); i++) {
   cout << user_seq3.complete_orf[k][i] << " ";
  }
  cout << endl;
 }
}
cout << endl << endl;
```

```
Orf user_seq4(user.GetRDna1());  // Orf객체에 분석할 서열 전달
user_seq4.TransferSeq();         // 서열의 스트링 벡터화
cout << "벡터화dna4: " << endl;
for (int i = 0; i < user_seq4.orf1.size();
     i++)  // 스트링 벡터화된 서열을 출력.
{
  cout << user_seq4.orf1[i] << " ";
}
cout << endl;
user_seq4.IndexFinder();  // start codon과 stop codon에 해당하는 인덱스 찾기
user_seq4.OrfFinder(user);  // orf 찾기
cout << "찾은 ORF: " << endl;
if (user_seq4.complete_orf.empty()) {
  cout << "ORF가 탐색되지 않았습니다." << endl;
} else  // 벡터가 차있다면 ORF를 출력
{
  for (int k = 0; k < user_seq4.complete_orf.size(); k++) {
    for (int i = 0; i < user_seq4.complete_orf[k].size(); i++) {
      cout << user_seq4.complete_orf[k][i] << " ";
    }
    cout << endl;
  }
}
cout << endl << endl;
```

```
Orf user_seq5(user.GetRDna2());  // Orf객체에 분석할 서열 전달
user_seq5.TransferSeq();         // 서열의 스트링 벡터화
cout << "벡터화dna5: " << endl;
for (int i = 0; i < user_seq5.orf1.size();
     i++)  // 스트링 벡터화된 서열을 출력.
{
  cout << user_seq5.orf1[i] << " ";
}
cout << endl;
user_seq5.IndexFinder();  // start codon과 stop codon에 해당하는 인덱스 찾기
user_seq5.OrfFinder(user);  // orf 찾기
cout << "찾은 ORF: " << endl;
if (user_seq5.complete_orf.empty()) {
  cout << "ORF가 탐색되지 않았습니다." << endl;
} else  // 벡터가 차있다면 ORF를 출력
{
  for (int k = 0; k < user_seq5.complete_orf.size(); k++) {
    for (int i = 0; i < user_seq5.complete_orf[k].size(); i++) {
      cout << user_seq5.complete_orf[k][i] << " ";
    }
    cout << endl;
  }
}
cout << endl << endl;
```

```
Orf user_seq6(user.GetRDna3());  // Orf객체에 분석할 서열 전달
user_seq6.TransferSeq();         // 서열의 스트링 벡터화
cout << "벡터화dna6: " << endl;
for (int i = 0; i < user_seq6.orf1.size();
     i++)  // 스트링 벡터화된 서열을 출력.
{
  cout << user_seq6.orf1[i] << " ";
}
cout << endl;
user_seq6.IndexFinder();  // start codon과 stop codon에 해당하는 인덱스 찾기
user_seq6.OrfFinder(user);  // orf 찾기
cout << "찾은 ORF: " << endl;
if (user_seq6.complete_orf.empty()) {
  cout << "ORF가 탐색되지 않았습니다." << endl;
} else  // 벡터가 차있다면 ORF를 출력
{
  for (int k = 0; k < user_seq6.complete_orf.size(); k++) {
    for (int i = 0; i < user_seq6.complete_orf[k].size(); i++) {
      cout << user_seq6.complete_orf[k][i] << " ";
    }
    cout << endl;
  }
}
cout << endl << endl;
```

(3) 얻어낸 ORF에서 intron을 제거

- 입출력

Orf user_seq1, user_seq2, user_seq3, user_seq4, user_seq5, user_seq6 : reading frame 6가지 경우를 각각 넣어 주기 위한 객체. 각 객체 에서 서열에 대한 분석을 진행

void Orf::IntronFinder() : 스트링 이중 벡터화된 찾은 ORF에 접근(complete_orf)하여 인트론을 찾으면 해당하는 범위는 인트론을 뜻하는"itr"로 교체하여 새로운 스트링 이중 벡터에 저장한다.

vector<vector<string>> intron_removed : 인트론이 제거된 ORF를 저장하기 위한 변수.

- 설명

앞서 찾아낸 ORF들에 존재하는 inron을 모두 "itr"로 교체하여 인트론임을 나타낸다. 이 결과를 확인을 위해 출력한다. 앞서 찾아낸 ORF가 없어서 분석할 것이 없다면 관련 메시지를 출력한다. 또한, stop codon이라면 그대로 stop codon 표시를 분석을 위해 남겨 놓는다.

- 적용된 배운 내용

벡터, 스트링, if 문, for 문, 비교연산자, 증감연산자, cin/cout, 배열접근, 객체 함수/변수 접근

- 코드 스크린샷

```
// 인트론 제거
cout << "인트론 가공 후: " << endl;
user_seq1.IntronFinder();
if (user_seq1.intron_removed.empty())  // 벡터비어있으면 관련 안내 출력
{
  cout << "ORF가 존재하지 않습니다." << endl;
} else  // 벡터가 차있다면 ORF를 출력
// 스트링 이중벡터에 존재하는 모든것을 출력.
{
  // 찾은 ORF 서열의 수만큼 반복
  for (int k = 0; k < user_seq1.intron_removed.size();
       k++) {  // 각 ORF에서 string 벡터를 출력
    for (int i = 0; i < user_seq1.intron_removed[k].size(); i++)  //
    {
      cout << user_seq1.intron_removed[k][i] << " ";
    }
    cout << endl;
  }
}
```

```
// 인트론 제거
cout << "인트론 가공 후: " << endl;
user_seq2.IntronFinder();
if (user_seq2.intron_removed.empty())  // 벡터비어있으면 관련 안내 출력
{
  cout << "ORF가 존재하지 않습니다." << endl;
} else  // 벡터가 차있다면 ORF를 출력
{       // 스트링 이중벡터에 존재하는 모든것을 출력.

  // 찾은 ORF 서열의 수만큼 반복
  for (int k = 0; k < user_seq2.intron_removed.size();
       k++) {  // 각 ORF에서 string 벡터를 출력
    for (int i = 0; i < user_seq2.intron_removed[k].size(); i++)  //
    {
      cout << user_seq2.intron_removed[k][i] << " ";
    }
    cout << endl;
  }
}
```

```
// 인트론 제거
cout << "인트론 가공 후: " << endl;
user_seq3.IntronFinder();
if (user_seq3.intron_removed.empty())  // 벡터비어있으면 관련 안내 출력
{
  cout << "ORF가 존재하지 않습니다." << endl;
} else  // 벡터가 차있다면 ORF를 출력
{       // 스트링 이중벡터에 존재하는 모든것을 출력.

  // 찾은 ORF 서열의 수만큼 반복
  for (int k = 0; k < user_seq3.intron_removed.size();
       k++) {  // 각 ORF에서 string 벡터를 출력
    for (int i = 0; i < user_seq3.intron_removed[k].size(); i++) {
      cout << user_seq3.intron_removed[k][i] << " ";
    }
    cout << endl;
  }
}
```

```
// 인트론 제거
cout << "인트론 가공 후: " << endl;
user_seq4.IntronFinder();
if (user_seq4.intron_removed.empty())  // 벡터비어있으면 관련 안내 출력
{
  cout << "ORF가 존재하지 않습니다." << endl;
} else  // 벡터가 차있다면 ORF를 출력
{       // 스트링 이중벡터에 존재하는 모든것을 출력.

  // 찾은 ORF 서열의 수만큼 반복
  for (int k = 0; k < user_seq4.intron_removed.size();
       k++) {  // 각 ORF에서 string 벡터를 출력
    for (int i = 0; i < user_seq4.intron_removed[k].size(); i++)  //
    {
      cout << user_seq4.intron_removed[k][i] << " ";
    }
    cout << endl;
  }
}
```

```
// 인트론 제거
cout << "인트론 가공 후: " << endl;
user_seq5.IntronFinder();
if (user_seq5.intron_removed.empty())  // 벡터비어있으면 관련 안내 출력
{
  cout << "ORF가 존재하지 않습니다." << endl;
} else  // 벡터가 차있다면 ORF를 출력
{       // 스트링 이중벡터에 존재하는 모든것을 출력.

  // 찾은 ORF 서열의 수만큼 반복
  for (int k = 0; k < user_seq5.intron_removed.size();
       k++) {  // 각 ORF에서 string 벡터를 출력
    for (int i = 0; i < user_seq5.intron_removed[k].size(); i++)  //
    {
      cout << user_seq5.intron_removed[k][i] << " ";
    }
    cout << endl;
  }
}
```

```
// 인트론 제거
cout << "인트론 가공 후: " << endl;
user_seq6.IntronFinder();
if (user_seq6.intron_removed.empty())  // 벡터비어있으면 관련 안내 출력
{
  cout << "ORF가 존재하지 않습니다." << endl;
} else  // 벡터가 차있다면 ORF를 출력
{       // 스트링 이중벡터에 존재하는 모든것을 출력.

