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 월 일

<전제 조건>

Reference DNA	길이 : 3000000000 생성 방식 : (길이가 11인 4^11개의 패턴 중에서 랜덤하게 호출해 사용) + (노이즈를 이용한 랜덤 -길이: 61)
Short-read 길이	Random(72, 80) && 정해지는 경우 : 32 / 54 / 76 / 100
Short-read 개수	Reference length/ min(Short-read)
Snp percentage	10 %
Mismatch 개수	노이즈 부분 기준 10 %

<비교 사항>

- Short-read 길이가 랜덤 한 경우와 랜덤 하지 않은 경우(32 / 54 / 76 / 100) 비교
- 보이어 무어 알고리즘과 KMP 알고리즘의 성능 비교
- trivial 알고리즘과 전체 알고리즘 성능 비교

<담당 알고리즘>

김규리	BWT
송승민	KMP
이서연	BOYER MOORE
최수정	DENOVO

< 알고리즘 특이 사항 >

- BWT에서 개선된 sorting을 사용할 것이다
- trivial 은 함께 구현하고, 두 명씩 짝을 지어 알고리즘을 개선해 나갈 예정이다
- trivial과 KMP에서 성능을 높이기 위해, 복원 표시를 할 것이다.
- 패턴을 이용해 short read의 위치를 좀 더 빠르게 찾고, 정확도를 높일 예정이다.
- 현실성을 높이기 위해, Short-read를 자를 때, 랜덤하게 앞으로 갔다 뒤로 갔다 하면서 자를 것이다.

< Struct 사용 >

▶ Struct 배열 사용해 dna & Short read 처음부분에 방문 여부 확인 (Trivial 에서 사용) 방문하지 않은 곳에 매칭이 되면 break

```
struct dna {
        char gene;
        char check; // 방문 여부 T or F
};
```

Denovo	X reference dna정보를 사용하지 않으므로, check 사용 안함
BWT	X 뒤에서 부터 비교해서 사용 못함
KMP	O 모든 read에 대해서 처음부터 패턴 찾기를 하므로, check를 써서, 앞에만 방문하고 멈추는 일을 방지해야 함.
BOYER MOORE	X 뒤에서 부터 비교해서 사용 못함

< 예상되는 문제 >

Denovo	
Bwt	마지막이 snp면 문제가 생김
Kmp	없음.
BOYER MOORE	마지막이 snp면 문제가 생김

< 다음 주 목표 >

- 각자 코드 짜면서 big-o 표현으로 average worst case 계산하기 (실행시간과는 별개)
- Trivial 실행결과

목차

1. 문제 정의

1) 패턴 생성 -규리

2) 랜덤 생성 -수정 **

[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 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.
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 -서연

4) short read 생성 -승민

```
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.

```
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 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 앞으로 인덱스 이동
2) 0~ DL/2 사이에서 my배열의 앞에서부터 k 길이의 read를 자른다.
- 겹치게 자르기 위해, overLap(gen)만큼 뒤로 인덱스 이동
3) DL/4부터 my배열의 앞에서부터 k 길이의 read를 자른다.
- 설정한 read개수를 맞추기 위해, overLap(gen)만큼 뒤로 인덱스를 이동

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

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

[그림 5]

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

2. 벤치마킹 알고리즘 -승민

3. REFERENCE DNA 사용 안하는 알고리즘

- DE NOVO 알고리즘 **

De novo알고리즘은 [overlap] - [layout] - [consensus] 단계로 이루어져 있다.
1. Overlap 단계
-
2. Layout 단계
3. Consensus 단계

4. REFERENCE DNA 사용 하는 알고리즘

- KMP 알고리즘

- BWT 알고리즘

- BOYERMOORE 알고리즘

5. 결과 분석

- DNA 길이에 따른 알고리즘 성능 분석

(복원율/실행시간)

(실행시간과 복원율 각각 4명것 평균)

- trivial VS 각 알고리즘

- kmp VS boyermoore

전체적으로 BoyerMoore가 KMP보다 복원율이 높다. 하지만, kmp는 복원시간이 BoyerMoore보다 짧다는 장점이 있다. DL길이가 증가할수록, BoyerMoore와 KMP 복원 시간 차이가 굉장히 많이 나는 것을 알 수 있다. KMP가 복원시간이 BoyerMoore보다 훨씬 작은 이유로는 크게 2가지가 있다. 첫째, KMP는 최대 접두부 테이블 SP를 사용하여, 접두부가 있는 길이만큼 비교 횟수가 줄어든다. 둘째, KMP는 복원된 위치는 check 표시를 통해, 실행이 되었다면 ReadMin만큼 움직여서 수행 시간이 줄어든다. KMP가 복원율이 BoyerMoore보다 작은 이유로는 check를 써서 한번 복원됐으면 더 이상 복원 못하게 하는 부분 때문이다. 복원된 곳이 잘못된 곳일 수 있다.

6. 역할 분담