설계 보고서

(알고리즘 프로젝트 최종 보고서)

과 목 명	알고리즘과 실습
팀 명	BINARY
학 과	컴퓨터공학과
팀 장	김규리 2018112088
팀 원	송승민 2018112042
	이서연 2018112013
	최수정 2018112011
제 출 일	2020년 06월 15일

목차

1. 문제 정의

1) 패턴 생성 - 중복 순열 생성

```
void MakePattern() {
    char alphabet[4] = { 'A', 'C', 'G', 'T' };
    char arr[PL];

    PermuWithRepet(alphabet, arr, 4, PL, PL, 4);
}
```

위 코드는 A, C, G, T 4개의 문자만을 사용해 만들 수 있는 모든 문자열(길이 11) 생성하는 함수를 호출한다.

```
void PrintPermuWithRepet(char*arr, int C) {//생성된 배열 arr_s에 넣어주기
    int i;

    for (i = 0; i < C; i++) {
        pattern[pcnt][i] = arr[i];
    }
    pcnt++;
    return;
}

void swap(char*num1, char*num2) {//swap 함수
    char temp = *num1;
    *num1 = *num2;
    *num2 = temp;
    return;
}

void PermuWithRepet(char*set, char*arr, int setsize, int arrsize, int C, int i) {

    if (arrsize == 0) {//마지막 중복 순열 생성완료
        PrintPermuWithRepet(arr, C);//생성된 배열 arr_S에 넣어주기
        return;
    }
    else {//중복 순열 생성
        for (i = setsize - 1; i >= 0; i--) {
            swap(&set[i], &set[setsize - 1]);
            arr[arrsize - 1] = set[setsize - 1];
            PermuWithRepet(set, arr, setsize, arrsize - 1, C, i);
            swap(&set[i], &set[setsize - 1]);
        }
    }
}
```

중복 순열을 생성하는 함수로 재귀적으로 호출해 생성한다. set은 사용되는 문자의 배열이고 arr은 생성한 패턴을 임시 저장하는 배열이다. setsize는 set의 크기, arrsize는 생성하려는 패턴의 길이 (11) 이다. C와 i는 처음 함수를 호출할 때 사용한 setsize와 arrsize를 저장하기 위해서 사용된다. 이는 함수를 재귀적으로 호출해 해당 값이 없어지지 않고 유지하기 위해서 사용된다.
결과적으로 A, C, G, T 4개의 문자만을 사용해 길이가 11인 4^11개의 패턴을 생성한다.

2) 랜덤 생성

```
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<int> dis(0, 3);
```

[그림 1]

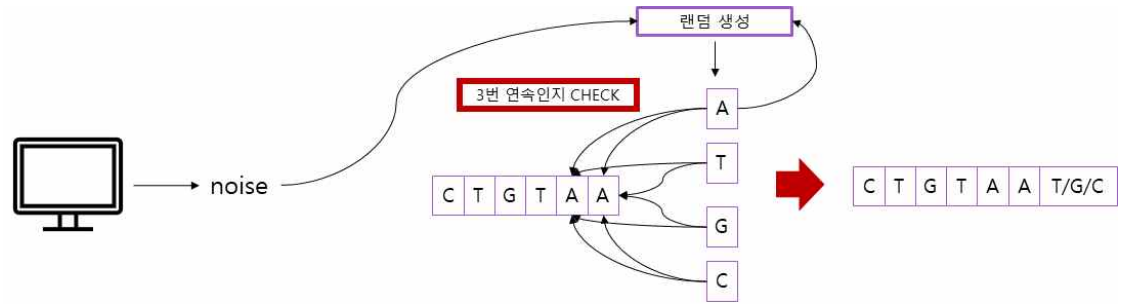
[그림 1]은 난수 생성을 위한 설정 부분이다. 이전 stdlib.h에서 제공하는 랜덤 함수는 여러 가지 수학기법을 이용해 난수처럼 보이는 의사 난수(pseudo random number)을 생성한다. 그렇기 때문에 랜덤 생성하는 수의 양이 많아지면 규칙성이 생긴다는 문제가 발생한다. 또 srand를 이용해 시드값을 얻는 경우 시간을 기준으로 하기 때문에 같은 시간에 프로그램을 작동시키면 같은 난수가 발생한다는 문제가 발생한다.

이런 문제를 해결하기 위해 c++ <random> 라이브러리에서 제공하는 난수 생성기를 사용하였다.

1. random_device rd; 부분은 시드값을 얻는 부분이다. random_device를 사용하면 운영체제 단계에서 제공하는 진짜 난수를 사용할 수 있다. 컴퓨터가 주변의 환경과 무작위적으로 상호작용하면서 만들어지는 것이기 때문에 의사난수보다 난수 생성 속도가 느려 초기화 과정에서 한번 설정해준다.
2. mt19937 gen(rd()); 부분은 random_device 객체인 rd를 이용해 난수 생성 엔진 객체를 정의한다. mt19937은 c++ <random> 라이브러리에서 제공하는 난수 생성 엔진 중 하나이다. '메르센 트위스터' 라는 알고리즘을 사용하며 생성되는 난수들 간의 상관관계가 매우 작다는 장점이 있다.
3. uniform_int_distribution<int> dis(0,3); 부분은 uniform_int_distribution<int> 생성자에 0~3범위를 설정하는 부분이다. 이제 랜덤한 수를 생성하면 0~3사이의 수가 마구잡이로 생성된다.

```
// 패턴 생성
MakePattern();

cout << "pattern 생성 완료 " << endl;

while (i < DL) {
    if (i % 10000 == 0)
        cout << i << endl;
    // 패턴 입력
    for (int t = 0; t < PL; t++)
        out << pattern[j][t];
    j++;

    // 랜덤 부분 생성
    for (int k = 0; k < RL; k++) {
        // random 부분 길이만큼 string 생성 후 삽입
        //tmp = rand() % 4;
        tmp = dis(gen);
        // a, c, g,t로 변환
        switch (tmp) {
        case 0: c = 'A'; break;
        case 1: c = 'C'; break;
        case 2: c = 'G'; break;
        case 3: c = 'T'; break;
        }
        if (k < 2) {
            stmp[k] = c;
        }
        // 연속해서 나오지 못하도록 체크
        else if (stmp[k - 1] != c || stmp[k - 2] != c) {
            stmp[k] = c;
        }
        // 연속해서 나오는 경우 다시 생성
        else {
            k--;
        }
    }

    // 랜덤 부분 입력
    for (int t = 0; t < RL; t++)
        out << stmp[t];

    //i=i+ 패턴 길이 + 랜덤 길이
    i = i + PL + RL;
}
cout << "파일 입력 완료" << endl;
```

[그림 2]



[그림 3]

[그림 2]는 reference DNA 생성 부분에서 [랜덤 부분]을 생성하고 [패턴+랜덤]을 reference DNA파일에 입력해 reference DNA를 생성하는 코드 부분이다.

1. 패턴 생성 함수를 호출해 패턴을 모두 생성한다.
2. while은 설정한 DNA 길이만큼 생성되기까지 반복된다.
3. 패턴을 먼저 파일에 입력한다.
4. dis(gen)으로 랜덤 한 부분을 생성하는 하고 0:A/1:C/2:G/3:T 규칙에 맞게 int형에서 char형으로 변환한다.
5. 생성된 DNA길이가 2보다 작은 경우 임시 저장 배열인 stmp에 바로 저장하고 2보다 큰 경우 앞에 2개와 비교해 연속해서 2개가 나오는지 확인한다. 연속해서 나오는 경우 다시 랜덤 값을 생성하도록 k-를 통해 증가를 상쇄시킨다. ([그림 3] 참고)
6. 이렇게 랜덤 부분의 생성이 끝나면 stmp배열 값을 파일에 입력하고 이전 dna생성 길이에 [패턴 길이+랜덤 길이]를 더해준다.
7. DNA 생성 길이가 설정 DNA길이가 될 때까지 3~6과정을 반복한다.