  // 찾은 ORF 서열의 수만큼 반복
  for (int k = 0; k < user_seq6.intron_removed.size();
       k++) {  // 각 ORF에서 string 벡터를 출력
    for (int i = 0; i < user_seq6.intron_removed[k].size(); i++)  //
    {
      cout << user_seq6.intron_removed[k][i] << " ";
    }
    cout << endl;
  }
}
```

(4) Intron이 제거된 CDS를 단백질 형태의 정보로 변환

- 입출력

Orf user_seq1, user_seq2, user_seq3, user_seq4, user_seq5, user_seq6 : reading frame 6 가지 경우를 각각 넣어 주기 위한 객체. 각 객체 에서 서열에 대한 분석을 진행

void Orf::CodonDecipher() : 인트론 제거된 ORF를 DNA 서열 형태의 정보에서 단백질 형태의 정보로 변화 시키는 함수

vector<vector<string>> protein : CodonDecipher()에 의해 단백질 정보로 변환된 서열들을 저장하기 위한 함수. 단백질 해독의 결과를 확인하기 위한 출력에서도 사용하기 위함.

- 설명

인트론을 제거한 6개의 Orf 객체의 ORF에 저장된 DNA 염기 서열 정보를 codon table에 따라 단백질 정보 형태로 교체하여 객체 변수에 저장한다. 저장한 단백질 객체 변수를 출력한다.

- 적용된 배운 내용

for 문, 클래스, 벡터, 비교연산자, cin/cout, 증감연산자, 사칙연산자

- 코드 스크린샷

```
// codon 변환 및 확인
user_seq1.CodonDecipher();  // 인트론 제거된 ORF를 단백질화 시키는 함수
cout << "codon 해독 후: " << endl;
if (user_seq1.protein.empty())  // 벡터비어있으면 관련 안내 출력
{
 cout << "단백질 서열이 존재하지 않습니다.";
} else  // 벡터가 차있다면 ORF를 출력
// 스트링 이중벡터에 존재하는 모든것을 출력.
{
 // 찾은 ORF 서열의 수만큼 반복
 for (int k = 0; k < user_seq1.protein.size();
     k++) {  // 각 ORF에서 string 벡터를 출력
  for (int i = 0; i < user_seq1.protein[k].size(); i++)  //
  {
   cout << user_seq1.protein[k][i] << " ";
  }
  cout << endl;
 }
}
```

```
// codon 변환 및 확인
user_seq3.CodonDecipher();  // 인트론 제거된 ORF를 단백질화 시키는 함수
cout << "codon 해독 후: " << endl;
if (user_seq3.protein.empty())  // 벡터비어있으면 관련 안내 출력
{
 cout << "단백질 서열이 존재하지 않습니다.";
} else  // 벡터가 차있다면 ORF를 출력
// 스트링 이중벡터에 존재하는 모든것을 출력.
{
 // 찾은 ORF 서열의 수만큼 반복
 for (int k = 0; k < user_seq3.protein.size();
     k++) {  // 각 ORF에서 string 벡터를 출력
  for (int i = 0; i < user_seq3.protein[k].size(); i++)  //
  {
   cout << user_seq3.protein[k][i] << " ";
  }
  cout << endl;
 }
}
```

```
// codon 변환 및 확인
user_seq2.CodonDecipher();  // 인트론 제거된 ORF를 단백질화 시키는 함수
cout << "codon 해독 후: " << endl;
if (user_seq2.protein.empty())  // 벡터비어있으면 관련 안내 출력
{
 cout << "단백질 서열이 존재하지 않습니다.";
} else  // 벡터가 차있다면 ORF를 출력
// 스트링 이중벡터에 존재하는 모든것을 출력.
{
 // 찾은 ORF 서열의 수만큼 반복
 for (int k = 0; k < user_seq2.protein.size();
     k++) {  // 각 ORF에서 string 벡터를 출력
  for (int i = 0; i < user_seq2.protein[k].size(); i++)  //
  {
   cout << user_seq2.protein[k][i] << " ";
  }
  cout << endl;
 }
}
```

```
// codon 변환 및 확인
user_seq4.CodonDecipher();  // 인트론 제거된 ORF를 단백질화 시키는 함수
cout << "codon 해독 후: " << endl;
if (user_seq4.protein.empty())  // 벡터비어있으면 관련 안내 출력
{
 cout << "단백질 서열이 존재하지 않습니다.";
} else  // 벡터가 차있다면 ORF를 출력
// 스트링 이중벡터에 존재하는 모든것을 출력.
{
 // 찾은 ORF 서열의 수만큼 반복
 for (int k = 0; k < user_seq4.protein.size();
     k++) {  // 각 ORF에서 string 벡터를 출력
  for (int i = 0; i < user_seq4.protein[k].size(); i++)  //
  {
   cout << user_seq4.protein[k][i] << " ";
  }
  cout << endl;
 }
}
```

```
// codon 변환 및 확인
user_seq5.CodonDecipher();  // 인트론 제거된 ORF를 단백질화 시키는 함수
cout << "codon 해독 후: " << endl;
if (user_seq5.protein.empty())  // 벡터비어있으면 관련 안내 출력
{
 cout << "단백질 서열이 존재하지 않습니다.";
} else  // 벡터가 차있다면 ORF를 출력
// 스트링 이중벡터에 존재하는 모든것을 출력.
{
 // 찾은 ORF 서열의 수만큼 반복
 for (int k = 0; k < user_seq5.protein.size();
     k++) {  // 각 ORF에서 string 벡터를 출력
  for (int i = 0; i < user_seq5.protein[k].size(); i++)  //
  {
   cout << user_seq5.protein[k][i] << " ";
  }
  cout << endl;
 }
}
```

```
// codon 변환 및 확인
user_seq6.CodonDecipher();  // 인트론 제거된 ORF를 단백질화 시키는 함수
cout << "codon 해독 후: " << endl;
if (user_seq6.protein.empty())  // 벡터비어있으면 관련 안내 출력
{
 cout << "단백질 서열이 존재하지 않습니다.";
} else  // 벡터가 차있다면 ORF를 출력
// 스트링 이중벡터에 존재하는 모든것을 출력.
{
 // 찾은 ORF 서열의 수만큼 반복
 for (int k = 0; k < user_seq6.protein.size();
     k++) {  // 각 ORF에서 string 벡터를 출력
  for (int i = 0; i < user_seq6.protein[k].size(); i++)  //
  {
   cout << user_seq6.protein[k][i] << " ";
  }
  cout << endl;
 }
}
```

(5) Kozak sequence score를 계산

- 입출력

Orf user_seq1, user_seq2, user_seq3, user_seq4, user_seq5, user_seq6 : reading frame 6 가지 경우를 각각 넣어 주기 위한 객체. 각 객체 에서 서열에 대한 분석을 진행

void Orf::KozakCalculator() : Kozak score를 계산하여 객체 변수 벡터에 입력한다.

vector<int> complete_score : Kozak score가 저장되어 있는 벡터. 출력을 위해 가져옴.

- 설명

ORF의 start codon 주변의 서열을 사용하여 Kozak score를 계산하고 이를 출력한다. Orf 클래스의 멤버함수, KozakCalculator()를 이용하여 각 ORF의 점수를 벡터에 저장하고, 이를 출력한다.

- 적용된 배운 내용

스트링, for문, 클래스, 벡터, cin/cout, 비교연산자, 증감연산자, 객체의 멤버함수, 변수에 접근, 배열 접근

- 코드 스크린샷

```
// kozak score 계산
user_seq1.KozakCalculator();  // 각 ORF의 kozak score를 계산하여 벡터에 저장
// 저장된 벡터를 출력하여 Kozak score를 출력
cout << "Kozak score : ";
for (int i = 0; i < user_seq1.complete_index.size(); i++) {
  // ORF가 너무 짧다는 등의 이유로 너무 계산에 필요한 인덱스에 접근 불가할 때
  if (user_seq1.complete_score[i] == 7777) {//오류 메시지 출력
    cout << "계산불가"
         << " ";
  } else {// 그것이 아니면 벡터(score)를 출력
    cout << user_seq1.complete_score[i] << " ";
  }
}
```

(6) 분석한 결과를 파일로 출력

- 입출력

Orf user_seq1, user_seq2, user_seq3, user_seq4, user_seq5, user_seq6 : reading frame 6 가지 경우를 각각 넣어 주기 위한 객체. 여기서 분석된 단백질 서열을 가져온다.

string file_name

vector<vector<string>> protein : 단백질 정보로 변환된 서열들을 저장하는 함수. 단백질 해독의 결과를 파일에 출력하기 위해 사용.

string file_name : 출력되는 파일을 사용자의 이름으로 출력하기 위해 사용자의 이름을 저장하는 변수.

ofstream result : 단백질 서열과 사용자가 입력했던 서열을 비교하기 쉽게 파일로 출력하기 위한 변수.

- 설명

파일에 가장 먼저 사용자가 입력했던 서열을 출력하고 그 아래에 찾았던 ORF를 하나씩 출력하되, 그 위치를 사용자가 입력했던 서열과 대응시켜 결과를 한눈에 알아보기 쉽게 출력한다. 이 때, ORF 앞에 공란을 추가하여서 위치를 일치시킨다.

- 적용된 배운 내용

for 문, 스트링, 벡터, 비교연산자, 증감연산자, 사칙연산자, 파일 입출력, 범위 기반 for 문

- 코드 스크린샷

```
// 파일로 결과 출력
string file_name =
    user.GetName() +
    ".txt";  // user의 이름으로 텍스트파일 생성하기 위해 이름 변수 설정
ofstream result(file_name,
                ios::out | ios::trunc);  // 분석결과를 출력하기 위함.
string out_sequence = user.GetSequence();
for (char ori : out_sequence)  // 유저에게 받은 서열 출력
{
  result << ori;
}
result << endl;
// 변환한 단백질 서열을 파일로 출력, 사용자가 제공한 서열과 위치가 일치하도록
// 공란을 추가.
// orf가 시작하는 곳과 원래서열과의 관계를 나타내기 위해, 앞만큼 공간을
// 주기위해 공란으로 처리
// user_seq1의 결과를 파일에 출력
for (int a = 0; a < user_seq1.protein.size();
     a++)  // 찾은 ORF의 수만큼 반복
{
  for (int i = 0; i < user_seq1.complete_index[a].start_index; i++) {
    result << "   ";//공란추가
  }
  for (int b = 0; b < user_seq1.protein[a].size(); b++) {
    result << user_seq1.protein[a][b];//단백질 서열출력
  }
  result << endl;
}
```

```
// user_seq2의 결과를 파일에 출력
for (int a = 0; a < user_seq2.protein.size(); a++) {
    result << " ";  // 공란추가
    for (int i = 0; i < user_seq2.complete_index[a].start_index; i++) {
        result << "   ";  // 공란추가
    }
    for (int b = 0; b < user_seq2.protein[a].size(); b++) {
        result << user_seq2.protein[a][b];  // 단백질 서열출력
    }
    result << endl;
}
// user_seq3의 결과를 파일에 출력
for (int a = 0; a < user_seq3.protein.size(); a++) {
    result << "  ";  // 공란추가
    for (int i = 0; i < user_seq3.complete_index[a].start_index; i++) {
        result << "   ";  // 공란추가
    }
    for (int b = 0; b < user_seq3.protein[a].size(); b++) {
        result << user_seq3.protein[a][b];  // 단백질 서열출력
    }
    result << endl;
}
// user_seq4~6은 역방향이기 때문에 stop codon와 서열의 차이만큼 공란추가
// user_seq4의 결과를 파일에 출력
for (int a = 0; a < user_seq4.protein.size(); a++) {
    for (int i = 0; i < out_sequence.length() -
                        3.* (user_seq4.complete_index[a].stop_index + 1);
         i++) {
        result << " ";  // 공란추가
    }
    for (int b = user_seq4.protein[a].size() - 1; b >= 0; b--) {
        result << user_seq4.protein[a][b];  // 단백질 서열출력
    }
    result << endl;
}
```

```
// user_seq5의 결과를 파일에 출력
for (int a = 0; a < user_seq5.protein.size(); a++) {
  for (int i = 0;
       i < out_sequence.length() -
              3 * (user_seq5.complete_index[a].stop_index + 1) - 1;
       i++) {
    result << " ";  // 공란추가
  }
  for (int b = user_seq5.protein[a].size() - 1; b >= 0; b--) {
    result << user_seq5.protein[a][b];  // 단백질 서열출력
  }
  result << endl;
}
// user_seq6의 결과를 파일에 출력
for (int a = 0; a < user_seq6.protein.size(); a++) {
  for (int i = 0;
       i < out_sequence.length() -
              3 * (user_seq6.complete_index[a].stop_index + 1) - 2;
       i++) {
    result << " ";  // 공란추가
  }
  for (int b = user_seq6.protein[a].size() - 1; b >= 0; b--) {
    result << user_seq6.protein[a][b];  // 단백질 서열출력
  }
  result << endl;
}
result.close();
```

3-2. 함수 구현

(1) 사용자가 입력하는 정보를 초기화 : void User::SetName(string name), void User::SetSequence(string sequence)

- 입출력

string user_name : 객체 내에 입력 받은 이름을 저장하는 변수

string user_dna_sequence : 객체 내에 입력 받은 서열을 저장하는 변수

매개변수 :

string name, sequence : 이름과 서열을 입력 받기 위함

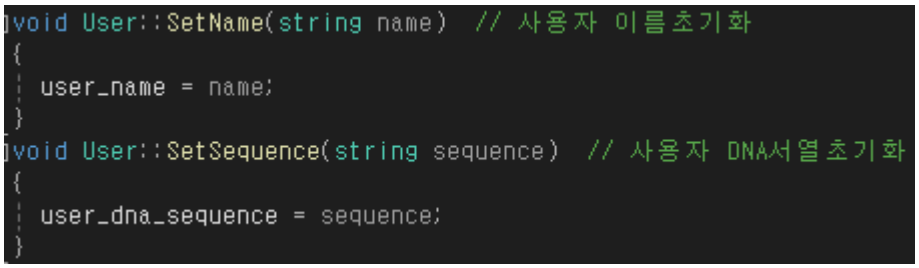
- 설명

이름과 서열을 받아서 객체 내의 변수에 저장하는 함수이다.

- 적용된 배운 내용

get/set 메소드, 클래스, 스트링

- 코드 스크린샷



(2) 사용자의 서열을 frameshift 함 : void User::FrameSetting()

- 입출력

string temp_dna : 임시적으로 서열을 저장하고 이것을 조작하여서 객체 변수에 저장시킴.

string dna1~dna3 : 객체 내에 조작한 서열을 저장해 놓기 위한 변수.

string reverse_dna1~reverse_dna3 : 객체 내에 조작한 서열을 저장해 놓기 위한 변수. 역방향의 frame을 갖는 경우

- 설명

서열의 맨 앞 부분을 0, 1, 2 개씩 지워서 frameshift가 일어난 효과를 준다. 역방향은 우선 기존 서열을 뒤집고 0, 1, 2 개씩 지워서 frameshift가 일어난 효과를 준다.

- 적용된 배운 내용

for문, 증감연산자, 사칙연산자, 비교연산자, 클래스, 스트링, 배열접근

- 코드 스크린샷

```
void User::FrameSetting()  // reading frame을 바꿔서 서열을 저장하는 함수
{
 string temp_dna =
     user_dna_sequence;  // DNA서열의 frame 별 저장을 위한 임시변수;

 dna1 = user_dna_sequence;  // 유저의 정방향, reading frame DNA 저장

 // 맨 앞 하나를 지워서 frameshift
 dna2 = temp_dna.erase(0, 1);  // 유저의 정방향, +1 reding frame DNA 저장

 // 맨 앞 둘을 지워서 frameshift
 dna3 = temp_dna.erase(0, 1);  // 유저의 정방향, +2 reding frame DNA 저장

 for (int i = (int)user_dna_sequence.length() - 1; i >= 0; i--) {
  reverse_dna1 +=
      user_dna_sequence[i];  // 유저의 역방향, reading frame DNA 저장
 }

 temp_dna = reverse_dna1;  // 역방향을 위한 저장

 // 맨 앞 하나를 지워서 frameshift
 reverse_dna2 =
     temp_dna.erase(0, 1);  // 유저의 역방향, +1 reding frame DNA 저장

 // 맨 앞 둘을 지워서 frameshift
 reverse_dna3 =
     temp_dna.erase(0, 1);  // 유저의 역방향, +2 reding frame DNA 저장
}
```

(3) 사용자가 입력한 정보를 출력 : string User::GetSequence(), string User::GetName()

- 입출력

string user_name : 유저의 이름을 저장하고 있는 User클래스 객체 변수

string user_dna_sequence : 유저의 서열을 저장하고 있는 User클래스 객체 변수

- 설명

객체 내에 저장된 사용자의 정보를 반환하는 Get메소드 함수이다. 은닉화를 위함

- 적용된 배운 내용

스트링, get/set 메소드, 클래스

- 코드 스크린샷

```
string User::GetName() {  // 사용자 이름반환
  return user_name;
}
string User::GetSequence() {  // 사용자 DNA서열반환
  return user_dna_sequence;
}
```

(4) frameshift하여 저장한 서열을 출력 : string User::GetDna1()~GetDna3(), GetRDna1()~3()

- 입출력

string dna1~dna3 : 객체 내에 조작한 서열을 저장해 놓기 위한 변수. 이것을 반환

string reverse_dna1~reverse_dna3 : 객체 내에 조작한 서열을 저장해 놓기 위한 변수. 역방향의 frame을 갖는 경우. 이것을 반환

- 설명

객체에 저장된 frame이 바뀐 서열을 반환하는 6개의 함수. get 메소드

- 적용된 배운 내용

get/set 메소드, 스트링, 클래스

- 코드 스크린샷