3) myDNA 생성

```
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<int> dis(0, 99);
FILE* fp = fopen("reference.txt", "r"); //500000개의 랜덤 reference DNA txt파일
if (fp == NULL)
{
    cout << "파일 오픈 불가능" << endl;
    fclose(fp); return;
}
else
{
    fseek(fp, 0, SEEK_END); //파일 포인터를 끝으로 이동시킴
    buf_size = ftell(fp); //파일포인터의 현재위치를 얻음
    buffer = (char*)malloc(buf_size + 1); //파일 크기만큼 동적메모리 할당
    memset(buffer, 0, buf_size + 1); //버퍼 메모리를 0으로 초기화
    fseek(fp, 0, SEEK_SET); //파일 포인터를 처음으로 이동시킴
    fread(buffer, buf_size, 1, fp); //버퍼사이즈만큼 buffer에 읽어옴(파일끝까지 읽어옴)
    refer = buffer;
    free(buffer);
    fclose(fp);
}
```

[그림 1]

[그림 1]은 reference dna가 저장된 파일을 열어 전체 파일 내용을 refer배열에 저장하는 부분이다.
fseek()함수를 이용하여 파일 포인터를 파일의 끝으로 이동시킨 후, ftell()함수를 이용하여 파일 포인터의 현재 위치를 얻어 그 크기만큼 buffer배열에 동적메모리를 할당한다. 그 후 fread를 이용하여 그 파일 크기만큼 파일을 읽어와서 buffer배열에 저장 후, refer배열로 복사한다. 이를 통해 refer배열에 reference.txt파일의 전체 내용이 저장된다.

```
while (i < DL)
{
    //앞에 패턴길이(PL-11)만큼은 그대로 입력
    for (int j = 0; j < PL; j++)
    {
        if (i == DL) break;
        fprintf(file2, "%c", refer[i]);
        //  cout << refer[i];
        i++;
    }
    if (i % 10000 == 0)
        cout << i << '#t';
    //랜덤길이(RL-61)일때
    while (k < RL)
    {
        if (i == DL) break;
        //지금까지 나온 스닙 수가 mismatch개이하 일때만 랜덤확률로 바뀜
        if ((num < mismatch) && (dis(gen) < 20)) // (SNP) 20%의 확률로 바뀜
        {
            num++;
            switch (refer[i])
            {
            case 'A':
                if (rand() % 10 < 3) fprintf(file2, 'C'); //30%의 확률로 C로 바뀜
                else if (rand() % 10 < 3) fprintf(file2, 'G'); //30%의 확률로 G로 바뀜
                else fprintf(file2, "T"); //나머지 확률로 T로 바뀜
                break;
            case 'C':
                if (rand() % 10 < 3) fprintf(file2, 'A');
                else if (rand() % 10 < 3) fprintf(file2, 'G');
                else fprintf(file2, "T");
                break;
            case 'G':
                if (rand() % 10 < 3) fprintf(file2, 'A');
                else if (rand() % 10 < 3) fprintf(file2, 'C');
                else fprintf(file2, "T");
                break;
            case 'T':
                if (rand() % 10 < 3) fprintf(file2, 'A');
                else if (rand() % 10 < 3) fprintf(file2, 'C');
                else fprintf(file2, "G");
                break;
            }
            //cout << "바뀜";
        }
        else //그대로 입력
        {
            fprintf(file2, "%c", refer[i]);
            //cout << refer[i];
        }
        i++;
        k++;
    }
    num = 0;
    k = 0;
}
```

[그림 2]

[그림 2]는 SNP가 포함된 myDNA파일을 생성하는 부분이다.
먼저 fopen()을 통해 myDNA파일을 쓰기모드로 열고 그 안에 쓰는 과정이다.
DL은 DNA길이로, while문을 DNA길이동안 반복해서 수행한다. 먼저 앞부분인 패턴부분에는 SNP가 발생되지 않아야 하므로 패턴길이인 PL(=11)만큼은 refer배열에 저장된 염기를 그대로 입력하며, 그 후인 랜덤부분에서는 SNP가 발생되어야 하므로 RL(=61)만큼 다음 과정을 반복한다.
num은 지금까지 나온 SNP의 개수로, 이 num이 missmatch허용개수인 mismatch보다 미만일 때, 20%의 확률로 SNP이 발생하며 이 확률은 임의로 정한 것이다. SNP의 종류는 A C G T 중 동일한 확률로 선택된다. 만약 SNP가 발생되지 않았거나 지금까지 나온 SNP의 개수(num)가 mismatch를 넘어서면, refer배열에 저장된 염기를 그대로 입력한다.

4) short read 생성

```
int readCount; // shortRead가 N개 생성되었는지 확인

random_device rd;
mt19937 gen(rd());
uniform_int_distribution<int> dis(ReadMin, ReadMax); // 범위 확인하기
uniform_int_distribution<int> overLap(ReadMin / 2, ReadMax / 2); // read 겹치게 하는 범위
```

[그림 1]

[그림 1]은 난수 생성을 위한 설정 부분이다. 1-2) 랜덤 생성에서 사용한 것과 같은 난수 생성기를 사용한다.
1. random_device rd; 부분은 시드값을 얻는 부분이다. random_device를 사용하면 운영체제 단계에서 제공하는 진짜 난수를 사용할 수 있다. 컴퓨터가 주변의 환경과 무작위적으로 상호작용하면서 만들어지는 것이기 때문에 의사난수보다 난수 생성 속도가 느려 초기화 과정에서 한번 설정해준다.
2. mt19937 gen(rd()); 부분은 random_device 객체인 rd를 이용해 난수 생성 엔진 객체를 정의한다. mt19937은 c++ <random> 라이브러리에서 제공하는 난수 생성 엔진 중 하나이다. '메르센 트위스터' 라는 알고리즘을 사용하며 생성되는 난수들 간의 상관관계가 매우 작다는 장점이 있다.
3. uniform_int_distribution<int> dis(ReadMin, ReadMax); 부분은 uniform_int_distribution<int> 생성자에 ReadMin~ReadMax 범위를 설정하는 부분이다. 이제 랜덤한 수를 생성하면 ReadMin~ReadMax 사이의 수가 마구잡이로 생성된다. 이 함수는 랜덤 길이의 read를 설정하는 역할을 담당한다.
4. uniform_int_distribution<int> dis(ReadMin/2, ReadMax/2); 부분은 uniform_int_distribution<int> 생성자에 ReadMin/2 ~ ReadMax/2 범위를 설정하는 부분이다. 이제 랜덤한 수를 생성하면 ReadMin/2 ~ ReadMax/2 사이의 수가 마구잡이로 생성된다. 이 함수는 read를 겹쳐지게 자르기 위해, 앞으로 가거나 뒤로 이동할 때 사용된다.

```
srand((unsigned int)time(0));

ifstream in(filename);
// MY DNA파일로 부터 dna정보를 받아오는 for문
for (i = 0; in.get(c) && i < DL; i++) {
    my[i] = c;
}
in.get(c);
```

[그림 2]

[그림 2]는 myDNA.txt 파일에서 myDNA를 받아서 char 배열에 저장하는 부분이다. 읽어야 할 filename을 make_SR함수의 매개변수로 받아서 읽는다.

```
i = 0; readCount = 0;

k = dis(gen);
string fileN = "SR.txt";
```

[그림 3]

[그림 3]은 SR.txt 파일에 read를 쓸 때 사용되는 변수 초기화 부분이다.
I는 my배열에서 DNA의 인덱스이다. readCount는 read 개수를 세는 변수로, read개수를 초과하거나 ReadN보다 적게 read가 생성되지 않도록 확인해준다. k는 read길이를 저장하는 변수이다.
k=dis(gen); 코드를 통해, 랜덤하게 read길이를 설정한다.
read를 저장할 파일을 SR.txt로 지정한다.

```
if (readCount == ReadN)
    cout << " 모든 정보가 포함된 short read생성!!! 완료!!!!!!!!!!1" << endl;

in.close();
outstm.close();
```

[그림 4]