```
// 저장된 reading frame 변환된 서열을 반환하는 함수들
string User::GetDna1() { return dna1; }
string User::GetDna2() { return dna2; }
string User::GetDna3() { return dna3; }
string User::GetRDna1() { return reverse_dna1; }
string User::GetRDna2() { return reverse_dna2; }
string User::GetRDna3() { return reverse_dna3; }
```

(5) 서열을 분석하기 쉽게 스트링 벡터화 : void Orf::TransferSeq()

- 입출력

string triplet : 임시적으로 3 개의 codon을 담아두기 위해 사용. 이것을 스트링 벡터에 넣는다.

string original_seq : 사용자가 입력한 서열을 frame에 따라 나눈 서열을 하나 저장하는 Orf의 객체 변수

vector<string> orf1 complete_orf : 서열을 스트링 벡터화 하여 3 개씩 서열을 하나의 스트링에 담기 위한 Orf 객체 변수.

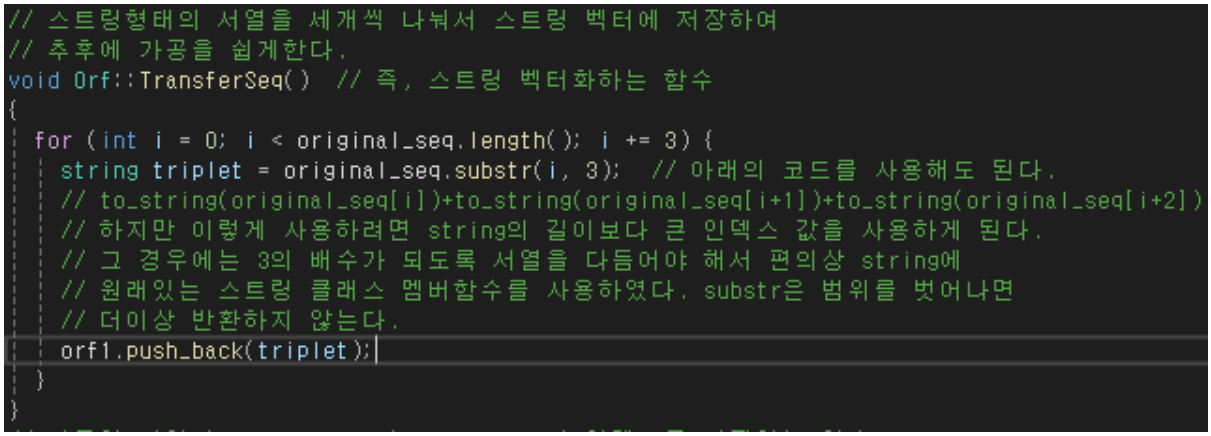
- 설명

서열의 조작을 용이하게 하기 위하여 서열을 스트링 구조에서 3 개씩 나누어 각각 스트링에 담아서 벡터화 한다.

- 적용된 배운 내용

스트링, 벡터, 비교연산자, 복합대입연산자

- 코드 스크린샷



```
// 스트링형태의 서열을 세개씩 나눠서 스트링 벡터에 저장하여
// 추후에 가공을 쉽게한다.
void Orf::TransferSeq()  // 즉, 스트링 벡터화하는 함수
{
  for (int i = 0; i < original_seq.length(); i += 3) {
    string triplet = original_seq.substr(i, 3);  // 아래의 코드를 사용해도 된다.
    // to_string(original_seq[i])+to_string(original_seq[i+1])+to_string(original_seq[i+2])
    // 하지만 이렇게 사용하려면 string의 길이보다 큰 인덱스 값을 사용하게 된다.
    // 그 경우에는 3의 배수가 되도록 서열을 다듬어야 해서 편의상 string에
    // 원래있는 스트링 클래스 멤버함수를 사용하였다. substr은 범위를 벗어나면
    // 더이상 반환하지 않는다.
    orf1.push_back(triplet);
  }
}
```

(6) 스트링 벡터화된 서열에서 ORF의 시작과 끝을 찾음 : void Orf::IndexFinder()

- 입출력

vector<string> orf1 complete_orf : 서열을 스트링 벡터화 하여 3 개씩 서열을 하나의 스트링에 담기 위한 Orf 객체 변수. 이 서열에서의 벡터의 인덱스를 찾는다.
vector<int> atg_index : start codon(ATG) 에 해당하는 string을 찾으면 이 때의 벡터의 인덱스를 저장하기 위해 사용.

vector<int> tga_index : stop codon(TGA) 에 해당하는 string을 찾으면 이 때의 벡터의 인덱스를 저장하기 위해 사용.

vector<int> taa_index : stop codon(TAA) 에 해당하는 string을 찾으면 이 때의 벡터의 인덱스를 저장하기 위해 사용.

vector<int> tag_index : stop codon(TAG) 에 해당하는 string을 찾으면 이 때의 벡터의

인덱스를 저장하기 위해 사용.

- 설명

orf1 에 저장되어 있는 스트링 벡터를 탐색하여 start codon과 stop codon에 해당하는 서열이 나오면 이때의 벡터 인덱스를 저장한다. 추후 이것을 이용하여 ORF를 저장할 것이다.

- 적용된 배운 내용

스트링, 벡터, 배열 접근, 비교연산자, 증감연산자, for 조건문, if 조건문

- 코드 스크린샷

```
// 가공한 서열의 start codon과 stop codon의 인덱스를 저장하는 함수
void Orf::IndexFinder() {  // ATG(start codon)인 스트링을 찾아서 그때의 인덱스
                           // 값을 벡터에 저장
 for (int i = 0; i < orf1.size(); i++) {
  if (orf1[i] == "ATG") atg_index.push_back(i);
 }
 // TGA(stop codon)인 스트링을 찾아서 그때의 인덱스 값을 벡터에 저장
 for (int i = 0; i < orf1.size(); i++) {
  if (orf1[i] == "TGA") tga_index.push_back(i);
 }
 // TAA(stop codon)인 스트링을 찾아서 그때의 인덱스 값을 벡터에 저장
 for (int i = 0; i < orf1.size(); i++) {
  if (orf1[i] == "TAA") taa_index.push_back(i);
 }
 // TAG(stop codon)인 스트링을 찾아서 그때의 인덱스 값을 벡터에 저장
 for (int i = 0; i < orf1.size(); i++) {
  if (orf1[i] == "TAG") tag_index.push_back(i);
 }
}
```

(7) 찾은 ORF의 시작과 끝을 가지고 가능한 모든 ORF를 찾음 : void Orf::OrfFinder(User user)

- 입출력

int range1, range2 : 사용자가 입력한 범위인 User user 객체에 있는 범위를 받아오기 위해 사용

int range1, range2 : 사용자가 입력한 범위를 담고있는 User의 멤버 변수

vector<int> atg_index : start codon(ATG)에 해당하는 string을 갖는 벡터의 인덱스를 사용하기 위함.

vector<int> tga_index : stop codon(TGA)에 해당하는 string을 갖는 벡터의 인덱스를 사용하기 위함.

vector<int> taa_index : stop codon(TAA)에 해당하는 string을 갖는 벡터의 인덱스를 사용하기 위함.

vector<int> tag_index : stop codon(TAG)에 해당하는 string을 갖는 벡터의 인덱스를 사용하기 위함.

vector<string> temp_com_orf : ORF를 찾으면 임시 저장해두기 위한 벡터. 이를 이중 벡터에 저장하여 모두 모아둔다.

vector<vector<string>> complete_orf : 찾은 ORF를 여기에 모두 저장하여 같은 reading frame을 가지는 서열에서 나온 ORF를 모두 저장해둔다.

SavedIndex temp_saved : ORF를 찾았을 때의 start codon과 stop codon의 인덱스를 저장해 두기 위해 사용. 이를 아래의 벡터에 저장.

vector<SavedIndex> complete_index : 찾은 ORf에 대응하는 start codon과 stop codon을 저장해두기 위한 변수. 이는 추후에 파일 출력에 사용한다

매개변수 : User user : 사용자에게서 받은 이름, 서열, 범위 정보가 존재하는 객체

- 설명

start codon과 stop codon이 존재하는 곳의 인덱스를 사용해서 start codon이 stop codon보다 더 앞에 존재할 때, 해당 범위의 string을 스트링 벡터에 모두 저장하고 이것을 또 이중 벡터에 저장한다. 이렇게 하여 하나의 서열에서 찾은 모든 ORF를 하나의 이중 벡터 안에 저장한다.

또한, 이때의 인덱스를 벡터에 저장하여서 각 ORF가 몇 번째 인덱스를 가졌는지를 저장한다. 이는 추후에 파일 출력에서 원래 서열과 비교하기 위해 사용한다.

- 적용된 배운 내용

if 조건문, for 문, 클래스, has-a 관계, 벡터, 스트링, 배열 접근

- 코드 스크린샷

```
// 저장된 인덱스의 값에 따라 시작(ATG)부터 끝(TAA,TAG,TGA)까지가 사용자의 범위에
// 들어가면 그 서열을 스트링 이중벡터에 저장한다.
void Orf::OrfFinder(User user) {
  int range1 = user.range1;  // 사용자가 입력한 범위를 가져온다.
  int range2 = user.range2;
  for (int i = 0; i < atg_index.size();
       i++)  // 저장된 시작코돈이 있는 인덱스를 사용하기위함
  {  // start codon은 동일하기 때문에 세개의 stop codon이 공유. 하위의 stop
    // codon을 세개의 for문으로 탐색
    for (int j = 0; j < tga_index.size();
         j++)  // stop codon이 저장된 인덱스를 사용하기위함
    {          // stop codon이 TGA인 경우
      if (atg_index[i] <
          tga_index[j])  // 시작 코돈이 stop codon보다 앞에 있을 때
      {  // 시작부터 끝까지의 서열의 크기가 사용자가 입력한 범위와 일치하는 경우
        if (tga_index[j] - atg_index[i] + 1 >= range1 &&
            tga_index[j] - atg_index[i] + 1 <= range2
          ) {  // 그 시작부터 끝까지(ORF)를 스트링 이중 벡터에
                          // 저장(push_back())한다.
          vector<string> temp_com_orf(orf1.begin() + atg_index[i],
                                      orf1.begin() + tga_index[j] + 1);
          complete_orf.push_back(temp_com_orf);

          // ORF가 추출 될 때마다의 인덱스 값을 저장
          //저장된 stop codon의 종류에 따라 0, 1, 2를 추가로 저장
          SavedIndex temp_saved(0, atg_index[i], tga_index[j]);
          complete_index.push_back(temp_saved);
        }
      }
    }
```

```
    for (int j = 0; j < taa_index.size();
         j++)  // stop codon의 인덱스를 사용하기 위함
    {  // 위의 for문과 동일한 원리, 하지만 stop codon이 TAA인 경우
      if (atg_index[i] < taa_index[j]) {
        if (taa_index[j] - atg_index[i] + 1 >= range1 &&
            taa_index[j] - atg_index[i] + 1 <= range2) {
          vector<string> temp_com_orf(orf1.begin() + atg_index[i],
                                      orf1.begin() + taa_index[j] + 1);
          complete_orf.push_back(temp_com_orf);

          SavedIndex temp_saved(1, atg_index[i], taa_index[j]);
          complete_index.push_back(temp_saved);
        }
      }
    }
    for (int j = 0; j < tag_index.size();
         j++)  // stop codon의 인덱스를 사용하기 위함
    {  // 위의 for문과 동일한 원리, 하지만 stop codon이 TAG인 경우
      if (atg_index[i] < tag_index[j]) {
        if (tag_index[j] - atg_index[i] + 1 >= range1 &&
            tag_index[j] - atg_index[i] + 1 <= range2) {
          vector<string> temp_com_orf(orf1.begin() + atg_index[i],
                                      orf1.begin() + tag_index[j] + 1);
          complete_orf.push_back(temp_com_orf);

          SavedIndex temp_saved(2, atg_index[i], tag_index[j]);
          complete_index.push_back(temp_saved);
        }
      }
    }
  }
}
```

(8) 찾은 ORF에서 intron을 제거 : void Orf::IntronFinder()

- 입출력

vector<vector<string>> complete_orf : 찾은 ORF가 모두 저장되어 있는 이중 벡터. Orf 클래스의 멤버 변수

vector<vector<string>> intron_removed : ORF의 인트론이 제거된 버전을 저장하기 위한 Orf의 클래스멤버 변수

- 설명

앞에서 찾은 ORF(complete_orf)를 인트론이 제거된 버전을 담을 그릇(intron_removed)에 복사하여 담는다. 담긴 서열에서 intron의 시작과 끝에 해당하는 서열을 만나면 그 사이, 즉, 해당하는 범위를 모두 intron으로 바꾼다. 이때, stop codon이 있는 경우에는 바꾸지 않는다.(분석결과의 분석에 도움이 될 수 있기 때문에) 인트론에 해당하는 부분을

바꾼 후에는 계속해서 진행하여 또 존재할 수 있는 인트론을 표시해 나간다.

- 적용된 배운 내용

논리연산자, 비교연산자, 증감연산자, if문, for 문, 클래스, 스트링, 벡터

- 코드 스크린샷

```
// 인트론을 제거하는 함수
void Orf::IntronFinder() {
  intron_removed =
      complete_orf;  // 찾은 orf를 인트론이 잘린 변수가 담길 곳에 복사

  for (int a = 0; a < intron_removed.size(); a++)  // 찾은 ORF의 수만큼 반복
  {//너무 짧으면 존재하지 않는 인덱스에 접근가능하니, 다음 시행으로 넘어감.
    if (intron_removed[a].size() < 6) continue;
    for (int b = 0; b < intron_removed[a].size() - 5;
         b++)  // 끝까지 못가게 하여 없는 인덱스를 넣지 않도록
    {          // 전체 서열을 탐색
      // 탐색하다가 인트론의 시작 부분을 만나면
      if (intron_removed[a][b] == "GTA" && intron_removed[a][b + 1] == "AGT") {
        for (int c = b + 1; c < intron_removed[a].size();
             c++)  // 그 이후의 서열을 탐색
        {          // 인트론의 끝부분에 해당하는 부분을 찾으면
          if (intron_removed[a][c - 3] == "TTT" &&
              intron_removed[a][c - 2] == "TTT") {
            if (intron_removed[a][c - 1][0] == 'T' &&
                intron_removed[a][c - 1][1] == 'T') {
              if (intron_removed[a][c] == "CAG") {
                for (int i = b; i <= c; i++)  // 찾아낸 인트론의 처음과 끝을,
                {
                  if (intron_removed[a][i] == "TGA" ||
                      intron_removed[a][i] == "TAA" ||
                      intron_removed[a][i] == "TAG")
                    {}//stop codon일 경우 그대로 남겨놓음.
                  else
                    {
                      intron_removed[a][i] =
                    "itr";  // 해당하는 범위를 모두 인트론으로 바꿈
                    }

                }
                break;  // 완료한 후 for문을 탈출 하여 가장 처음으로 만난
                        // 인트론의 끝부분까지만
                // 인트론으로 바꿈.
              }
            }
          }
        }
      }
    }
  }
}
```

(9) intron이 제거된 ORF를 단백질화 : void Orf::CodonDecipher()

- 입출력

string pro_cpy : 단백질 정보를 저장할 벡터(protein)에 저장된 인트론 제거버전의 triplet을 받아서 codon table과 비교하기 위해 값을 저장하는 변수.

vector<vector<string>> intron_removed : ORF의 인트론이 제거된 버전을 저장하기 위한 Orf클래스의 멤버 변수

vector<vector<string>> protein : 인트론이 제거된 버전(intron_removed)을 단백질 정보로 변환하여 저장하기 위한 Orf 클래스의 멤버 변수

- 설명

먼저 intron_removed의 값을 받아서 protein 변수에 넣는다. protein에 저장된 값에 따라 이를 단백질 정보로 바꿔준다.

- 적용된 배운 내용

if 조건문, for 문, 벡터, 스트링, 배열 접근, 비교연산자, 논리연산자, 증감연산자, 클래스

- 코드 스크린샷

```
// 인트론 제거된 ORF를 단백질화 시키는 함수
void Orf::CodonDecipher() {
  protein = intron_removed;
  for (int a = 0; a < protein.size(); a++)  // 찾은 ORF에 접근
  {
    for (int b = 0; b < protein[a].size();
        b++)  // 각각 저장된 orf 안의 string들을 탐색
    {  // 그에 맞는 코돈을 만나면 protein 변수에 단백질로 바꿔서 저장.
      string pro_cpy = protein[a][b];
      if (pro_cpy == "TTT" || pro_cpy == "TTC")
        protein[a][b] = " F ";
      else if (pro_cpy == "TTA" || pro_cpy == "TTG" || pro_cpy == "CTT" ||
               pro_cpy == "CTC" || pro_cpy == "CTA" || pro_cpy == "CTG")
        protein[a][b] = " L ";
      else if (pro_cpy == "ATT" || pro_cpy == "ATC" || pro_cpy == "ATA")
        protein[a][b] = " L ";
      else if (pro_cpy == "ATG")
        protein[a][b] = " M ";
      else if (pro_cpy == "GTT" || pro_cpy == "GTC" || pro_cpy == "GTA" ||
               pro_cpy == "GTG")
        protein[a][b] = " V ";
      else if (pro_cpy == "TCT" || pro_cpy == "TCC" || pro_cpy == "TCA" ||
               pro_cpy == "TCG" || pro_cpy == "AGT" || pro_cpy == "AGC")
        protein[a][b] = " S ";
      else if (pro_cpy == "CCT" || pro_cpy == "CCC" || pro_cpy == "CCA" ||
               pro_cpy == "CCG")
        protein[a][b] = " P ";
      else if (pro_cpy == "ACT" || pro_cpy == "ACC" || pro_cpy == "ACA" ||
               pro_cpy == "ACG")
        protein[a][b] = " T ";
      else if (pro_cpy == "GCT" || pro_cpy == "GCC" || pro_cpy == "GCA" ||
               pro_cpy == "GCG")
        protein[a][b] = " A ";
      else if (pro_cpy == "TAT" || pro_cpy == "TAC")
        protein[a][b] = " Y ";
      else if (pro_cpy == "CAT" || pro_cpy == "CAC")
        protein[a][b] = " H ";
      else if (pro_cpy == "CAA" || pro_cpy == "CAG")
        protein[a][b] = " Q ";
```