[그림 4]는 read를 생성하는 부분이다. 앞에서부터 순차적으로 read를 자르면, 복원 난이도도 낮아지고, 성능이 read 순서 때문에 좋아졌다고 생각할 수 있으므로, read 생성 순서를 임의로 설정한다. 매번 read를 1개 자르고 나서, k=dis(gen);을 통해 read길이를 랜덤하게 바꿔준다.
1) DL/2~DL 사이에서 my배열의 뒤에서부터 k 길이의 read를 자른다.
- 겹치게 자르기 위해, overLap(gen)만큼 뒤로 인덱스 이동
2) 0~ DL/2 사이에서 my배열의 앞에서부터 k 길이의 read를 자른다.
- 겹치게 자르기 위해, overLap(gen)만큼 앞으로 인덱스 이동
3) DL/4부터 my배열의 앞에서부터 k 길이의 read를 자른다.
- 설정한 read개수를 맞추기 위해, overLap(gen)만큼 뒤로 인덱스를 이동

```
ofstream outstm(fileN);
if (outstm.is_open()) {

    for (i = DL; i > DL / 2; i -= k) { // 뒤에서부터 k만큼 앞으로 이동.
        // i-k부터 i-1번째까지 k개의 문자열.
        for (j = (i - k); j < i; j++)
            outstm << my[j]; // 파일에 바로 shortread 문자 1개씩 쓰기
        outstm << '₩n';

        i += overLap(gen);

        k = dis(gen);
        readCount++;
    }
    cout << readCount << endl;

    for (i = 0; i <= DL / 2; i += k) { // 앞에서부터 k만큼 뒤로 이동.
        // i부터 i+k-1번째까지 k개의 문자열.
        for (j = i; j < i + k; j++)
            outstm << my[j]; // 파일에 바로 shortread 문자 1개씩 쓰기
        outstm << '₩n';

        i -= overLap(gen);

        k = dis(gen);
        readCount++;
    }
    cout << readCount << endl;
    i = DL / 4;
    while (readCount < ReadN) { // n개가 되도록 또다른 랜덤규칙으로 read마저 자르기
        // i부터 i+k-1번째까지 k개의 문자열.
        for (j = i; j < i + k; j++)
            outstm << my[j]; // 파일에 바로 shortread 문자 1개씩 쓰기
        outstm << '₩n';

        i += overLap(gen);
        k = dis(gen);
        readCount++;
    }
}
```

[그림 5]

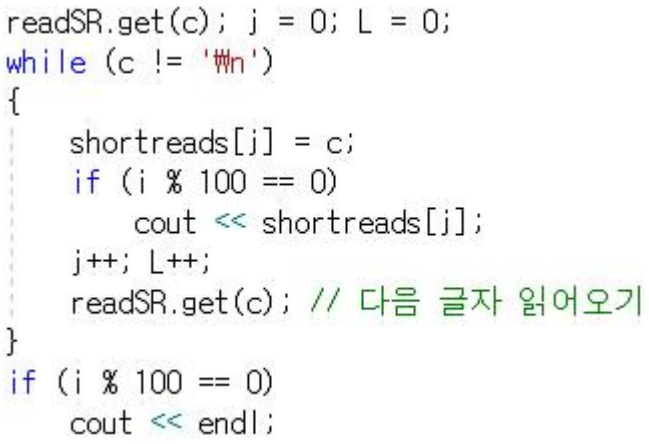
[그림 5]에서는 read가 정확히 ReadN개 생성되었는지 확인한다.
읽고 쓰기 위해 열었던 ifstream과 ofstream을 닫는다.

2. 벤치마킹 알고리즘

```
// ref 읽어오기
cout << "getRF함수에서 ref 배열에 refDNA 저장" << endl;
getRF(ref);
```

[그림 1]

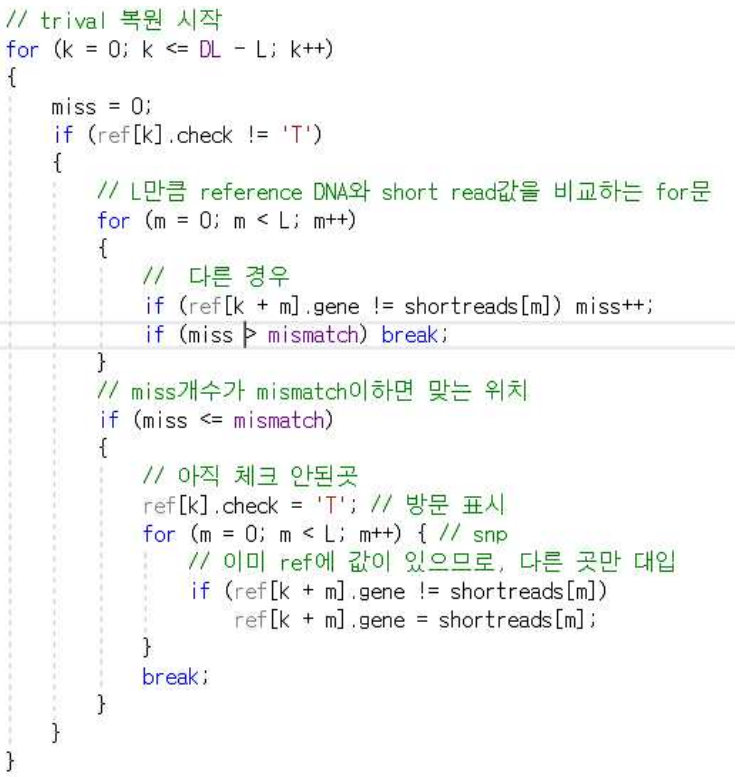
[그림 1]에서는 reference.txt에서 reference DNA를 읽어서 ref에 저장한다.
모든 알고리즘들은 복원된 myDNA를 ref배열에 복원한 다음 파일에 저장한다. 그 다음에,
main.cpp에서 similarity 함수를 이용해서 DNA가 저장된 파일을 불러와 비교하는 방식이다.



```
readSR.get(c); j = 0; L = 0;
while (c != '₩n')
{
    shortreads[j] = c;
    if (i % 100 == 0)
        cout << shortreads[j];
    j++; L++;
    readSR.get(c); // 다음 글자 읽어오기
}
if (i % 100 == 0)
    cout << endl;
```

[그림 2]

[그림 2]에서는 SR.txt 파일에서 \n이 나오기 전까지 문자 1개씩 읽어서 read 1개를 shortreads 배열에 저장하는 부분이다.



```
// trival 복원 시작
for (k = 0; k <= DL - L; k++)
{
    miss = 0;
    if (ref[k].check != 'T')
    {
        // L만큼 reference DNA와 short read값을 비교하는 for문
        for (m = 0; m < L; m++)
        {
            //  다른 경우
            if (ref[k + m].gene != shortreads[m]) miss++;
            if (miss > mismatch) break;
        }
        // miss개수가 mismatch이하면 맞는 위치
        if (miss <= mismatch)
        {
            // 아직 체크 안된곳
            ref[k].check = 'T'; // 방문 표시
            for (m = 0; m < L; m++) { // snp
                // 이미 ref에 값이 있으므로, 다른 곳만 대입
                if (ref[k + m].gene != shortreads[m])
                    ref[k + m].gene = shortreads[m];
            }
            break;
        }
    }
}
```

[그림 3]

[그림 3]에서는 ref 배열에 trivial방식으로 DNA를 복구한다.
0부터 (DL-L)까지 sliding하면서 read와 ref를 비교한다.
1) check 표시가 되어있지 않으면, ref에서 L개 문자와 shortreads의 L개 문자를 비교한다.
2) 하나씩 일대일 비교를 하면서, 불일치하는 경우, miss++;을 한다. 만약에, mismatch 허용개수를 초과하면, 더 매칭되는 문자열이 아니므로, break하고, t+1 인덱스의 ref와 비교를 한다.
3) mismatch 허용 개수 이하이면, 방문했다는 check 표시를 하고, ref에 복원한다.
check를 써서, 모든 read가 DL-L만큼 이동하면서 비교하지 않아도 된다. 복원시간이 줄어든다는 장점이 있다.
이와 같이 trivial 복원을 마친 후, 다시 SR.txt 파일에서 read를 읽어와서 복원을 수행한다.

```
readSR.close();

// 복원된 ref를 파일에 저장
ofstream writeTriv("trivialMyDNA.txt");
if (writeTriv.is_open())
{
    for (i = 0; i < DL; i++)
        writeTriv << ref[i].gene;
}
writeTriv.close();
```

[그림 4]

[그림 4]는 SR.txt에서 read를 모두 읽어서 trivial 복원 후, ifstream을 닫아주는 모습이다. 그리고 복원된 ref 배열을 trivialMyDNA.txt에 쓰는 부분이다. ofstream을 열어서 파일을 쓰고, close한다.

3. REFERENCE DNA 사용 안하는 알고리즘

- DENOVO 알고리즘

3-1. 알고리즘 설명

```
void denovo() {
    struct SHORT *First;

    // 1. 모든 short read 정보 load
    cout << "1. 모든 short read 정보 load" << endl;
    getSR(AllSR);

    // 2. 연결 관계 및 처음 파악
    cout << "2. 연결 관계 및 처음 파악" << endl;
    First = findConnect(AllSR);

    // 3. 복원
    cout << "3. 복원" << endl;
    DE_NOVO(First);


}
```

[그림 1]

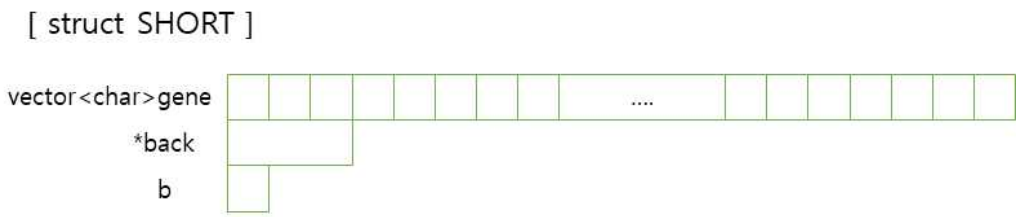
원래 De novo알고리즘은 [overlap] - [layout] - [consensus] 단계로 이루어져 있다. 코드에서는 [overlap]전에 short read를 불러오는 과정이 추가되어 있고 [overlap]단계와 [layout] 단계가 합쳐져 있다고 생각하고 봐야한다.

1. getSR(AllSR); 부분은 모든 short read 정보를 가져오는 부분이다.
2. First= findConnect(AllSR); 부분은 short read간의 연결 관계를 찾고 시작 short read를 반환하는 부분이다. Overlap 단계와 Layout 단계가 합쳐져 있다.
3. DE_NOVO(First); 부분은 시작 short read의 정보를 가지고 모든 DNA정보를 알아내는 부분이다. Consensus 단계가 여기에 포함된다.

1 단계 : 모든 short read 정보 load

```
void getSR(struct SHORT *AllSR) {
    char c;
    ifstream in("SR.txt");

    // 모든 SR에 대한 정보 저장
    for (int i = 0; i < ReadN; i++) {
        // short read DNA 정보 저장
        for (int j = 0; in.get(c) && c != '\n'; j++) {
            AllSR[i].gene.push_back(c);
        }
        AllSR[i].back = NULL; // NULL로 초기화
        AllSR[i].b = 0; // 0으로 초기화
    }

    in.close();
}
```

[그림 2]



[그림 3]

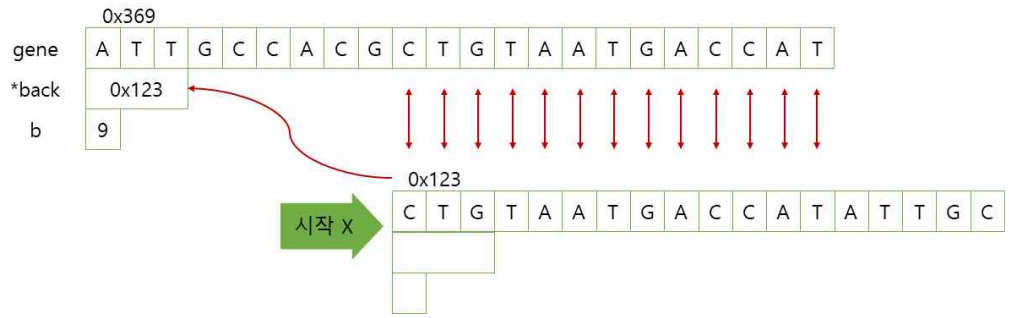
[그림 2]은 모든 short read의 정보를 가져오는 코드 부분이다. SR.txt파일에 저장되어 있는 short read들을 [그림 3]에 나와있는 struct 구조에 저장한다. 이때 short read길이가 달라지는 경우를 고려해 벡터를 사용한다. *back는 다음에 연결되는 struct SHORT의 주소를 저장하는 변수이며, b는 연결되는 시작 인덱스를 저장하는 변수이다. back과 b은 'NULL'과 0으로 초기화 한다.

2 단계 : 연결 관계 및 시작 파악

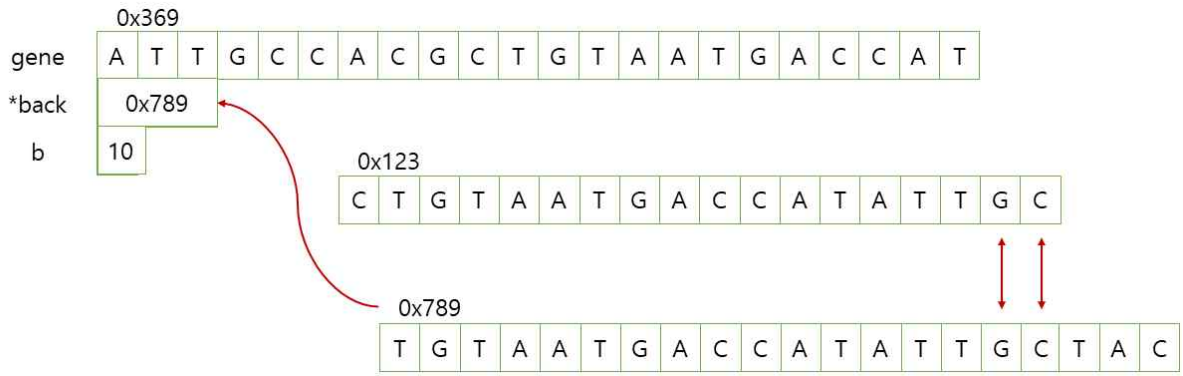
● 전제 조건

- 뒤에 오는 short read가 앞에 오는 short read길이의 0.25 ~0.75 사이에서 겹치기 시작해야한다.
- 뒤에 연결되는 short read가 여러 개 나올 수 있다.
- 어떤 short read의 뒤에 연결되면 해당 short read는 시작 지점이 될 수 없다.

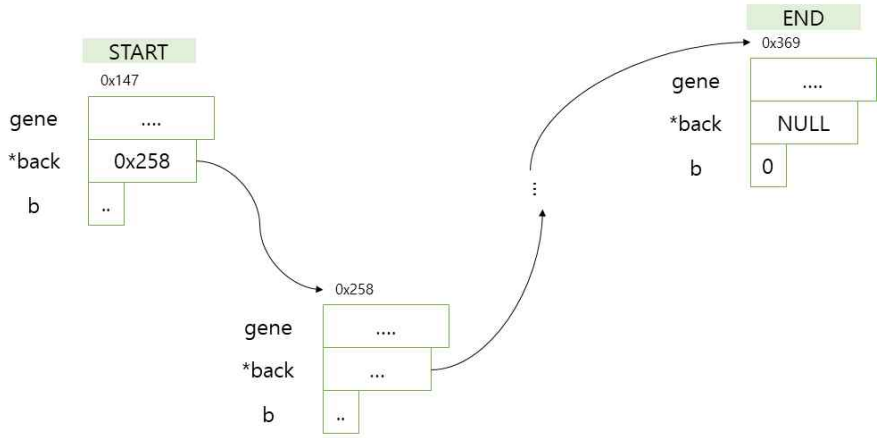
● 알고리즘



[그림 4]



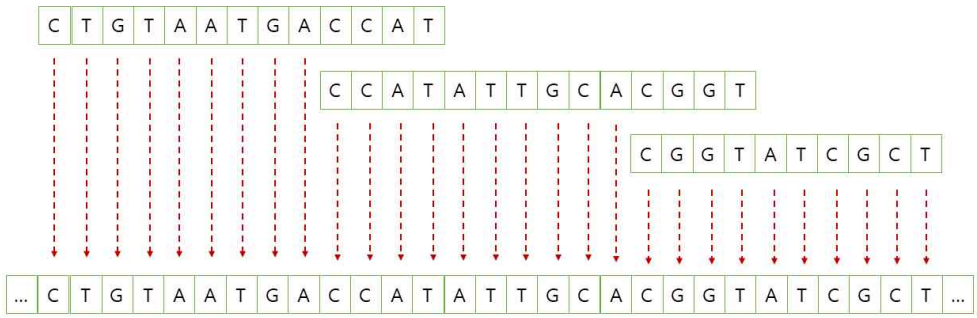
[그림 5]



[그림 6]

1. 모든 short read를 자신을 제외한 모든 short read와 비교한다.
2. 앞에 오는 short read의 0.25 ~ 0.75 사이를 시작점으로 같은지 모두 비교한다.
3. 먼저 겹치는 부분의 처음이 같은지 확인하고 다른 경우 비교를 끝낸다.
4. 처음이 같다면 겹치는 부분의 마지막 부분을 확인하고 다른 경우 비교를 끝낸다.
5. 마지막도 같다면 처음+1 ~ 마지막 -1 부분을 비교한다. 앞에서부터 비교를 시작해 중간에 다른 부분이 있으면 비교를 끝낸다.
6. 겹치는 부분이 일치하는 경우 앞 short read의 back이 처음인 경우 바로 해당 short read정보를 입력한다. [그림 4] 참고
7. 처음이 아닌 경우 연결되어있는 oldB과 새로운 newB를 비교한다. 먼저 두 개의 short read가 연결될 short read보다 밖으로 나와 있는 정도를 비교한다. oldB가 더 긴 경우 다음으로 넘어간다.
8. newB가 더 긴 경우 oldB의 빠져나온 부분과 겹치는 부분을 검사하여 일치하면 newB로 이어지는 short read를 갱신하고 ([그림 5] 참고)불일치하면 아닌 것으로 판단하여 넘어간다.
9. 모든 short read에 대해 2~8번 과정을 반복한다.
10. 9번 과정이 끝나면 아무 노드에도 연결되지 않은 short read가 표시되어있으므로 해당 부분을 찾아 반환한다. ([그림 6] 참고)