```
    else if (pro_cpy == "AAT" || pro_cpy == "AAC")
      protein[a][b] = " N ";
    else if (pro_cpy == "AAA" || pro_cpy == "AAG")
      protein[a][b] = " K ";
    else if (pro_cpy == "GAT" || pro_cpy == "GAC")
      protein[a][b] = " D ";
    else if (pro_cpy == "GAA" || pro_cpy == "GAG")
      protein[a][b] = " E ";
    else if (pro_cpy == "TGT" || pro_cpy == "TGC")
      protein[a][b] = " C ";
    else if (pro_cpy == "TGG")
      protein[a][b] = " W ";
    else if (pro_cpy == "CGT" || pro_cpy == "CGC" || pro_cpy == "CGA" ||
             pro_cpy == "CGG" || pro_cpy == "AGA" || pro_cpy == "AGG")
      protein[a][b] = " R ";
    else if (pro_cpy == "GGT" || pro_cpy == "GGC" || pro_cpy == "GGA" ||
             pro_cpy == "GGG")
      protein[a][b] = " G ";
    else if (pro_cpy == "TAA" || pro_cpy == "TAG" || pro_cpy == "TGA")
      protein[a][b] = "stp";
  }
 }
}
```

(10) 각 ORF의 Kozak score를 계산 및 저장 : void Orf::KozakCalculator()

- 입출력

int num : ORF의 start codon의 인덱스를 저장해 놓기 위한 변수.

int score : 점수를 벡터(complete_score)에 넣기 위해 임시 저장용 변수.

int sequence_num : start codon이 존재하는 벡터의 인덱스를 알아내고 이것을 실제 서열에서 존재하는 위치 번호로 바꿔 저장해 놓기 위한 변수.

vector<SavedIndex> complete_index : start codon과 stop codon, stop codon을 담고 있는 객체를 벡터로 저장하여, 각 ORF마다의 객체 정보를 갖게 하기 위함. 이 변수에 담긴 각 ORF의 start codon의 정보를 가져온다.

vector<int> complete_score : 각 ORF별 Kozak score를 저장해 넣기 위한 벡터.

- 설명

각 ORF의 start codon의 인덱스를 알아내서 실제 서열의 번호로 바꾸고, 그 주변의 서열이 무엇인지 확인한다. 특정 조건이 맞다면 점수를 가산한다. 염기서열의 +4위치에 C

또는 G가 있으면 1점 가산. 염기서열의 -2위치에 A 또는 G가 있으면 1점 가산. 염기서열의 -3위치에 A 또는 G가 있으면 1점 가산.

만약, 그 start codon 주변에 접근이 불가하면 7777을 score에 저장해서 계산이 불가함을 나타낸다.

- 적용된 배운 내용

클래스, 배열 접근, 복합대입연산자, 비교연산자, 논리연산자, 증감연산자, if 조건문, for 문

- 코드 스크린샷

```
void Orf::KozakCalculator() {//ORF 수만큼 반복
  for (int i = 0; i < complete_index.size(); i++) {
    int num = complete_index[i]
                  .start_index;  // start인덱스는 저장된 스트링에서의 번호니까
    int sequence_num = num + 3 ;  // 실제 서열에서의 순번을 저장.
    int score = 0;                    // 점수를 저장하기 위한 변수
    // 만약 시작코돈 주변을 접근하려는데 그곳에 서열이 존재하지 않을 경우.
    // 오류상황.
    if (sequence_num - 3 < 0 || sequence_num + 3 > original_seq.length()) {
      score = 7777;
    }
    // 오류가 아닌경우
    else {
      // 염기서열의 +4위치에 C또는 G가 있으면 1점 가산.
      if (original_seq[sequence_num + 3] == 'C' ||
          original_seq[sequence_num + 3] == 'G')
        score += 1;
      // 염기서열의 -2위치에 A 또는 G가 있으면 1점 가산.
      if (original_seq[sequence_num - 2] == 'A' ||
          original_seq[sequence_num - 2] == 'G')
        score += 1;
      // 염기서열의 -3위치에 A 또는 G가 있으면 1점 가산.
      if (original_seq[sequence_num - 3] == 'A' ||
          original_seq[sequence_num - 3] == 'G')
        score += 3;
    }
    complete_score.push_back(score);  // 합산한 최종 스코어를 저장.
  }
}
```

4. 테스트 결과

4-1. 기능계획 테스트 결과

(1) 사용자의 이름과 염기서열 정보, 찾을 ORF 범위 저장

- 이름을 입력

```
이름:
김철수
```

- DNA 서열파일을 찾지 못한 경우

```
DNA서열:
에러: 파일을 불러오지 못했습니다.
```

- 찾고자 하는 ORF의 크기 범위를 입력

```
찾고자하는 ORF의 크기 :
1 100
```

- 먼저 입력 한 수가 더 큰 경우

```
찾고자하는 ORF의 크기 :
100 1
```

```
범위 : 1,100
```

- 이름, 서열파일, ORF 범위의 크기를 모두 입력한 경우. 입력한 결과 출력.

```
이름 :김철수
서열 : ATGTTTTTCTTATTGCTTCTCCTACTGATTATCATAATGGTTGTCGTAGTGTCTTCCTCATCGAGTAGCCCTCCCC
TCACCAACAGAATAACAAAAAGGATGACGAAGAGTGTTGCTGGCGTCGCCGACGGAGAAGGGGTGGCGGAGGGTAATAGTGA
범위 : 1,100
```

(2) 염기서열에서 ORF를 저장

- 6개의 가능한 reading frame에 따라 가공된 결과를 출력

```
dna1: ATGTTTTTCTTATTGCTTCTCCTACTGATTATCATAATGGTTGTCGTAGTGTCTTCCTCATCGAGTAGCCCTCCCCCACCGACTACCACAACGGCTGCCGCAGCGTATTACCAT
CACCAACAGAATAACAAAAAGGATGACGAAGAGTGTTGCTGGCGTCGCCGACGGAGAAGGGGTGGCGGAGGGTAATAGTGA
dna2: TGTTTTTCTTATTGCTTCTCCTACTGATTATCATAATGGTTGTCGTAGTGTCTTCCTCATCGAGTAGCCCTCCCCCACCGACTACCACAACGGCTGCCGCAGCGTATTACCATC
ACCAACAGAATAACAAAAAGGATGACGAAGAGTGTTGCTGGCGTCGCCGACGGAGAAGGGGTGGCGGAGGGTAATAGTGA
dna3: GTTTTTCTTATTGCTTCTCCTACTGATTATCATAATGGTTGTCGTAGTGTCTTCCTCATCGAGTAGCCCTCCCCCACCGACTACCACAACGGCTGCCGCAGCGTATTACCATCA
CCAACAGAATAACAAAAAGGATGACGAAGAGTGTTGCTGGCGTCGCCGACGGAGAAGGGGTGGCGGAGGGTAATAGTGA
dna4: AGTGATAATGGGAGGCGGTGGGGAAGAGGCAGCCGCTGCGGTCGTTGTGAGAAGCAGTAGGAAAAACAATAAGACAACCACTACCATTATGCGACGCCGTCGGCAACACCATCA
GCCACCCCCTCCCGATGAGCTACTCCTTCTGTGATGCTGTTGGTAATACTATTAGTCATCCTCTTCGTTATTCTTTTTGTA
dna5: GTGATAATGGGAGGCGGTGGGGAAGAGGCAGCCGCTGCGGTCGTTGTGAGAAGCAGTAGGAAAAACAATAAGACAACCACTACCATTATGCGACGCCGTCGGCAACACCATCAG
CCACCCCCTCCCGATGAGCTACTCCTTCTGTGATGCTGTTGGTAATACTATTAGTCATCCTCTTCGTTATTCTTTTTGTA
dna6: TGATAATGGGAGGCGGTGGGGAAGAGGCAGCCGCTGCGGTCGTTGTGAGAAGCAGTAGGAAAAACAATAAGACAACCACTACCATTATGCGACGCCGTCGGCAACACCATCAGC
CACCCCCTCCCGATGAGCTACTCCTTCTGTGATGCTGTTGGTAATACTATTAGTCATCCTCTTCGTTATTCTTTTTGTA
```

- 벡터화한 분석할 DNA를 출력

```
벡터화dna1:
ATG TTT TTC TTA TTG CTT CTC CTA CTG ATT ATC ATA ATG GTT GTC GTA GTG TCT TCC TCA TCG AGT AGC
CCT CCC CCA CCG ACT ACC ACA ACG GCT GCC GCA GCG TAT TAC CAT CAC CAA CAG AAT AAC AAA AAG GAT
GAC GAA GAG TGT TGC TGG CGT CGC CGA CGG AGA AGG GGT GGC GGA GGG TAA TAG TGA
```

```
벡터화dna2:
TGT TTT TCT TAT TGC TTC TCC TAC TGA TTA TCA TAA TGG TTG TCG TAG TGT CTT CCT CAT CGA GTA GCC
CTC CCC CAC CGA CTA CCA CAA CGG CTG CCG CAG CGT ATT ACC ATC ACC AAC AGA ATA ACA AAA AGG ATG
ACG AAG AGT GTT GCT GGC GTC GCC GAC GGA GAA GGG GTG GCG GAG GGT AAT AGT GA
```

```
벡터화dna3:
GTT TTT CTT ATT GCT TCT CCT ACT GAT TAT CAT AAT GGT TGT CGT AGT GTC TTC CTC ATC GAG TAG CCC
TCC CCC ACC GAC TAC CAC AAC GGC TGC CGC AGC GTA TTA CCA TCA CCA ACA GAA TAA CAA AAA GGA TGA
CGA AGA GTG TTG CTG GCG TCG CCG ACG GAG AAG GGG TGG CGG AGG GTA ATA GTG A
```

```
벡터화dna4:
AGT GAT AAT GGG AGG CGG TGG GGA AGA GGC AGC CGC TGC GGT CGT TGT GAG AAG CAG TAG GAA AAA CAA
TAA GAC AAC CAC TAC CAT TAT GCG ACG CCG TCG GCA ACA CCA TCA GCC ACC CCC TCC CGA TGA GCT ACT
CCT TCT GTG ATG CTG TTG GTA ATA CTA TTA GTC ATC CTC TTC GTT ATT CTT TTT GTA
```

```
벡터화dna5:
GTG ATA ATG GGA GGC GGT GGG GAA GAG GCA GCC GCT GCG GTC GTT GTG AGA AGC AGT AGG AAA AAC AAT
AAG ACA ACC ACT ACC ATT ATG CGA CGC CGT CGG CAA CAC CAT CAG CCA CCC CCT CCC GAT GAG CTA CTC
CTT CTG TGA TGC TGT TGG TAA TAC TAT TAG TCA TCC TCT TCG TTA TTC TTT TTG TA
```