3 단계 : DNA 생성



[그림 7]

2단계에서 생성한 연결 관계를 이용해 DNA를 완성한다. 0~b-1까지 파일에 입력하고 연결된 short read정보로 이동해 해당 과정을 반복한다. 마지막 short read인 경우 뒤에 연결되는 것이 없어 b=0인 상태이므로 b를 해당 short read의 길이로 설정해 마지막까지 모두 입력되도록 한다. ([그림 7] 참고)

3-2. 알고리즘 취약점

- [(short read 길이 + 포인터 + int변수) * short read 개수] 만큼의 많은 메모리가 사용된다.
- 중간에 끊기는 경우가 발생하면 복원률이 급격하게 줄어든다.
- 0.25보다 앞에서부터 겹치는 경우 겹친다고 판별되지 않는다.
- short read의 개수가 적어 겹치는 부분이 0.75보다 뒤에서 시작하면 겹친다고 판별되지 않는다.
- DNA길이와 short read의 길이, 개수가 증가하면 걸리는 시간이 단순 몇 배가 아니라 제곱배로 증가한다.
- 겹치는 경우가 많을수록 비교횟수가 늘어 오래 걸린다.

3-3 시간 복잡도

(N : read 개수 / M : read길이 / 상수 k: read개수-2~0 / 상수 m : 겹치는 길이)
1. short read정보 가져오기 = bigO(NM)
2. 연결 관계 및 시작 부분 파악
= (최악의 경우 : 모두 일치하는 경우 && 시작이 맨 뒤 short read인 경우)
= N * (N -1)* (M*0.5)*(N -2)* (N -k)*(m) + N
= bigO(N^3M)
3. DNA 생성 = (최악의 경우 : 모든 dna정보가 모두 있는 경우) DNA 길이

4. REFERENCE DNA 사용 하는 알고리즘

- KMP 알고리즘

KMP 진행 순서
1. read를 읽어온다.
2. read의 최대 접두부 테이블(SP)를 만든다. O(L), L=read 길이
3. kmp를 수행하여 ref를 복원한다. O(L+DL), DL=DNA길이
4. 1~3 과정을 모든 read에 대해 수행한다. O(ReadN*(L+DL)), ReadN=read 개수
5. 복원된 DNA인 ref 배열을 kmpMyDNA.txt에 쓴다.

```
// ref 읽어오기
cout << "getRF함수에서 ref 배열에 refDNA 저장" << endl;
getRF(ref);

cout << "read 읽어와서, ref 배열에 반영" << endl;
// SR.txt에서 read 매번 읽어서, kmp로 복원
short SP[ReadMax] = { 0 };
```

[그림 1]

[그림 1]에서는 getRF함수를 이용하여 ref 배열에 refDNA를 저장한다.
KMP 진행순서 2번에서 SP를 만들 때 사용할 SP 배열을 선언한다. read 길이는 랜덤하게 생성되더라도, 최대 ReadMax이므로 ReadMax 사이즈로 고정된 크기의 배열을 생성한다. 메모리 차지를 조금이라도 줄이기 위해 short 배열로 선언하였으며, 배열 내의 모든 원소를 0으로 초기화하였다.

```
// 1) read 읽어오기
readSR.get(c); int r = 0; L = 0;
while (c != '₩n')
{
    shortreads[r] = c;
    if (s % 100 == 0)
        cout << shortreads[r];
    r++; L++;
    readSR.get(c); // 다음 글자 읽어오기
}
if (s % 100 == 0)
    cout << endl;
```

[그림 2]

[그림 2]에서는 SR.txt 파일에서 \n이 나오기 전까지 문자 1개씩 읽어서 read 1개를 shortreads 배열에 저장하는 부분이다.

```
// 2) read의 최대 접두부 테이블 만들기
for (int sp = 0; sp < L; sp++)
    SP[sp] = 0;
int i, j = 0;
// 최대 접두부 테이블 생성 for문
for (i = 1; i < L; i++) {
    while (j > 0 && shortreads[j] != shortreads[i])
        j = SP[j - 1];
    if (shortreads[j] == shortreads[i])
        SP[i] = ++j;
}
```

[그림 3]

[그림 3]에서는 read의 최대 접두부 테이블을 만드는 부분이다.
1) SP를 만들 때는 shortreads[j]!= shortreads[i]일 때 최악의 경우에는 일치하는 문자가 하나도 없어서 0을 SP에 저장한다. 따라서, 매번 read의 최대 접두부 테이블을 만들기 전에 read 길이만큼 SP 테이블을 0으로 초기화해준다.
2) SP 테이블을 만들 때는 mismatch 없으므로, mismatch를 고려하지 않고 SP 테이블을 생성하면 된다.
- while (j > 0 && shortreads[j] != shortreads[i]) j = SP[j - 1]; 부분은 불일치 발생 직전의 SP테이블의 값만큼 j를 이동하여, shortreads[i]와 일치하는 위치를 찾는다.
- if (shortreads[j] == shortreads[i]) SP[i] = ++j; 부분은 SP[i]에 j를 하나 증가시켜 대입한다.
- 이 같은 방식으로 i<L까지 접두부 테이블을 만든다.

```
// 3) kmp 일치 하는 부분 찾는 for문 & while문
miss = 0;
t = 0;
tmpM = 0; m = 0;
for (t = 0; t < DL - L + 1; t) {
    if (ref[t].check == 'T')
        t = t + ReadMin;
    while (m < L) {
        if (ref[t + m].gene != shortreads[m]) {
            miss++;
            // 처음부터 다른 경우
            //if (j == 0) i++;
            if (miss == 1) tmpM = m;

            if (miss < mismatch) {
                m++;
            }
            else {
                if (miss == mismatch && m == L - 1) {
                    m++;
                    break;
                }
                miss = 0;
                if (tmpM == 0) {
                    // t = t - m + 1;
                    t++;
                    m = 0;
                }
                else {
                    t = t + tmpM;
                    m = SP[tmpM - 1];
                }
                break;
            }
            else {
                m++; // 문자가 일치할때
            }
        }
        // 일치
        if (m == L && miss <= mismatch && ref[t].check != 'T') {
            ref[t].check = 'T';
            for (int z = 0; z < L; z++)
            {
                // 이미 ref에 값이 있으므로, 다른 곳만 대입
                if (ref[t + z].gene != shortreads[z])
                    ref[t + z].gene = shortreads[z];
            }
            break;
        }
    }

    if (s % 100 == 0)
        cout << s << " 번째 read로 KMP복구 중" << endl;
}
```

[그림 4]

KMP 원리
DL-L까지 ref 인덱스(t)를 이동하면서 문자열을 복원한다. (for문 사용)
1. 방문한 위치라면 t를 ReadMin만큼 움직인다.
- 왜냐하면, 최소 read 길이만큼은 비교를 ref에 복원을 했기 때문이다.
2. while문을 사용하여, t의 인덱스는 변화하지 않고, m이 변화하면서, ref[t+m]과 shortreads[m]를 비교한다.
1) ref[t+m]!= shortreads[m]인 경우
 - miss 개수 1 증가
 - 처음 miss가 발생한 shortreads의 인덱스를 tmpM에 저장한다.
 - miss가 mismatch 허용 개수보다 작으면, m++;을 한다.
 - miss>=mismatch인 경우에서 while문을 나가야 한다.
    - miss == mismatch && m == L - 1 이면, 문자열이 매칭된다는 뜻이므로, m++을 하고 break;하여 while문을 나간다.
    - miss가 mismatch 허용 개수를 초과하므로, miss=0으로 초기화하고, 다시 비교를 시작해야 한다.
        - tmpM==0인 경우: shortreads의 처음부터 miss가 발생했으므로, t의 다음 부분부터 비교를 시작해야 한다. m도 m=0;으로 초기화해야 한다.
        - tmpM!=0인 경우: SP 테이블을 이용한다. t는 처음 mismatch가 발생한 인덱스를 더하여 비교를 진행한다. m도 처음 mismatch 발생 전 인덱스의 SP테이블 값으로 이동
        - break 한다.
2) 문자가 일치하면, m만 다음 인덱스로 넘어간다.
3. m==L이고, mismatch를 넘지 않으면서, 아직 방문하지 않은 ref[t]라면, 복원한다.
- 방문 표시를 한다.
- ref와 다른 곳만 ref에 복원해준다.
- 복원이 완료되면, 전체 for문을 break; 해서 나가고, 다음 read를 갖고 와서 위 과정을 반복한다.

```
readSR.close();

ofstream writekmp("kmpMyDNA.txt");
if (writekmp.is_open())
{
    for (int w = 0; w < DL; w++)
        writekmp << ref[w].gene;
}
writekmp.close();
```

[그림 5]

[그림 5]는 SR.txt에서 read를 모두 읽어서 trivial 복원 후, ifstream을 닫아주는 모습이다. 그리고 복원된 ref 배열을 kmpMyDNA.txt에 쓰는 부분이다. ofstream을 열어서 파일을 쓰고, close한다.

- KMP 알고리즘 취약점

- check를 사용하여, 한번 복원됐으면 더 이상 복원 못하게 하는 기능 때문에, 복원된 곳이 잘못된 곳일 수 있다.

- KMP 공간, 시간 복잡도

(DL : DNA길이 / L : short read의 길이 /ReadN : short read 개수) - 공간 복잡도 : O(2L) ->SP 테이블 사용 - 시간 복잡도 : - 일반적인 경우 : O(L+DL) - 최악의 경우 : O(DL*L)

- BWT 알고리즘

BWT 알고리즘은 크게 3가지 단계로 이루어져 있다.
1. Generate all suffix & fill the rest
2. Sort – Heap sort O(nlog n)
3. Short read matching (mismatching)

비교적 시간복잡도가 작은 heap sorting을 사용하여 수행 시간을 단축하였다.

Generate all suffix & fill the rest
1. 주어진 reference dna에 $를 더해준다.
2. 맨 뒤($)에서부터 reference dna를 잘라 suffix를 생성한다.
3. reference dna의 처음시작부터 suffix 시작 전까지 더해줘 fill the rest를 수행한다.
4. 첫 번째 문자(first_char)의 A, C, G, T 개수 count

이때 생성한 string은 위와 같은 struct 구조를 사용해 저장해 주었다.
dna_st에는 생성한 string전체를 저장하였고, last_index에는 dna_st의 마지막 문자의 원래 reference 위치를 저장한다. 이때, dna_st의 첫 번째 문자의 원래 reference의 위치를 나타내는 first_index는 last_index와 1씩 차이가 난다. (이후 short read 매칭에서 사용됨)

```
struct bwt_string {
    string dna_st;
    char first_char;
    int last_index;//마지막 문자의 원래 T에서의
};
```

```
int cnt = 0;// fill the rest 배열 순서
int index = 0;//string 읽어오는 순서
char* tmp = new char[DL + 2];// bwt_sub string의 tmp 변수
for (int i = length - 1; i >= 0; i--) //fill the rest 배열 생성
{
    memset(tmp, 0, DL + 2);

    for (int j = i; j < length; j++) { tmp[index++] = ref[j]; }//suffix
    for (int j = 0; j < i; j++) { tmp[index++] = ref[j]; }//prefix
    bwt_sub[cnt].dna_st = tmp;//전체 string
    bwt_sub[cnt].first_char = tmp[0];//맨 처음 char
    if (bwt_sub[cnt].first_char == 'A') //처음 dna 개수 count (A,C,G만 count T는 전체 길이에서 A+C+G뺀것(사용 x))
        A_cnt++;//A면
    else if (bwt_sub[cnt].first_char == 'C')//처음 dna가 C면
        C_cnt++;
    else if (bwt_sub[cnt
        G_cnt++;                                    /처음 dna G면

    bwt_sub[cnt].last_index = (i + length - 1) % length;//마지막 원소 원래 reference 의 위치
    cnt++;
    index = 0;//초기화

}
delete[] tmp;//tmp 제거

cout << "fill_the_rest 완료" << cnt << endl;
```

int cnt
fill the rest 배열 순서

Heap sort
sort 중에 시간복잡도가 O(nlogn) 으로 낮아 사용하였다.
1. 최대힙 생성
2. heap sort 진행 -> 스트링 전체 비교가 아닌 compare_text 함수를 사용해 문자 하나씩 비교해 더 빠른 판단이 가능하다.

```
void make_heap_tree(bwt_string *bwt_sub, int root, int lastnode)//힙생성
{
    int parent = root;
    bwt_string rootvalue = bwt_sub[root];
    int left = 2 * parent + 1; int right = left + 1;
    int son;
    while (left <= lastnode)//마지막 노드보다 작으면
    {
        if (right <= lastnode && compare_text(bwt_sub[left].dna_st, bwt_sub[right].dna_st) == 1)//오른쪽이 왼쪽보다 크면
            son = right;
        else son = left;

        if (compare_text(rootvalue.dna_st, bwt_sub[son].dna_st) == 1) {
            swap(bwt_sub[parent], bwt_sub[son]);
            parent = son;
            left = parent * 2 + 1;
            right = left + 1;
        }
        else break;
    }

}
void heap_sort(bwt_string *bwt_sub, int length)//heap sort
{
    for (int i = length / 2; i >= 0; i--) { make_heap_tree(bwt_sub, i, length - 1); }//최대 heap 생성
    cout << "heap 생성" << endl;
    for (int i = length - 1; i > 0; i--)//sort
    {
        swap(bwt_sub[0], bwt_sub[i]);
        make_heap_tree(bwt_sub, 0, i - 1);
    }
}
```

```
int compare_text(string parent, string child)// string원소 한개씩 비교
{
    int i = 0;

    if (parent[0] == '$')  return 1;
    if (child[0] == '$') return  2;
    while (1)
    {
        if (parent[i] < child[i])
            return 1;
        else if (parent[i] > child[i])
            return 2;
        i++;
    }
    return 0;
}
```

Short read matching

1. Short read를 한 개씩 호출해 온다
2. find 함수를 사용해 short read의 마지막 문자와 bwt_string struct의 first_char이 일치하는 시작 위치를 찾아준다. (short read는 뒤에서부터 탐색한다)
3. last_index를 활용해 다음 위치를 찾는다.
   이때, 해당 위치의 last_index = 찾으려는 위치의 (last_index+1)% DL+1 인 곳으로 연쇄적으로 이동한다. 항상 read와 reference의 일치여부를 확인해 miss에 저장한다.
4. miss> match 경우, 2번 find에서 얻은 first 위치에서 +1 을 해주고 3번 과정을 반복한다. 즉, 다음 인덱스를 시작으로 한다.
5. miss< mastch 고 모든 read를 탐색했다면 3번에서 마지막으로 끝난 곳의 (last_index+1) %DL이 reference에서의 read의 시작 위치이다. matching 성공으로 reference와 read 가 일치하지 않은 부분을 수정해준다.

```
find(L - 1, first, A_cnt, C_cnt, G_cnt);// short read의 마지막 원소의 bwt_sub에서 첫번째 위치 찾기

if (bwt_sub[first].first_char == shortreads[L - 1])//일치 확인
{
    next = bwt_sub[first].last_index;//다음 탐색 last_index가져오기
    for (int j = L - 2; j >= 0; j--)// read j번째 매칭
    {
        if (miss > mismatch) {//mismatch개수를 넘기면 |
            first++; //처음 위치 한칸 앞으로
            if (bwt_sub[first%length].first_char == shortreads[L - 1]) {//처음 dna와 일치시 초기화
                next = bwt_sub[first%length].last_index;//다음 탐색 last_index가져오기
                miss = 0;//초기화
                j = L - 2; //일치하는지 확인 --> L-2부터 다시 비교
            }
            else { break; }// 실패

        }


        for (int x = 0; x < length; x++)// read 에서 i번째 문자 매칭
        {
            if ((bwt_sub[x].last_index+1)%length == next) {//next와 first index와 일치시
                if (bwt_sub[x].first_char != shortreads[j]) miss++;// dna mismatch 확인
                next = bwt_sub[x].last_index%length;//다음 index넣어주기
                break;
            }

        }
        if (j == 0) {//short read 위치 찾기 성공
            next += 1; //first index로변경 == patter시작 위치
            next %= length;
            for (int y = 0; y < L; y++)
            {
                if (ref[(next + y) % length] != shortreads[y])
                    ref[(next + y) % length] = shortreads[y];
            }
        }//성공
    }
```