```
벡터화dna6:
TGA TAA TGG GAG GCG GTG GGG AAG AGG CAG CCG CTG CGG TCG TTG TGA GAA GCA GTA GGA AAA ACA ATA
AGA CAA CCA CTA CCA TTA TGC GAC GCC GTC GGC AAC ACC ATC AGC CAC CCC CTC CCG ATG AGC TAC TCC
TTC TGT GAT GCT GTT GGT AAT ACT ATT AGT CAT CCT CTT CGT TAT TCT TTT TGT A
```

- ORF가 존재하지 않는 경우

```
찾은 ORF:
ORF가 탐색되지 않았습니다.
```

- ORF를 여러 개 찾은 경우

```
찾은 ORF:
ATG GGA GGC GGT GGG GAA GAG GCA GCC GCT GCG GTC GTT GTG AGA AGC AGT AGG AAA AAC AAT AAG ACA
ACC ACT ACC ATT ATG CGA CGC CGT CGG CAA CAC CAT CAG CCA CCC CCT CCC GAT GAG CTA CTC CTT CTG
TGA
ATG GGA GGC GGT GGG GAA GAG GCA GCC GCT GCG GTC GTT GTG AGA AGC AGT AGG AAA AAC AAT AAG ACA
ACC ACT ACC ATT ATG CGA CGC CGT CGG CAA CAC CAT CAG CCA CCC CCT CCC GAT GAG CTA CTC CTT CTG
TGA TGC TGT TGG TAA
ATG GGA GGC GGT GGG GAA GAG GCA GCC GCT GCG GTC GTT GTG AGA AGC AGT AGG AAA AAC AAT AAG ACA
ACC ACT ACC ATT ATG CGA CGC CGT CGG CAA CAC CAT CAG CCA CCC CCT CCC GAT GAG CTA CTC CTT CTG
TGA TGC TGT TGG TAA TAC TAT TAG
ATG CGA CGC CGT CGG CAA CAC CAT CAG CCA CCC CCT CCC GAT GAG CTA CTC CTT CTG TGA
ATG CGA CGC CGT CGG CAA CAC CAT CAG CCA CCC CCT CCC GAT GAG CTA CTC CTT CTG TGA TGC TGT TGG
TAA
ATG CGA CGC CGT CGG CAA CAC CAT CAG CCA CCC CCT CCC GAT GAG CTA CTC CTT CTG TGA TGC TGT TGG
TAA TAC TAT TAG
```

(3) 얻어낸 ORF에서 intron을 제거

- 인트론이 존재하는 경우

```
인트론 가공 후:
ATG GGG AAA GGG itr itr itr itr itr itr itr itr itr itr itr itr itr AAA CCC AAA CCC AAA GGG TGA
```

- 인트론이 존재하지 않는 경우

```
인트론 가공 후:
ATG GGG AAA GGG AAA CCC AAA GGG TGA
```

- 가공할 ORF가 존재하지 않는 경우

- 인트론이 존재하는 경우

- 인트론이 두 개 이상 존재하는 경우

- stop codon이 intron에 끼어들어간 경우

(4) Intron이 제거된 CDS를 단백질 형태의 정보로 변환

- ORF 자체가 존재하지 않는 경우

-단백질 해독한 결과

- 인트론이 포함된 서열을 해독한 경우

(5) Kozak sequence score를 계산

-계산이 필요치 않은 경우

- start codon 근처에 존재하는 계산에 필요한 인덱스에 접근 불가한 경우

```
Kozak score : 계산불가
```

- 계산할 서열이 존재하는 경우

```
Kozak score : 5 5
```

- 계산할 ORF가 여러 개 존재하는 경우

```
Kozak score : 5 5 5 5 5 5 5 5 5 5 5 5
```

-계산 불가한 ORF가 여러 개인 경우

```
Kozak score : 계산불가 계산불가
```

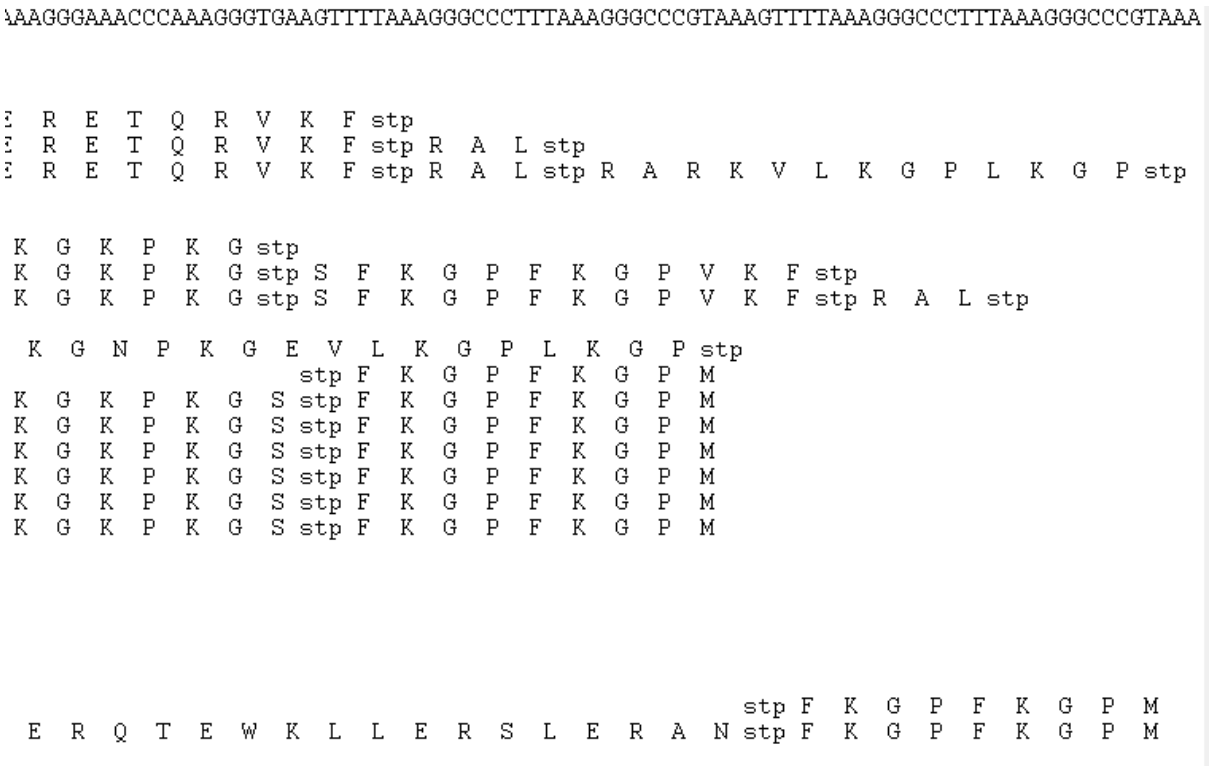
(6) 분석한 결과를 파일로 출력

- 찾은 CDS 크기가 주어진 서열과 딱 맞는 경우

```
ATGGGGAAAGGGGTAAGTAAAAAAAAAAAAAAAAAAAAAAAAAAATAATTTTTTTTTTCAGAAACCCAAACCCAAAGGGTGA
 M   G   K   G itritritritritritritritritritritritrstpitritritritritr K   P   K   P   K   G stp
 M   G   K   G   V   S   K   K   K   K   K   K   K   K stp
```

- 주어진 서열 중간에서 CDS를 찾은 경우

```
```



4-2. 함수계획 테스트 결과

(1) 사용자가 입력하는 정보를 초기화 : void User::SetName(string name), void User::SetSequence(string sequence)

-이름을 입력

```
김철수
```

-서열을 입력

```
AGTTTTAAAGGGCCCTTTAAAGGGCCCGTAAAAAGCCAAAAAATGGGGAAAGGGAAACCCAAAGGGTGATAATAGAAAAGATAATAGTG
```

(2) 사용자의 서열을 frameshift 함 : void User::FrameSetting()

-맨 앞 서열들을 지우거나 남겨서 framshift

```
dna1: ATGGGGAAAGGGAAACCCAAAGGGTGA
dna2: TGGGGAAAGGGAAACCCAAAGGGTGA
dna3: GGGGAAAGGGAAACCCAAAGGGTGA
```

-맨 앞 서열들을 지우거나 남겨서 framshift(역방향)

```
dna4: AGTGGGAAACCCAAAGGGAAAGGGGTA
dna5: GTGGGAAACCCAAAGGGAAAGGGGTA
dna6: TGGGAAACCCAAAGGGAAAGGGGTA
```

(3) 사용자가 입력한 정보를 출력 : string User::GetSequence(), string User::GetName()

-반환된 이름을 출력한 결과

```
김철수
```

-반환된 서열을 출력한 결과

```
ATGGGGAAAGGGAAACCCAAAGGGTGA
```

(4) frameshift하여 저장한 서열을 출력 : string User::GetDna1()~GetDna3(), GetRDna1()~3()

-반환된 DNA의 정방향 frame 서열을 출력한 경우