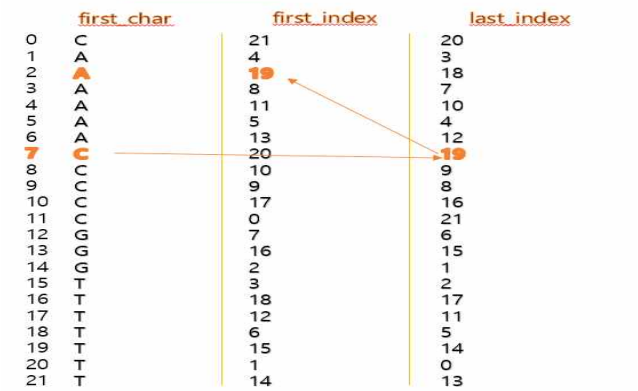
```
void find(int i, int &first, int A_cnt, int C_cnt, int G_cnt)
{

    char check = shortreads[i];
    if (check == 'A')
    {
        first = 1;
    }
    else if (check == 'C')
    {
        first = A_cnt + 1;
    }
    else if (check == 'G')
    {
        first = A_cnt + C_cnt + 1;
    }
    else if (check == 'T')
    {
        first = A_cnt + C_cnt + G_cnt + 1;
    }

}
```

1,2 단계를 마친후 결과이다. first_index는 실제 저장하는 값은 아니다. matching의 설명을 위해 표현하였다. 아래는 reference를 CTGTAATGACCATATTGCTAC로 했을 때 결과이다. A_cnt=6 ,C_cnt=5 ,G_cnt=3이다.

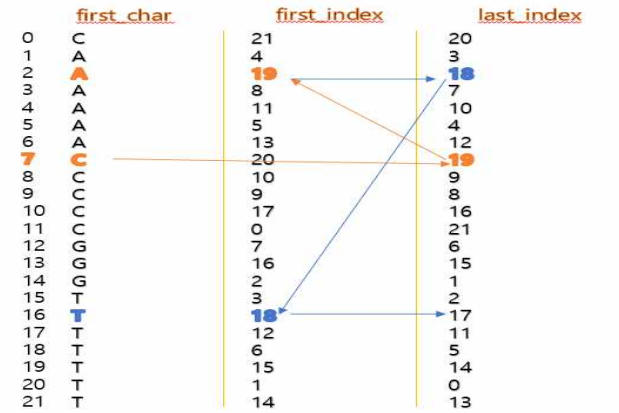
BWT_String	first_index	String dna_st	last_index	first_char
0	21	$CTGTAATGACCATATTGCTAC	20	C
1	4	AATGACCATATTGCTAC$CTGT	3	A
2	19	AC$CTGTAATGACCATATTGCT	18	A
3	8	ACCATATTGCTAC$CTGTAATG	7	A
4	11	ATATTGCTAC$CTGTAATGACC	10	A
5	5	ATGACCATATTGCTAC$CTGTA	4	A
6	13	ATTGCTAC$CTGTAATGACCAT	12	A
7	20	C$CTGTAATGACCATATTGCTA	19	C
8	10	CATATTGCTAC$CTGTAATGAC	9	C
9	9	CCATATTGCTAC$CTGTAATGA	8	C
10	17	CTAC$CTGTAATGACCATATTG	16	C
11	0	CTGTAATGACCATATTGCTAC$	21	C
12	7	GACCATATTGCTAC$CTGTAAT	6	G
13	16	GCTAC$CTGTAATGACCATATT	15	G
14	2	GTAATGACCATATTGCTAC$CT	1	G
15	3	TAATGACCATATTGCTAC$CTG	2	T
16	18	TAC$CTGTAATGACCATATTGC	17	T
17	12	TATTGCTAC$CTGTAATGACCA	11	T
18	6	TGACCATATTGCTAC$CTGTAA	5	T
19	15	TGCTAC$CTGTAATGACCATAT	14	T
20	1	TGTAATGACCATATTGCTAC$C	0	T
21	14	TTGCTAC$CTGTAATGACCATA	13	T

Read가 CGAC 일 경우

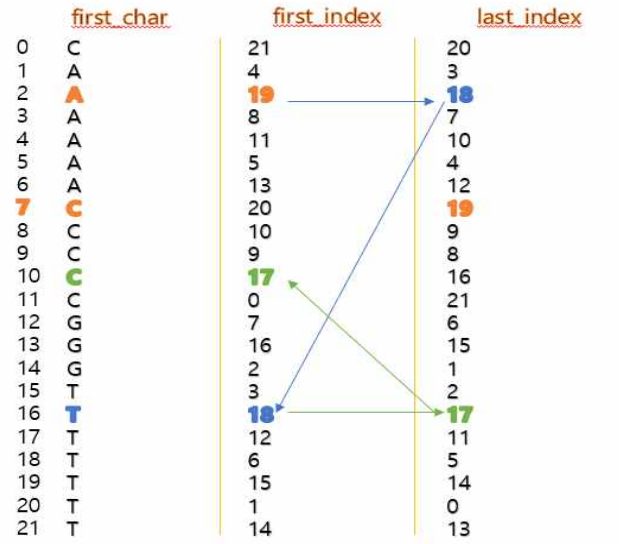
1. 마지막 문자인 C를 find 함수에 넣으면 first=7 된다. 7부터 탐색을 시작한다.
2. last_index = first_index 일 때 연결 해 나간다. (first_ index는 실제 저장된 값이 아니라 last_index에 1을 더한 값)
3. first_char과 short의 문자 일치 여부 확인



C시작 --> A일치 확인 miss=0



T != G 불일치 miss++ miss=1



C=C 일치 miss=1 로 종료해 matching을 성공하였다. 이때 first_index의 위치가 reference에서의 short read의 시작 위치를 나타낸다. 따라서 reference의 17번째부터 short read와 불일치 하는 부분을 수정하면 CTGTAATGACCATATTGC**G**AC 가 된다.
만약에 처음 first에서 성공하지 못한다면 8번부터 시작해 위 과정을 반복한다.

BWT 알고리즘 취약점

- [reference Dna 길이] ^ 2 만큼의 많은 메모리가 사용된다.
- 컴퓨터 사양에 따라 가능한 최대 dna 길이가 차이가 난다.
- short read의 마지막에 snp이 존재한다면 matching에 실패한다.

BWT 공간, 시간 복잡도

(N : reference 길이 / M : read 개수 / m : read의 길이 / 상수 m : 겹치는 길이)
1. 공간 복잡도 : (N+1)(N+5)=N^2
2. 시간 복잡도 :
   heap sort : O(nlogn)
   matching : (M*m)*N = O(MN)
--> O(MN)

- BOYERMOORE 알고리즘

```
//bad character 테이블을 만듦
int* createBC(char* pat, int M)
{
    int* bc = new int[NUM_OF_CHARS];

    // 테이블을 -1로 초기화(패턴에 존재하지 않는 문자는 모두 -1)
    for (int i = 0; i < NUM_OF_CHARS; ++i)
        bc[i] = -1;

    //패턴에 존재하는 캐릭터의 마지막 위치를 bc테이블에 넣음
    for (int i = 0; i < M - 1; ++i)
        bc[(int)pat[i]] = i; //캐릭터의 int형인 숫자의 배열에 위치 삽입

    return bc;
}
```

[그림 1]

Boyer-Moore알고리즘은 상용화된 서비스에서 가장 많이 적용되는 알고리즘으로, 일반 스트링 매칭 알고리즘이 앞에서부터 비교하는 것과 달리 boyer-moore는 패턴의 뒤에서부터 비교한다. 이는 문자열 매칭이 앞보다 끝에서 불일치할 확률이 더 높기 때문이다. 보통 boyer-moore는 나쁜 문자 이동방법과 착한 접미부 이동방법을 함께 적용하여, 이 둘 중 더 많이 이동하는 함수를 택하여 이동한다. 하지만 이 프로젝트에선 둘 다 적용해본바, 나쁜 문자 이동방법만을 사용했던 것과 달리 착한 접미부 이동방법을 같이 적용하면 복원률이 80%이하로 떨어지기 때문에 본 프로젝트에서는 나쁜 문자 이동방법만 사용할 것이다.

나쁜 문자 이동방법은 불일치가 발생했을 때, 텍스트의 나쁜 문자가 패턴에 존재한다면 그 위치로 패턴을 이동시키고, 존재하지 않다면 패턴의 길이만큼 이동하는 것이다. 이때, 나쁜 문자는 매칭 동안 텍스트의 문자 중 패턴의 문자와 일치하지 않는 문자를 말한다.

boyer-moore는 bc(bad character)테이블을 생성하는 creatBC()함수와 bc테이블을 사용하여 매칭을 수행하는 BoyerMoore()함수로 구성되어 있다.

[그림 1]은 bc테이블을 만드는 함수이다. 먼저 bc배열을 NUM_OF_CHARS만큼 동적할당으로 생성한다. 여기서 NUM_OF_CHARS는 1바이트에서 담을 수 있는 수의 개수로 각 문자의 숫자 위치를 뜻한다. bc배열을 먼저 -1로 초기화한다. (따라서 패턴에 존재하지 않는 문자는 모두 -1이 된다.) 그 후 패턴에 존재하는 캐릭터의 마지막 위치를 테이블에 넣는다. 이렇게 만들어진 bc테이블을 리턴하며 끝난다.

```
while (x < SR_num) //short read 개수만큼 반복
{
    fgets(buf, ReadMax + 2, file); //file에서 한줄읽어옴
    pattern = buf;
    l = pattern.length() - 1; // l: 패턴길이
    bc = createBC(buf, l); //패턴의 bc테이블을 만듦
    i = 0;
    while (i <= n - l)
    {
        miss = 0;
        //보이어 무어 알고리즘은 뒤에서부터 접근하므로 pattern길이의 - 1을 해준다.
        //-1을 해주는 이유는 인덱스가 0부터 시작하기 때문
        j = l - 1;
        //뒤에서부터 검사하고 인덱스를 감소하는 형식이므로 j가 0보다 이상일때
        while (j >= 0)
        {
            //패턴과 텍스트가 같지 않다면
            if (pattern[j] != text[i + j])
            {
                miss++; //miss 증가
                //miss개수가 mismatch(허용 개수)보다 크다면
                if (miss > mismatch)
                {
                    //텍스트의 bad character가 패턴에 존재한다면 그 위치로 패턴을 이동시킴.
                    //bad character의 위치가 패턴의 현재 비교 위치보다 뒤에 있을 수 있는데
                    //이럴 경우, j - bc[text[i + j]]가 음수가 될 수 있으므로
                    //이때는 max를 이용하여 1만큼 증가시킨다.
                    i += max(1, j - bc[text[i + j]]);
                    break;
                }
            }
            j = j - 1;
        }
        //miss개수가 mismatch(허용 개수) 이하일때 매칭이 된 것이므로
        if (miss <= mismatch)
        {
            //끝글자부터 비교하면서 하나씩 감소하면서 비교한다.
            for (j = l - 1; j >= 0; j--) //for문을 돌려 하나씩 비교
            {
                //동일하지않으면 그 부분이 SNP이므로 text(referenceDNA)부분을 pattern(shortRead)으로 바꿈
                if (pattern[j] != text[i + j])
                {
                    cout << "바뀜" << text[i + j] << "에서" << pattern[j] << "로 바뀜" << endl;
                    text[i + j] = pattern[j];
                }
            }
            break; // for문 나가서 나머지 short read 수행
```

[그림 2]

[그림 2]는 BoyerMoore를 수행하여 DNA를 복원하는 부분이다. 먼저 fopen을 통해 shortread파일을 열고 위 과정을 수행한다.
fgets()함수를 통해 파일에서 한 줄(shortread)을 읽어와서 저장하고 createBC()함수를 호출하여 그 short read(패턴)의 bc테이블을 만든다. 그 다음에 DNA길이동안 그 short read의 위치를 찾는 과정을 반복한다.
보이어무어 알고리즘은 뒤에서부터 비교하므로 j=l-1로 해주고, 패턴과 텍스트를 비교한다. 만약 같지 않으면 miss++를 해주며, miss가 최대 허용개수인 mismatch를 넘어서면 그 자리를 나쁜 문자로 하여 아까 만들어둔 bc테이블을 이용하여 이동할 위치를 찾는다. 이때 텍스트의 나쁜 문자가 패턴에 존재한다면 j - bc[text[i + j]]를 통하여 그 위치로 패턴을 이동시키고, 이와 반대로 나쁜 문자가 패턴의 현재 비교 위치보다 뒤에 있을 수 있는데 이럴 경우 j - bc[text[i + j]]가 음수가 될 수 있으므로 max를 이용하여 1만큼 증가시킨다.
j가 0이되어 while문이 끝나면 매칭이 된 것이므로, if문을 통해 miss개수가 mismatch보다 작으면 for문을 통해 패턴과 텍스트의 다른 부분(SNP)을 찾아서 텍스트를 패턴의 염기로 바꿔준다. 이 과정이 DNA복원 과정이다. 이 과정들을 shortread개수만큼 수행하면 복원이 완료되게 된다.

```
    FILE* dna_file = fopen("boyermooreMyDNA.txt", "wt");
    if (dna_file != NULL)
    {
        for (int i = 0; i < DL; i++)
        {
            fprintf(dna_file, "%c", text[i]);
        }
    }
    free(buf);
    fclose(dna_file);
    delete[] bc;
}
```

[그림 3]

[그림 3]은 복원된 dna(text)배열을 복원 파일에 저장하는 과정이다.
전체 복원이 완료되면, 위에서 바꾼 text배열을 fopen을 통해 복원 파일을 만들어 그 파일에 저장하면 된다.

BOYER-MOORE 알고리즘 취약점

- 패턴의 마지막에 SNP이 존재한다면 matching에 실패한다.

BOYER-MOORE 공간, 시간 복잡도

(N : reference 길이 / M : short read의 길이)
- 공간 복잡도 : O(2M)
- 시간 복잡도 :
  - 일반적인 경우 : O(N)이하
  - 최악의 경우 : O(MN)

5. 결과 분석

- DNA 길이에 따른 알고리즘 성능 분석 (복원율/실행시간)

실행 환경

: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80GHz

기본 속도

: 1.80GHz

DNA 길이	trivial	kmp	denovo	boyermoore	bwt
1080	0.007 / 100%	0.011 / 94.35%	0.062 / 100%	0.002 / 100%	0.041 / 99.44%
5400	0.017 / 100%	0.04 / 93.79%	0.326 / 100%	0.023 / 100%	1.712 / 99.51%
10800	0.042 / 99.93%	0.107 / 93.94%	0.536 / 100%	0.094 / 99.93%	14.18 / 99.55%
32400	0.293 / 99.97%	0.666 / 94.08%	1.577 / 100%	0.547 / 99.96%	545.7 / 99.42%
43200	0.504 / 99.96%	1.22 / 93.94%	2.659 / 100%	1.009 / 99.96%	1324.6 / 99.24%
108000	2.918 / 99.94%	6.786 / 94.1%	7.975 / 100%	5.972 / 99.93%	-
540000	65.02 / 99.97%	168.9 / 94.13%	64.09 / 100%	158.6 / 99.95%	-
1080000	273.8 / 99.96%	681.2 / 94.18%	216.5 / 100%	622.1 / 99.94%	-

- trivial VS 각 알고리즘

위 결과 표를 보면 trivial이 다른 알고리즘보다 더 빠른 것을 알 수 있다.
이는 trivial에서 check라는 변수를 이용하여 이미 check한 곳이면 다음으로 넘어가기 때문이다. 원래 기본 trivial은 check라는 변수가 없기 때문에 기본 trivial과 비교한다면 다른 알고리즘들의 시간이 더 빠르게 나올 것이다.

- kmp VS boyermoore

위 결과 표를 보면 전체적으로 BoyerMoore가 KMP보다 복원율이 높은 것을 볼 수 있다. BoyerMoore가 KMP보다 복원율이 더 높은 이유는 KMP는 접두부를 이용하여 이동하는 반면 BoyerMoore는 패턴의 맨 뒤 문자만 비교하여 이동하기 때문이다. 접두부를 사용하게 되면 패턴내에 SNP이 있을 수 있기 때문에 제대로 된 접두부 위치를 찾지 못한다. 따라서 BoyerMoore는 맨 뒤 문자가 SNP가 아니면 되는 것에 비해 KMP는 접두부에 SNP가 없어야 하므로 복원율이 더 낮다.
또한 KMP가 복원율이 BoyerMoore보다 작은 이유 두 번째로는 check를 써서 한번 복원됐으면 더 이상 복원 못하게 하는 부분 때문이다. 복원된 곳이 잘못된 곳일 수 있다.

6. 역할 분담

담당 알고리즘을 정하였으나 서로 협력하며 진행하였다.

김규리	BWT
송승민	KMP
이서연	BOYER MOORE
최수정	DENOVO