```
ATGGGGAAAGGGAAACCCAAAGGGTGA
TGGGGAAAGGGAAACCCAAAGGGTGA
GGGGAAAGGGAAACCCAAAGGGTGA
```

-반환된 DNA의 역방향 frame 서열을 출력한 경우

```
AGTGGGAAACCCAAAGGGAAAGGGGTA
GTGGGAAACCCAAAGGGAAAGGGGTA
TGGGAAACCCAAAGGGAAAGGGGTA
```

(5) 서열을 분석하기 쉽게 스트링 벡터화 : void Orf::TransferSeq()

- `ATGGGGAAAGGGAAACCCAAAGGGTGA` 다음의 서열을 벡터화한 경우

```
ATG GGG AAA GGG AAA CCC AAA GGG TGA
```

- 서열이 3의 배수가 아닌 경우

```
TGG GGA AAG GGA AAC CCA AAG GGT GA
```

(6) 스트링 벡터화된 서열에서 ORF의 시작과 끝을 찾음 : void Orf::IndexFinder()

-ATG ATG ATG ATG TAA TAA TAA TAA TGA TGA TGA TGA TAG TAG TAG TAG 다음의 서열을 출력한 결과

ATG인덱스0 1 2 3

TAA인덱스4 5 6 7

TGA인덱스8 9 10 11

TAG인덱스12 13 14 15

(7) 찾은 ORF의 시작과 끝을 가지고 가능한 모든 ORF를 찾음 : void Orf::OrfFinder(User user)

-다음의 서열에서 ORF를 찾음

```
AGC CAA AAA ATG GGG AAA GGG AAA CCC AAA GGG TGA TAA TAG AAA AGA TAA TAG TGG GTT TCC CGG GGG GAA AGG GGT A
```

-저장된 ORF를 출력

```
ATG GGG AAA GGG AAA CCC AAA GGG TGA
ATG GGG AAA GGG AAA CCC AAA GGG TGA TAA
ATG GGG AAA GGG AAA CCC AAA GGG TGA TAA TAG AAA AGA TAA
ATG GGG AAA GGG AAA CCC AAA GGG TGA TAA TAG
ATG GGG AAA GGG AAA CCC AAA GGG TGA TAA TAG AAA AGA TAA TAG
```

-이때 저장된 stopcodon의 종류(0, 1, 2), start codon의 인덱스, stop codon의 인덱스

0 3 11 1 3 12 1 3 16 2 3 13 2 3 17

(8) 찾은 ORF에서 intron을 제거 : void Orf::IntronFinder()

- 인트론이 가공된 모습

```
ATG GGG AAA GGG itr itr itr itr itr itr itr itr itr itr itr itr itr AAA CCC AAA CCC AAA GGG TGA
```

- 인트론이 두 개 이상 존재하는 경우

```
인트론 가공 후:
ATG GGG AAA GGG itr itr itr itr itr itr itr itr itr itr itr itr itr AAA CCC for the tes tar itr itr itr itr itr itr
itr itr itr itr itr itr itr itr AAA CCC AAA GGG TGA
```

- stop codon이 intron에 끼어들어간 경우

```
인트론 가공 후:
ATG GGG AAA GGG itr itr itr itr itr itr itr itr itr itr itr itr itr itr AAA CCC for the tes tar itr itr itr itr itr itr
itr itr itr itr itr itr itr itr AAA CCC AAA GGG TGA
```

-너무 짧은 ORF(존재하지 않는 배열에 접근가능성이 있는 경우)

```
ATG ATG TAA
```

(9) intron이 제거된 ORF를 단백질화 : void Orf::CodonDecipher()

-codon table의 모든 codon을 순차적으로 입력한 결과

```
M  A  A  A  A  R  R  R  R  R  R  D  D  C  C  Q  Q  G  G  G  G  H  H  I  I  I  L  L  L  L
L  L  K  K  M  N  N  P  P  P  P  T  T  T  T  W  Y  Y  V  V  V  V  N  Q  Q  N  E  E  stp stp
stp
```

(10) 각 ORF의 Kozak score를 계산 및 저장 : void Orf::KozakCalculator()

- +4 위치에 G나 C가 존재하는 경우

```
BBBBBBBBBATGCBBBBBBBBBTAGBBBBBBBBBATGGBBBBBBBBTAG

ATG CBB BBB BBB TAG
ATG CBB BBB BBB TAG BBB BBB BBB ATG GBB BBB BBB TAG
ATG GBB BBB BBB TAG

Kozak score : 1 1 1
```

- -2 위치에 G나 A가 존재하는 경우

```
BBBBBBABATGBBBBBBBBBTAGBBBBBBGBATGBBBBBBBBBTAG

ATG BBB BBB BBB TAG
ATG BBB BBB BBB TAG BBB BBB BGB ATG BBB BBB BBB TAG
ATG BBB BBB BBB TAG

Kozak score : 1 1 1
```

-+4 위치에 G나 C가 존재하는 경우

```
BBBBBBABBATGBBBBBBBBBTAGBBBBBBGBBATGBBBBBBBBBTAG
```

```
ATG BBB BBB BBB TAG
ATG BBB BBB BBB TAG BBB BBB GBB ATG BBB BBB BBB TAG
ATG BBB BBB BBB TAG
```

```
Kozak score : 3 3 3
```

-계산에 필요한 곳의 서열이 존재하지 않는 경우. 저장되는 값.

```
7777
```

5. 계획 대비 변경 사항

1) 변경 내역 제목

- 이전

새롭게 추가

- 이후

2) 염기서열의 정보를 분석

(5) 분석한 결과를 파일로 출력

- 얻어낸 단백질 정보로 바뀐 CDS를 원래의 서열과 비교할 수 있도록 파일로 출력한다.

- 사유

최종 결과를 파일로 한눈에 보기 쉽도록 원래 DNA서열과 비교하기 위해서 추가하였다.

6. 느낀점

이번에 졸업하기 전에 진로를 확실히 정하지 못하여 많은 경험을 해보고자 평소에 하고 싶었던 게임 관련 업종에 종사하면 어떨까 하여 c++ 수업을 수강하게 되었습니다. 프로그래밍이라는 것을 처음 해봤기 때문에 모든 단어들이 생소하였습니다. 수업 첫 날 수업을 듣고 모르는 단어가 너무 많아서 혼자서 공부해야겠다는 생각에 책을 사서 부랴부랴 혼자서 독학하기 시작했습니다. 매번 새 챕터에 진입할 때마다 모르는 개념들이 많아서 그 책을 완독하지는 못했지만 기본적인 개념을 이해하고 혼자 독학할 때와 달리 수업을 들으니 훨씬 수월하였습니다. 지금 생각하면 정말 웃긴데 저는 코딩할 때 빨간 줄이 의미하는 것을 중간고사가 끝나고 알게 되었습니다.

코딩을 처음 해보면서 느낀 점은 생각대로 되지 않으면 절망스럽고 짜증이 나다가도 문제가 풀리거나 코드가 내가 생각한대로 실행되었을 때는 정말 재미있었습니다. 코딩이라는 것이 피드백이 즉각적이기 때문에 공부한 만큼 바로 피드백이 와서 게임 같은 면이 있어서 더 재미있었던 것 같습니다. 제가 시간적 여유가 있었더라면 지금과는 다른 학과를 택했을 것 같습니다. 하지만, 게임 업계나 AI 업계나 제가 느낀 바로는 chatgpt같은 AI들이 등장하면서 적당히 잘해서는 살아남을 수 없지 않을까 하는 생각이 들었습니다. 어느 업계나 마찬가지인데 제가 도망가는 걸까요? 아무튼 다른 직업을 가지고 취미로 계속하면 좋을 것 같다는 결론에 이르렀습니다.

프로젝트를 진행하면서 느낀 바는 다음과 같습니다. 항상 완벽하게 코드를 짰다고 생각하고 실행시키면 저도 예상치 못한 부분에서 문제가 발생하였습니다. 그때마다 짜증도 나고 수업시간에 빠르게 문제를 해결하는 다른 사람들이 부럽기도 하고 신기하였습니다. 하지만 포기하지 않고 계속 고민하다 보면 문제는 어찌저찌 해결되었고 해결되었을 때의 짜릿함은 정말 좋았습니다. 지나고 보면 너무 쉽고 자명한데 당시에는 그 생각을 왜 못했을까 생각되기도 합니다. 코딩뿐만 아니라 인생의 모든 일이 그러한 것 같습니다. 해결되는 과정을 보면 항상 근본을 생각하면 쉽게 풀렸습니다. 개념과 경험의 부족으로 인한 기본 전제 자체가 잘못된 경우와 머릿속으로 생각한 것과 다르게 오타나 이해 부족으로 잘못 입력한 경우가 대부분 이었습니다.

벌써 한 학기가 끝나고 졸업하게 되지만 아직도 확실히 진로를 정하지 못한 것 같아 걱정이 되지만 수업을 들으면서 느꼈던 것과 마찬가지로 포기하지 않고 계속 해 나아가다 보면 너무 쉽고 자명한 해결책이 나타날 것 입니다. 한 학기 동안 열정적이고 친절하게

수업해 주셔서 정말 잘 따라갈 수 있었습니다. 다른 사람에게는 시작이지만 저에게는 마지막이 된 수업이어서 아쉽고 알싸한 기분이 듭니다. 뭔가를 창조한다는 것은 항상 설레고 멋진 일인 것 같습니다. 잠시 예술가의 기분을 느끼게 해주셔서 감사합니다.